ZBasic Language Reference Manual

Including Information on the
ZX Series Microcontrollers

Version 2.2.0

Publication History

November 2005	First publication
December 2005	Minor corrections and updated information
January 2006	Minor corrections and updated information
February 2006	Minor corrections and updated information
May 2006	Added information on structures and other updates
August 2006	Updated for new ZX models
October 2006	Added new information on tasks and memory allocation
January 2007	Minor corrections and updated information
February 2007	Added information on new ZX models
August 2007	Added information on a new ZX model
March 2008	Added information on new ZX models

Table of Contents

The ZBasic Language
and ZX Series Microcontrollers

Chapter 1 - Introduction

The purpose of this manual is to describe the elements of the ZBasic language and the `ZX-series` microcontrollers for those who have, at a minimum, a rudimentary knowledge of programming concepts and an understanding of basic electronics. It is not intended to teach programming skills or electronics. There are many texts available for that purpose.

The ZBasic programming language will be familiar to anyone who has used Microsoft's Visual Basic language or NetMedia's BasicX language. ZBasic was designed to achieve a high level of compatibility with these languages but it also incorporates some additional capabilities to improve its utility in the microcontroller environment. Even if you aren't familiar with either of these two particular dialects of the Basic language, if you have used another fairly modern dialect you will find many familiar elements. In contrast, ZBasic is quite different from archaic dialects of Basic like the PBasic language developed by Parallax for use on the Basic Stamp series of microcontrollers. Even so, moving from PBasic to ZBasic should be a relatively easy and rewarding experience.

ZBasic incorporates the modern block structured programming elements that help make programs easier to write, easier to maintain and more reliable. Moreover, it provides a good range of data types allowing efficient representation of the data items you need in your application. Beyond that, it has facilities to implement multi-tasking systems, allowing complex problems to be solved more easily.

All ZBasic programs comprise, at a minimum, one subroutine. That one required subroutine must be named `Main()` and it must be defined as taking no parameters. Additionally, the `Main()` subroutine must have "public" visibility as opposed to being private to a particular module. The minimal ZBasic program is shown below, in its entirety.

```
' the minimal program
Sub Main()
End Sub
```

This do-nothing program illustrates several points. First is the structure of comments as illustrated on the first line of the program. A comment consists of an apostrophe (sometimes called a single quote mark) and all characters following it all the way to the end of the line. It is not necessary for a comment to be on a line by itself. Characters occurring before the apostrophe are processed as if the comment portion of the line weren't present.

The second point illustrated by the program above is the manner in which subroutines are defined. A subroutine definition begins with the keyword `Sub`. That is followed by at least one space or tab character and then the name of the subroutine. Following the subroutine name is the parameter list, details of which are discussed later in this document. The parameter list is always enclosed in a pair of parentheses which must be present even if there are no parameters as is the case above. All of the statements, if any, between the Sub line and the End Sub line constitute the body of the subroutine.

Here is another program, intended for the ZX-24, that is a bit more complicated.

```
Const redLED as Byte = 25      ' define the pin for the red LED
Const grnLED as Byte = 26      ' define the pin for the green LED

Sub Main()
      ' configure the pins to be outputs, LEDs off
      Call PutPin(redLed, zxOutputHigh)
      Call PutPin(grnLed, zxOutputHigh)
```

```
    ' alternately blink the LEDs
    Do
        ' turn on the red LED for one half second
        Call PutPin(redLed, zxOutputLow)
        Call Delay(0.5)
        Call PutPin(redLed, zxOutputHigh)

        ' turn on the green LED for one half second
        Call PutPin(grnLed, zxOutputLow)
        Call Delay(0.5)
        Call PutPin(grnLed, zxOutputHigh)
    Loop
End Sub
```

The first two lines above define some constant values. It is a good idea to use constant values like this instead of using the value explicitly in many different places. One benefit of doing so is that it makes the program somewhat easier to understand assuming, of course, that the name chosen for the constant is suggestive of its intended use or meaning. A second benefit is that if you later wish to change the value, you can do so easily by modifying it in only one place. This way you'll avoid the common error of changing a constant value in some places but not in others.

It is important to note that alphabetic case is not significant in ZBasic. You may type keywords, variable names, etc. in upper case, lower case or mixed case and it makes no difference. The subroutine `Main()` is the same as the subroutine `main()`. Many programmers use alphabetic case to improve readability. Some even use it to remind them of the visibility attribute of the subroutine, constant, etc. One convention for this is to begin all public names with an upper case letter and begin all private names with a lower case letter. You're free to adopt these conventions or not as you see fit.

The public `Main()` routine is where execution begins when you run a program. That's why every program must have a public subroutine `Main()`. In this program, the first two lines of `Main()` utilize a call to a System Library subroutine that sets the state of an I/O pin on the processor. The first parameter that is provided to the `PutPin()` subroutine is the pin number that we wish to configure and we've used the previously defined constants to do so. The second parameter to the `PutPin()` subroutine is a value that indicates whether the pin should be an output or an input and, additionally, its characteristics. In this case, we've used a built-in constant `zxOutputHigh` which both configures the pin to be an output and sets it to be at a logic high level. The LEDs that are present on the processor are illuminated when the pin to which they are attached is at logic low level. The effect of these first two lines, then, is to make the pins associated with both LEDs outputs and to make sure the LEDs are off.

The next part of the code is the block beginning with `Do` and ending with `Loop`. This construct is useful for continually repeating a sequence of statements. The logic flow is that each of the statements inside the `Do...Loop` construct is executed in turn. When the `Loop` statement is reached, control transfers back to the top of the Do loop and the process repeats. ZBasic has several different kinds of control structures for looping, each with different useful characteristics. There are also control structures to allow you to write statements that are executed only if certain conditions prevail. The set of allowable statements is described in detail later in this manual.

Inside of the Do loop, there are two sequences of statements that perform similar functions. The first three lines of code within the Do loop turn on the red LED for a brief period of time. This is done by first setting the corresponding pin to a logic low, then calling the System Library subroutine to delay for a half second and finally setting the LED pin to a logic high. The second sequence does the same for the green LED.

A subroutine is nothing more than a collection of statements that can be executed by invoking the subroutine name. Although it is possible to implement all of the logic of your application inside the subroutine `Main()`, this is usually ill advised except for fairly simple applications like the example above. It is usually better to decompose your application into logical elements and to implement the functionality of each element using a subroutine or a function. The difference between a subroutine and a function is that a function returns a value. Because of this, a function can be used wherever a value may be used.

A subroutine is invoked by using the `Call` statement that specifies the name of the subroutine to invoke and, optionally, the parameters to pass to the subroutine. A function is invoked by using its name in place of a value (directly or in an expression) and specifying the parameters, if any, that are to be passed to the function.

This concludes a rather brief introduction to the ZBasic language. The remaining sections of this manual describe in more detail the various elements of ZBasic including data types, variables, constants, statements, subroutines and functions. Also, more advanced topics like multi-tasking, queues, serial communications and others are presented.

1.1 The ZBasic System Library

The ZBasic System Library provides a rich collection of over 150 subroutines and functions that you can use to quickly add functionality to your application. The routines fall into several fundamental categories including mathematical functions, string-oriented routines, I/O-related routines, type conversion functions, etc. The routines are fully documented in the ZBasic System Library Reference Manual.

1.2 The ZX- Series Microcontrollers

The ZBasic language was designed to be well suited for programming microcontrollers. It is based on a subset of Microsoft's popular Visual Basic (VB6) with modifications and extensions to address the special needs of microcontroller programming. The ZX- series microcontrollers were designed to run ZBasic programs efficiently. At the time of publication, the ZX- series devices available from Elba Corp. include several models with different capabilities, based on several different CPU types as shown in the table below. The various ZX devices will be referred to collectively in this document as simply ZX unless the context requires a specific reference to a particular model.

Underlying CPU Type for ZX Devices

Device	CPU
ZX-24, ZX-40, ZX-44	mega32
ZX-24a, ZX-40a, ZX-44a	mega644
ZX-24p, ZX-40p, ZX-44p	mega644P
ZX-24n, ZX-40n, ZX-44n	mega644P
ZX-1281	mega1281
ZX-1280	mega1280

The ZX-24, ZX-24a, ZX-24p and ZX-24n are 24-pin DIP format modules that are pin-compatible with the Parallax Basic Stamp and the NetMedia BX-24 microcontrollers. However, many improvements have been made with respect to those pioneering products in order to provide a more powerful and flexible programming platform. Because of its ready-to-use configuration it is an ideal starting point for someone just beginning to use microcontrollers. Yet, it has enough powerful capabilities for experts and advanced users as well. See Appendix B for more detailed information on the ZX-24 series devices including several suggested connection diagrams and detailed descriptions of the function of each pin.

The ZX-40, ZX-40a, ZX-40p and ZX-40n are 40-pin DIP format integrated circuits that are pre-programmed with the ZX control program. However, in order to use them you must add a few external components such as a regulated supply, a crystal, a memory chip and serial interface circuitry. The compensating advantages of the ZX-40/ZX-40a are reduced cost, more design flexibility and availability of more I/O pins. These attributes make the 40-pin ZX well-suited for advanced projects and commercial applications. See Appendix C for more detailed information on the ZX-40/ZX-40a including several suggested connection diagrams.

The ZX-44, ZX-44a, ZX-44p and ZX-44n are 44-pin TQFP format integrated circuits that are pre-programmed with the ZX control program. Similarly, the ZX-1280 and ZX-1281 are 100-pin and 64-pin, respectively, TQFP format integrated circuits that are pre-programmed with the ZX control program. All of these devices are bare CPUs that require some additional support components. Because of their smaller size, they are well suited for larger volume applications or those that require compact size. See Appendix D for more detailed information on the ZX-44/ZX-44a including several suggested connection diagrams.

See Appendix E for more detailed information on the suggested ZX-1281 circuitry and Appendix F for the ZX-1280.

It should be noted that ZBasic is a superset of NetMedia's BasicX language, incorporating many improvements and advanced features. Many, perhaps even most, BasicX programs can be recompiled using the ZBasic compiler and the resulting code should run exactly as it would on the BX-24 microcontroller except for differences due to the fact that the ZX is twice as fast as the BX-24. See Chapter 5 for more information on compatibility issues.

The n-suffix ZX devices run in "native mode" meaning that the ZBasic application is compiled to native object code for the processor. In contrast, the other ZX devices contain a "virtual machine" (commonly referred to as a VM) and ZBasic applications are compiled to instruction codes that are executed by the VM. The primary advantage of the native mode devices is that the application executes faster but the disadvantage is that the resulting program is larger than the corresponding VM mode program.

1.3 Conventions

In this manual, specific examples of things that can appear in your ZBasic program are shown in `Courier` typeface as are descriptions of the syntax of program elements. Otherwise, discussion and general references to program elements will be shown in Arial typeface.

When describing the syntax of a ZBasic language element, sometimes portions of it are optional. This fact is indicated by enclosing the optional portion with left and right square brackets. Literal elements, those that must appear as typed (but without regard to alphabetic case, of course) appear as themselves. Often, it is useful to describe an element generally rather than giving a specific instance. When this is done, the description will be italicized and enclosed with left and right angle brackets and subsequent text will describe the characteristics of the generalized item.

Here is an example that uses these conventions:

```
[Private] Const <name> As <type> = <value>
```

The idea being conveyed is that this program element may begin with the `Private` keyword or it may be omitted, i.e. it is optional. The keywords `Const` and `As` are specified literally as is the equal sign. The name of the constant to be defined, its type and value are all specified using a descriptive word or phrase enclosed in angle brackets. These placeholder components will be explicitly described in the discussion of the program element.

In some instances, there may be more than one optional element that may appear in a given position. In these cases, the vertical bar character is used to indicate that any one element may be used, but not more than one, of course.

```
[Public | Private] Const <name> As <type> = <value>
```

This conveys the idea that the either the keyword `Public` or the keyword `Private` may be present but it is also permissible for neither to be present.

In other cases, there may be a set of elements from which exactly one must be chosen.

```
{Public | Private | Dim} <name> As <type>
```

The curly braces indicate that one of the keywords `Public`, `Private` or `Dim` must precede the `<name>` element.

In various places in this document you will find a text box with information about BasicX compatibility. Most often, the text will describe how some particular feature is different when the BasicX compatibility mode is enabled. In a few instances, the text will describe how ZBasic is fundamentally different from BasicX. This information is most useful to those who are upgrading to ZBasic from BasicX. If that is not the case for you, you may safely ignore such information.

Chapter 2 - ZBasic Language Elements

The logic of a program is described in the ZBasic language using identifiers, keywords, literal values, expression operators, statements, etc. These elements are all more fully described in this chapter.

2.1 Identifiers

In ZBasic, as in most computer languages, all constants, variables, subroutines and functions have a name. Such names are generally referred to as identifiers and each computer language has rules that specify how an identifier may be formed. In ZBasic an identifier must begin with an alphabetic character (A-Z, a-z) and may contain zero or more additional characters which may be alphabetic, numeric (0-9) or an underscore. There is virtually no maximum number of characters that an identifier may contain (in reality it is limited by the amount of memory available on your computer) but as a practical matter identifiers seldom exceed 15 to 20 characters. Beyond that they become somewhat cumbersome to type.

As mentioned previously, the alphabetic case of the letters of an identifier is insignificant. The variable `myVar` is the considered the same as the variable `MyVar`.

There is a set of identifiers that are reserved for special purposes and cannot be otherwise used in your program. Some of the reserved words are data type names, some are used as keywords in ZBasic statements (`If`, `For`, `Sub`, etc.). Some reserved identifiers have no current use in ZBasic but they are reserved nonetheless because they are keywords in other Basic dialects and may be incorporated into ZBasic in the future. If you attempt to use a reserved word in a role other than its predetermined role the compiler will issue an error message pointing out the unacceptable use. See Appendix A for a complete list of reserved words.

2.2 Data Types

The table below describes the fundamental data types available in ZBasic. There are a few additional data types that are used for special purposes that will be described later.

Fundamental Data Types

Data Type Name	Range of Values
`Boolean`	`True, False`
`Byte`	0 to 255
`Integer`	-32,768 to 32,767
`UnsignedInteger`	0 to 65,535
`Long`	-2,147,483,648 to 2,147,483,647
`UnsignedLong`	0 to 4,294,967,295
`Single`	approximately ±1.4e-45 to ±3.4e+38 and 0.0
`String`	0 to 255 characters

It is important to note that some mathematical operations on floating point values (type `Single`) result in certain special values that indicate an exceptional result. Among these special values are ones that represent positive infinity, negative infinity and a general category referred to as “not a number”, called NaN. The System Library function `SngClass()` returns a value indicating the class to which a floating point value belongs. See the ZBasic System Library Reference Manual for more information on the `SngClass()` function.

In addition to the fundamental data types enumerated in the table above, ZBasic supports several additional special-purpose types. There are two additional integral-value types that are collectively referred to as sub-byte types – `Bit` and `Nibble`. These are useful for reducing the amount of space used for small-valued data but they are not quite as efficient with respect to code size as using `Byte` types. See Section 3.22 for more information on these special types. Also, two special string types are available – Bounded Strings and Fixed Length strings. These are supported largely for compatibility with BasicX but may be useful in special situations. See Section 2.11.1 and Section 2.11.2 for more information on Bounded Strings and Fixed Length strings, respectively.

ZBasic is a strongly typed language that doesn't allow you to freely intermix different data types. This helps prevent programming errors or unexpected program behavior that results from unanticipated type conversions. However, a series of well-defined type conversion functions is provided that allow you to make explicit type conversions when necessary. See Section 2.13 for more information. Lastly, Section 3.26 contains technical details of the implementation of the various data types. This information is generally not necessary for most programming purposes but it is sometimes useful to know.

2.3 Modules

A ZBasic program may comprise one or more modules each contained within a separate file. When a module is compiled, a module name is derived from the name of the file. The module name is mostly for internal use by the compiler but you'll find references to the module name in the generated map file. Also, in some special situations you may need to use the module name as part of an identifier in order to refer to a variable or constant in your code. This is usually not necessary but it is good to remember that it is available. See Section 3.1 for more information on resolving identifier references using the module name.

The module name is derived by first removing the path prefix, if any, and the extension (typically .bas). The remaining characters of the filename are examined and any character that is not a letter, a digit or an underscore is replaced with an underscore. For the most part, this process converts the filename into a legal ZBasic identifier.

As an example, consider the filename "`C:\temp\my test code-1.bas`". When you compile this module the derived module name is `my_test_code_1`. If you compile a file whose corresponding module name does not begin with a letter, the compilation will proceed as normal. However, you will not be able to refer to that module by its module name since it will not be a legal ZBasic identifier.

A module is divided conceptually into two sections: an options section which, if present, must be first, and a definitions section. The definitions section may also be omitted but with no definitions, the module serves no useful purpose.

2.3.1 The Options Section

The options section contains directives that tell the compiler how to process the definitions section that follows. In the absence of any option directives, the compiler uses certain default settings as described below. This means that it is perfectly reasonable, and quite common, not to have any option directives in a module. The options section may also contain comments. This is useful for documenting what you're trying to accomplish with each option directive.

Note that some of the settings affected by the option directives described in this section may also be affected by compiler command line options. Option directives have precedence over command line options. See Section 7.1 for more information on command line options.

`Option <pin> <pin-state>`

This option directive provides a way to configure an I/O pin to be an input or an output. If it is being configured to be an input, you can further specify that whether or not the pull-up resistor is enabled. If it is being configured to be an output, you can further specify the logic level that you want to be output. This option directive may only appear in the first module compiled - usually the one containing the `Main()` subroutine.

The *`<pin>`* element specifies the pin to be configured. The first way to specify the pin to configure is to use the word `Pin` followed immediately by a physical pin number, e.g. `Pin20`. Of course, the set of pin numbers that can be specified in this way depends on the ZX processor that you are using. For example, for the 24-pin ZX devices the allowable values are `Pin5` through `Pin20` as well as `Pin25`, `Pin26` and `Pin27`. The second way to specify the pin to be configured is to give the port letter and a bit number of the port with a period separating them, e.g. `C.0`. The advantage of the second method is that it is largely independent of the ZX processor that you're using.

The <*pin-state*> element may be one of the entries in the table below:

Pin Configuration Values

Pin State	Meaning
`zxInputTriState`	The pin should be a tri-state input (pull-up disabled).
`zxInputPullUp`	The pin should be an input with the pull-up enabled.
`zxOutputLow`	The pin should be an output set to logic zero.
`zxOutputHigh`	The pin should be an output set to logic one.

Alternatively, for compatibility with the `Option Port` directive, the <*pin-state*> element may be a quoted string containing a single character `T`, `P`, `0` or `1` corresponding to the entries in the table above. For convenience when using the 24-pin ZX devices, `RedLED` and `GreenLED` are allowed as synonyms for `Pin25` and `Pin26` respectively. When using these synonyms the <*pin-state*> must be specified as `On` or `Off`.

By default, all pins are configured to be inputs with the pullup disabled.

Examples

```
Option Pin5 zxOutputLow        ' make pin 5 an output at logic zero
Option Pin20 "T"               ' make pin 20 a tri-state input
Option C.0 zxOutputHigh        ' make bit 0 of Port C an output at logic one
Option RedLED On               ' turn on the red LED
```

Option *<port> <config-string>*

This option directive provides a way to configure an entire I/O port at once instead of doing so pin by pin. For each bit of the port being configured to be an input, you can further specify that whether or not the pull-up resistor is enabled. For each bit of the port being configured to be an output, you can further specify the logic level that you want to be output. This option directive may only appear in the first module compiled - usually the one containing the `Main()` subroutine.

The allowable values for the <*port*> element are `PortA`, `PortB`, `PortC`, etc. Note, however, that `PortB` and `PortD` will rarely be used with the 24 pin ZX devices since the many of the pins of those ports are not directly available. The <*config-string*> element must be a series of characters, enclosed in quote marks, which specify for each bit one of four configuration states. The allowable configuration characters are described in the table below. The <*config-string*> must contain exactly 8 configuration characters, the leftmost of which corresponds to the most significant bit of the port and the rightmost of which corresponds to the least significant bit of the port.

Certain bits of PortB and PortD have dedicated uses on some ZX devices that require a specific configuration. Due to this requirement, those bits are protected from being changed by user-specified configuration. By default, all bits are configured to be inputs with the pullup disabled except for the special purpose bits.

Port Configuration Designators

Configuration Character	Meaning
`T`	The corresponding bit should be a tri-state input (pull-up disabled).
`P`	The corresponding bit should be an input with the pull-up enabled.
`0`	The corresponding bit should be an output set to logic zero.
`1`	The corresponding bit should be an output set to logic one.

Note that it is not necessary to configure your ports using this option directive. The configuration may also be done using the System Library subroutine `PutPin()` or built-in registers like `Register.DDRA`.

Also, note that if you specify configuration directives for both the pins and the port containing them, the last occurring directive will prevail.

Example

```
Option PortA "TPTP0001"
```

`Option Base` *`<value>`* — Default: `0`

Although zero-based arrays are common in many programming languages, some people find it easier to think about arrays being indexed beginning with 1. The `Option Base` directive is provided to allow you to specify that the default array base is either 0 or 1, the *`<value>`* element of the directive must be one of those values.

Note that it is not necessary to use this directive to define 1-based arrays. The syntax for array definitions (see page 14) allows you to specify any base value that you wish for each array, including a negative base value if you wish. Some programmers believe that it is better to explicitly indicate the base of the array indices in each definition. That way, you never have to remember or go find out what the default base might be.

This option does not affect Program Memory data items which are always 1-based.

Example

```
Option Base 1
```

`Option Explicit` — Default: `Off`

Traditionally, the purpose of the `Option Explicit` directive has been to require that each variable in the program be explicitly defined. Historically, early Basic dialects allowed programs to be written so that when the compiler encountered a new variable name, it automatically created a definition for it. Although this may have been thought to be a nice feature, it turned out to be the source of many programming errors and bugs. If you accidentally misspelled a variable name, a new variable was created entirely unbeknownst to you. Because of this serious drawback implicit variable definition is not supported by ZBasic. The `Option Explicit` directive is accepted by the compiler for compatibility reasons but neither its presence nor its absence affects any aspect of the compiler's operation.

`Option Language` *`{`* `BasicX | ZBasic` *`}`* — Default: `ZBasic`

By default the compiler processes modules in "native" mode, i.e. using the rules and defaults for ZBasic. This option directive can be used to instruct the compiler to process the module using the syntax rules defined by a specific language variant. If `Option Language BasicX` is specified, the compiler will process the module in BasicX compatibility mode. This may be useful in certain peculiar situations if you have existing modules written for BasicX. In most cases, existing BasicX code will compile correctly in native mode with few, if any, changes.

If you need to utilize BasicX compatibility mode you should be aware that none of the enhanced language features nor any of the enhanced System Library routines of ZBasic will be available in that module. Also, using `Option Language BasicX` changes the defaults for some of the other options as listed in the table below. You may still change the prevailing setting of these other options by utilizing the related option directive either before or after this option directive.

Option Directive	Default for BasicX	Default for ZBasic
`Strict`	`On`	`Off`
`AllocStr`	`Off`	`On`

`Option AllocStr` *`[`***`On | Off | Default`***`]`* — Default: `On`

By default, native compilation mode uses an allocation strategy for strings called dynamic string allocation. In contrast, BasicX uses an allocation strategy called static allocation. This option directive allows you to select the dynamic allocation strategy (`Option AllocStr On`) or the BasicX-compatible static allocation strategy (`Option AllocStr Off`). Specifying `Option AllocStr Default` sets the option to the default for the currently selected language. If neither `On, Off` nor `Default` is specified, the value `On` is assumed. See Section 3.26.2 for more information on the two allocation strategies.

Examples

```
Option AllocStr
Option AllocStr Off
```

`Option StringSize` *`{<value>`* **`| Default`***`}`* — Default: `20`

When string variables are defined, space to hold the characters of the string is allocated either statically or dynamically. When the space is statically allocated a fixed number of bytes of space is set aside for the string at compile-time thus setting the maximum size of that string. This option specifies the number of characters that should be reserved for statically allocated string storage. See the description of the `Option AllocStr` directive for more information about dynamically allocated string storage. If `Option StringSize Default` is specified the default value for the string size is used. This is useful if the string size was set using a command line option and you want to restore it to the default for this particular module.

Examples

```
Option StringSize 25
Option StringSize Default
```

`Option Strict` *`[`***`On | Off | Default`***`]`* — Default: `Off`

This option directive, supported for BasicX compatibility, enables or disables so-called “strict syntax checking”. You can enable strict mode by using `Option Strict` by itself or by including the keyword `On`. You disable strict syntax checking by using `Option Strict Off`. Specifying `Option Strict Default` sets the option to the default for the currently selected language.

The implications of strict syntax checking are noted in the description of each affected element but a summary of the effects is given here.

- the loop index variable of a For loop has restrictions on lifetime, visibility and accessibility
- logical operators like Not, And and Or may not be used with signed integral data types

Example

```
Option Strict Off
```

`Option TargetDevice` *`<device-name>`* — Default: `ZX24`

Due to differences between the various members of the ZX- series, the compiler needs to know for which device it should compile the code. This allows it to generate the correct code for the intended device. At present, the supported values for *`<device-name>`* are `ZX24`, `ZX24a`, `ZX-24n`, `ZX40`, `ZX40a`, `ZX40n`, ZX44, ZX44a, ZX40n, ZX1281, ZX1281n, ZX1280, ZX1280n, ZX24e, ZX24ae, ZX128e and ZX1281e. The device names are not case sensitive.

Example

```
Option TargetDevice ZX24a
```

Option TargetCPU *<CPU-type>* Default: ZX24

This option is deprecated, use Option TargetDevice instead.

Option PortPinEncoding *[***On | Off***]* Default: see text

When the compiler encounters an I/O port pin designator like C.2 it can convert it to the corresponding pin number for the target device or it can convert it to a composite value whose fields specify the port and the pin. This directive controls which of these conversions is performed. When the option is off, the result is a pin number and when it is on the result is the encoded port/pin value. The encoded result is a Byte value that has the binary form 1PPPPnnn where PPPP represents a port index (PortA=0, PortB=1, etc.) and nnn represents the bit number (0-7). The option is on by default for ZX devices based on the mega128, mega1280 and mega1281 and off by default for other devices.

Example

```
Option PortPinEncoding On
```

Option CodeLimit *<limit-value>* Default: none

This option can be used to have the compiler check the size of the generated code and issue an error message if it exceeds a specified size. The limit value is specified by a decimal number, optionally using the suffix K or k to denote a multiple of 1024.

Examples

```
Option CodeLimit 32K
Option CodeLimit 32768
```

Option SignOn *{***On | Off***}* Default: On

This option can be used to control the generation of a sign-on message when the ZX begins running after a reset. The flag to control the sign-on is stored in Persistent Memory of the processor. This is important to remember because if you download one program to the ZX that, say, turns it off and then you download another program that has no Option SignOn directive at all, the sign-on will still be in the off state. In other words, if you want to ensure that it is in particular state for your particular program, be sure to include this directive. Of course, if you've never turned it off, it will still be in the On state.

Example

```
Option SignOn Off
```

Option TaskStackMargin *<margin-value>* Default: 10

When the compiler compares the size of the stack allocated to a task to the estimated task stack usage it pads the estimate with a safety margin. This option can be used to specify a value for that safety margin different from the default of 10 bytes. The safety margin may be any non-negative decimal integral value.

Example

```
Option TaskStackMargin 20
```

Option ExtRamConfig *[***On | Off |** *<constant-expression>]* Default: (see text)

For ZX devices that support external RAM, you may control whether external RAM is enabled using this directive. Using the keyword `On` enables external RAM support in the default configuration (16-bit address, no wait states). If you need a different configuration, you may specify a constant expression whose 16-bit value is written to the external RAM configuration registers of the CPU. The high byte is written to the register `XMCRB` register and the low byte is written to `XMCRA` register. Note that for the mega128 CPU, the high bit of the XMCRA register is undefined but the configuration value must have that bit asserted in order for the external RAM interface to be enabled.

The default external RAM configuration value is `&H0000` for the ZX-1281 and ZX-1280 while it is `&H0080` for the ZX-128e and ZX-1281e.

Examples

```
Option ExtRamConfig On
Option ExtRamConfig &H0084   ' enable with 1 wait state
```

Option RamSize *<constant-expression>* Default: (see text)

The compiler compares the aggregate size of the statically allocated data items to the amount of RAM available in the target device. If the aggregate size is too large, the compiler issues a warning to that effect. For ZX models that support external RAM (e.g. ZX-1281), the compiler must know the resultant size of User RAM in order to avoid generating meaningless warnings. That is the purpose of this directive, which may only be used on ZX models that support external RAM. The example below shows the simplest way of specifying the augmented User RAM size. The example is for the case where the maximum amount of additional RAM was added.

Example

```
Option RamSize 65536 – Register.RamStart
```

Option HeapSize *<constant-expression>* Default: (see text)

The function of this directive differs depending on whether the target device operates in VM mode (e.g. ZX-24) or native mode (e.g. ZX-24n). For VM mode devices, when the compiler performs various checks regarding the use of RAM, it takes into account an amount of RAM reserved for the heap (from which memory for strings, for example, is allocated). This directive provides a means to specify a different amount to reserve for the heap. Note that the reservation is conceptual only – it does not affect how the program operates in any way. It only affects whether or not the compiler will issue a warning about the use of RAM in a particular circumstance.

For native mode devices, this directive affects the allocation of RAM between the string heap and task stacks. The effect is to set a hard limit beyond which the heap will not grow, thereby preventing the heap from encroaching on a task stack.

If a size less than the minimum size (64 bytes) is specified, the minimum is used instead. Also, for devices with external RAM, the special value of 65535 (&Hffff) can be used to specify that all external

RAM should be used for the heap. The default heap size is 256 bytes for VM mode devices and 512 for native mode devices. See the section on setting heap and task stack sizes for more information.

Example

```
Option HeapSize 500
```

`Option HeapLimit` *`<constant-expression>`*	Default: n/a

This directive, useful only for native mode devices, provides a different way of allocating RAM between the string heap and the task stacks. The value provided is interpreted as a RAM address and specifies the limit beyond which the heap will not grow. This directive is perhaps more useful with ZX devices that support external RAM. See the section on setting heap and task stack sizes for more information.

`Option MainTaskStackSize` *`<constant-expression>`*	Default: n/a

This directive, useful only for native mode devices, provides a different way of allocating RAM between the string heap and the task stacks. The value provided is interpreted as the desired size of the task stack for the Main() task and the heap limit is set at the end of the task stack. See the section on setting heap and task stack sizes for more information.

`Option TxQueueSize` *`<constant-expression>`*	Default: 25
`Option RxQueueSize` *`<constant-expression>`*	Default: 50

These directives can be used to modify the sizes of the default transmission and reception queues for Com1. They are effective only for native mode devices such as the ZX-24n.

Example

```
Option TxQueueSize 100
```

`Option Com1Speed` *`<constant-expression>`*	Default: 19200

This directive can be used to modify the default speed of the Com1 serial channel. It is effective only for native mode devices such as the ZX-24n.

Example

```
Option Com1Speed 9600
```

2.3.2 The Definitions Section

The definitions section of a ZBasic program may contain constant definitions, variable definitions, subroutine definitions and function definitions. There may be any number of each of these types of definitions and the definitions may occur in any order. It is a common practice, however, to place constant and variable definitions at the top of the definitions section followed by subroutine and function definitions. On the other hand, some programmers prefer to define the constants and variables closer to the routine or routines that use them.

BasicX Compatibility Note

In BasicX mode, variables and constants may not be defined following any subroutine or function.

Each of these program items may be defined to be `Public` or `Private`. A public item is visible to other modules and may be referenced in the definitions contained in other modules. A private item is visible only within the module in which it is defined. Generally speaking, unless there is a specific need for an item to be public, it should be private. If you make something private and later decide that you need to reference it in another module, it is a simple matter to change the definition from private to public.

Defining Constants

It is often convenient to define constants that can be used in other parts of the program. Doing so generally helps clarify the purpose of the value, assuming a reasonably descriptive name is chosen, and also facilitates easier maintenance and modification of the program.

The syntax for defining a constant is as follows:

```
[Public | Private] Const <name> As <type> = <value>
```

If neither `Public` nor `Private` is specified, the constant will be private. The *<name>* must be a legal identifier as described in Section 2.1. The *<type>* must be one of the fundamental data type names described in Section 2.2 or an Enum type (described in Section 3.2). Lastly, the *<value>* element must be value or an expression that has a constant value and is the same type as (or compatible with) the specified *<type>*.

In many cases, the *<value>* will be a simple numeric literal like `-55` or `3.14159`. In the case of string constants, it may be a literal string like `"Hello, world!"`. However, it is sometimes convenient to define a constant in terms of another constant. Consider the example below.

```
Private Const Pi as Single = 3.14159
Const TwoPi as Single = Pi * 2.0
```

You may also use certain System Library functions in the constant's value expression. The restriction is that the expression must be able to be evaluated at compile time.

```
Private Const Pi as Single = ACos(-1.0)
Private Const InitialValue as Single = Sin(Pi / 2.0)
```

The definition of the value of *pi* in the manner shown in the first example is useful because it results in the maximum accuracy of the constant value.

String constants are sometimes useful as well.

```
Public Const VersionNum as String = "V1.0"
Public Const VersionDate as String = "Oct 2005"
Public Const VersionStr as String = VersionNum & " " & VersionDate
Public Const VersionDateUC as String = UCase(VersionDate)
```

You may define multiple constants of the same or different types on the same line.

```
Const c1 as Integer = 7, c2 as Single = 3e10
```

BasicX Compatibility Note

In BasicX mode, `UnsignedInteger`, `UnsignedLong` and `String` constants are not supported. Also, constant expressions cannot utilize built-in functions.

Defining Variables

To define a variable at the module level (as opposed to within a subroutine or function, described later) the syntax is:

```
{Public | Private | Dim} <name> As <type>
```

`Dim` has exactly the same effect as `Private`, i.e., the variable will only be directly accessible to code within the module.

Examples

```
Dim ival as Integer, pulseCount as Byte
Private busy as Boolean
Dim msg as String
```

The first example above shows two different variables being defined on the same line.

The initial value of a variable depends on its type and how it is defined. See Section 2.12 for information on variable initialization.

Defining Arrays of Variables

Variables which hold a single value like those discussed above are called scalar variables. ZBasic also supports arrays of variables. An array of a fundamental type may be defined using the syntax:

```
{Public | Private | Dim} <name>(<dim-spec-list>) As <type>
```

The *<dim-spec-list>* is a list of up to 8 dimension specifications each separated from the next by a comma. A dimension specification consists of a constant expression giving the upper bound of elements along that dimension or two constant expressions separated by the keyword `To` specifying the lower bound and upper bound, respectively, of the elements along that dimension.

When only the upper bound is given, the lower bound defaults to either zero or one depending on whether or not an `Option Base` directive is in effect or not. If no `Option Base` directive has been specified, the lower bound is zero.

Note that in order to be passed as a parameter to a subroutine or function an array must have a lower bound of 1. For this reason it is probably more common to define arrays with a lower bound of 1. Many people find it easier to think about arrays this way as well. The default lower bound of zero is kept for compatibility reasons.

Examples

```
Option Base 0
Dim ival(5) as Integer
```

This defines an array of `Integer` values with 6 elements. The lower bound is zero and the upper bound is 5.

```
Option Base 1
...
Dim ival(5) as Integer
```

This defines an array of `Integer` values with 5 elements. The lower bound is 1 and the upper bound is 5. The lower bound may be explicitly specified as well. The second example below illustrates the use of a negative value as one of the bounds. Note that the upper bound must be greater than or equal to the lower bound.

```
Public pulseCount(1 to 6) as Byte
Private itemData(-3 to 45, 1 to 4) as Boolean
```

You may also define an array of strings but only when `Option AllocStr` is in effect as it is by default.

```
Dim msg(1 To 4) as String
```

BasicX Compatibility Note

In BasicX mode, each array dimension must have at least two elements. Also, arrays of type `String` are not supported.

There is no checking, either at compile time or at run time, for array index underflow or overflow. If you write code that uses an index outside of the defined range of indices, the results are undefined.

Defining Subroutines

A subroutine is a collection of statements that can be executed by using the subroutine name in a `Call` statement. The advantage of creating subroutines is that we can think of them as logical blocks instead of thinking about all of the details that are dealt with by the statements within the subroutine.

A subroutine may be defined as taking zero or more parameters. If it has parameters defined, you must supply a value for each of the parameters when you invoke the subroutine using the `Call` statement. The syntax for defining a subroutine is shown below.

```
[Public | Private] Sub <name> ( [<parameter-list>] )
      [<statements>]
End Sub
```

If neither `Public` nor `Private` is specified, `Public` is assumed.

The *`<parameter-list>`* consists of zero or more parameter specifications, each separated from the next by a comma. Note that the parentheses are required even when there are no parameters. The parameters given in the subroutine’s definition are called the “formal parameters”. The parameters that appear in each invocation of the subroutine are referred to as the “actual parameters”.

The syntax for a formal parameter specification is:

```
[ByVal | ByRef] <name> As <type>
```

The *`<name>`* element is the name by which the passed parameter is known within the subroutine. The *`<type>`* element is one of the fundamental types listed in the table in Section 2.2. You may also specify a default value for a parameter, a topic that is discussed in more detail in Section 3.20.

The keywords `ByVal` and `ByRef` refer to the method by which the parameter is passed to the subroutine. The keyword `ByVal` means that the parameter is passed to the subroutine “by value” while `ByRef` means that the parameter is passed “by reference”. Each of these parameter passing conventions has its own advantages and disadvantages and the full explanation of the difference between these two parameter passing methods is given in Section 2.14. Suffice to say here that the primary difference is whether or not the called subroutine is able to change the value of the passed parameter from the perspective of the caller. Some variable types may be passed by one of the methods but not by the other. Again, see Section 2.14 for complete details. The default passing convention, if neither `ByVal` nor `ByRef` is specified, is `ByRef`.

To specify a formal parameter that is an array, simply add a set of parentheses following the parameter name. For example,

```
Sub mySub(ByRef data() as Byte)
End Sub
```

Only one-dimensioned arrays whose lower bound is 1 may be passed as parameters and they must be passed by reference. The upper bound is indeterminate – it is the responsibility of the programmer to ensure that the subroutine does not access elements beyond the upper bound of the passed array. Often, programmers will include a parameter that specifies the upper bound so that the code may safely operate with different sizes of arrays.

```
Sub mySub(ByRef data() as Byte, ByVal count as Byte)
End Sub
```

Lastly, the `<statements>` element in the subroutine definition represents zero or more valid ZBasic statements that are described in Section 2.5 of this manual.

Example

```
Private Const redLED as Byte = 25   ' define the pin for the red LED
Private Const grnLED as Byte = 26   ' define the pin for the green LED

Public Sub Main()
      ' configure the pins to be outputs, LEDs off
      Call PutPin(redLed, zxOutputHigh)
      Call PutPin(grnLed, zxOutputHigh)

      ' alternately blink the LEDs
      Do
            ' turn on the red LED
            Call Blink(redLed)

            ' turn on the green LED
            Call Blink(grnLed)
      Loop
End Sub

Private Sub Blink(ByVal pin as Byte)
      ' turn on the LED connected to the specified pin for one half second
      Call PutPin(pin, zxOutputLow)
      Call Delay(0.5)
      Call PutPin(pin, zxOutputHigh)
End Sub
```

In this program, we have factored out the code that turns an LED on and off into a subroutine named `Blink()`. In the definition of `Blink()`, `pin` is called the formal parameter. In `Main()` where `Blink()` is invoked, the parameters `redLed` and `grnLed` are the actual parameters.

By factoring out the code that was common to blinking the two LEDs we have simplified the program. The details of how an LED is blinked are encapsulated in the definition of `Blink()`. No longer does the `Main()` subroutine need to know how to blink an LED; it just calls the subroutine to handle all of the details of blinking an LED connected to a specific pin and provides the necessary data for `Blink()` to do the work. In this case, all that is needed is a pin number.

If we wished to do so, we could add another parameter to the blink subroutine to specify how long we want the LED illuminated.

```
Private Sub Blink(ByVal pin as Byte, ByVal duration as Single)
      ' turn on the LED connected to the specified pin for the time given
      Call PutPin(pin, zxOutputLow)
      Call Delay(duration)
      Call PutPin(pin, zxOutputHigh)
End Sub
```

With this definition, we would need to add a second actual parameter in each call. For example,

```
Call Blink(redLed, 0.5)
```

It is recommended that you invoke subroutines as shown using the `Call` keyword as shown. However, for compatibility with other Basic dialects it is also possible, although not recommended, to invoke a subroutine by using its name as if it were a statement. If this is done, the actual parameters are not allowed to have enclosing parentheses as illustrated below.

```
Blink redLed, 0.5
```

One powerful aspect of subroutines is that variables and constants may be defined within a subroutine itself. When this is done, the variable or constant is private to the subroutine and cannot be directly accessed from any other routine. Although it is common to place such definitions near the beginning of the subroutine, the definitions may occur anywhere in the subroutine as long as they occur before their first use.

Note that the normal execution sequence of a subroutine may be altered by using the `Exit Sub` statement. When this statement is executed, it causes control to return to the caller immediately bypassing the remaining statements in the subroutine.

Defining Functions

A function is a collection of statements that can be executed by using the function name in place of a value or in an expression. The advantage of creating functions is that we can think of them as logical blocks instead of thinking about all of the details that are dealt with by the statements within the functions.

Like a subroutine, a function may have zero or more parameters defined. If it has parameters defined, you must supply values for each of the parameters when you invoke the function.

The syntax for defining a function is very similar to that for defining a subroutine and is shown below.

```
[Public | Private] Function <name> ( [<parameter-list>] ) As <type>
      [<statements>]
End Function
```

The primary difference is the use of the keyword `Function` in place of `Sub` and the specification of a type for the value to be returned by the function. Like a subroutine, if neither `Public` nor `Private` is specified, `Public` is assumed. The *`<parameter-list>`* syntax is identical to that for a subroutine.

Example

```
' compute the factorial of the value provided
Public Function Factorial(ByVal val as Long) As Long
      Dim factVal as Long
      factVal = 1
      Do While (val > 1)
            factVal = factVal * val
            val = val - 1
      Loop
      Factorial = factVal
End Function
```

This function implements a mathematical operation called factorial. The value of N factorial (usually written N!) is the product of all the integers from 1 up to and including N. The value of 0! is, by definition, 1. The value of 3! is 6 and so on.

This example introduces a couple of statements that we haven't seen yet but they are fairly straightforward. Notice that a variable named `factVal` was defined within the function. This variable is called a "local variable" and it is not visible to any code outside of the function. The other aspect of this local variable that is different from variables defined at the module level is that this variable only takes up space while the function is executing. When the function returns, the space used by the local variable is reclaimed by the system and can be used for other purposes. See Section 3.1 for more information on the concepts of scope and lifetime of variables.

The second line of the function shows a value being assigned to the variable `factVal`. Instead of using the literal constant value as shown on the right hand side of the equal sign, we could just as well have written an expression that involved several values (perhaps including function invocations) and operators like addition, subtraction, multiplication, etc. This statement is known as an assignment statement and is described in more detail in Section 2.5.1.

The third line of code illustrates a useful variation on the `Do` loop that we have already seen. In this case, the `Do` has a condition associated with it. The condition is tested before every pass through the loop and as long as the conditional expression evaluates to the Boolean value `True`, the statements within the loop are executed again. In this case, the condition tests if the `val` parameter has a value greater than `1`, the statements in the loop will be executed. Otherwise, control will transfer to the first statement following the `Loop` statement.

Within the loop, there are two more assignment statements, each of which has an expression on the right hand side, the first involving multiplication and the second involving subtraction. The meaning of the expressions should be self-evident.

The last statement of the function, just before the `End Function`, illustrates how the value to be returned by the function is set. It is useful to think of there being a variable automatically defined that has the same name as the function as well as the same type. We call this a return value variable. Within the function, you can refer to the return value variable, in most cases, just as you would any other variable or parameter. In the example above, instead of introducing a new variable `factVal`, we could have instead simply used the reference to the return value variable `Factorial`. The function's code is re-written below using this idea.

```
Function Factorial(ByVal val as Long) As Long
      Factorial = 1
      Do While (val > 1)
            Factorial = Factorial * val
            val = val - 1
      Loop
End Function
```

As with subroutines, variables and constants may be defined within a function. When this is done, the variable or constant is private to the function and cannot be directly accessed from any other routine. Although it is common to place such definitions near the beginning of the function, the definitions may occur anywhere in the subroutine as long as they occur before their first use.

Note that the normal execution sequence of a function may be altered by using the `Exit Function` statement. When executed, this statement causes control to return to the caller immediately. Also, it is important to know that the return value variable of a function is not automatically initialized. If your function returns without having assigned a value to the return value variable, the return value will have an undefined value.

Once a function is defined, it may be used anywhere a variable may be used, e.g. in an expression. One exception is that a function name may not be used on the left hand side of an assignment.

```
Dim lval as Long
lval = Factorial(4)
```

If a function is defined as taking zero parameters, it may be invoked by giving its name without the parentheses following it. This form is supported for compatibility reasons but its use is discouraged. If the parentheses are present a reader of the code knows immediately that it is a function invocation as opposed to the use of a variable or constant.

BasicX Compatibility Note

In BasicX compatibility mode, when a function is defined as returning an UnsignedInteger or UnsignedLong type, the very first line of the function must be a Set statement

2.4 Expressions

Expressions are an important part of most ZBasic programs. They provide the means by which your programs implement the mathematical, logical and comparison operations necessary for your application. Generally, anywhere a value may be used, an expression may be used as well. An expression consists of one or more values, called operands, and one or more operators that indicate the function to perform on the operands.

Example

```
b = b * 5 + 3
```

In this assignment statement, the value being assigned to the variable b is an expression comprising three operands and two operators. Some operators, like both of those in the expression above, are called “binary operators” because they require two operands. Other operators require only one operand and are called “unary operators”. The available operators and their characteristics are described in subsequent sections.

ZBasic is a strongly typed language. This means that operands supplied for binary operators must generally be of the same type. The only exception to this rule is the exponentiation operator which allows a restricted mixing of types as described in Section 2.4.3.

The order of evaluation of the components of an expression is governed by operator precedence and associativity as described in the next two sections. However, that order may be overridden by the use of parentheses.

2.4.1 Operator Precedence

Consider again the example expression `b = b * 5 + 3`. Each of the operators requires two operands but it may not be immediately clear what the operands are in each case. It could be that the expression above means to add 5 to 3 and then multiply the result by the value of b. On the other hand, it could mean to multiply the value of b by 5 and then add 3 to the result. The property that dictates the order in which operators are applied in this case is called “operator precedence”. An operator having higher precedence has priority over an operator with lower precedence and is therefore applied first. The table below depicts the precedence of ZBasic operators.

Operator Precedence

Precedence Level	Operators
11 (highest)	`^`
10	`+(unary) -(unary)`
9	`* /`
8	`\`
7	`Mod`

6	`+ -`
5	`&`
4	`= < > <= >= <>`
3	`Not`
2	`And`
1	`Or`
0 (lowest)	`Xor`

The actual precedence level values are of no particular significance. All that matters is whether the precedence value of one operator is higher, equal to or lower than that of another operator. In the example used above, since the multiplication operator has higher precedence than the addition operator, the meaning of the expression is to multiply the value of the variable b by 5 and then add 3 to the result. It happens in this case that the order of application of the operators is left to right but that is not always the case. Consider this slightly different example:

```
b = b + 5 * 3
```

The meaning of this expression is to multiply 5 by 3 and then add the value of the variable b to the result. The relative precedence level of the operators requires that they be applied in an order that is not left to right.

BasicX Compatibility Note

In BasicX mode, a different operator precedence set is used, as shown in the table below, in order to be compatible with BasicX.

Operator Precedence

Precedence Level	**Operators**
5 (highest)	`^`
4	`+(unary) -(unary)`
3	`Not`
2	`* / \ Mod And`
1	`+ - Or Xor &`
0 (lowest)	`= < > <= >= <>`

2.4.2 Operator Associativity

After studying the precedence table in the preceding section, the obvious question would be what happens when two operators have the same precedence?

```
b = b - 5 + 3
```

In case of equal precedence, the operators are applied in the order dictated by the associativity of the operator. Except for the exponentiation operator, all operators in ZBasic are left associative. This means that given equal precedence operators they are applied in left to right order. Exponentiation is right associative meaning that they will be applied in right to left order.

However, no matter what the precedence levels are you can force the operators to be applied in any order that you wish by utilizing parentheses. Consider the two examples below.

```
b = (b – 5) + 3
b = b – (5 + 3)
```

The parentheses indicate that the expression within should be evaluated first after which the result may be used as an operand in another operation.

2.4.3 Arithmetic Operators

The arithmetic operators are listed in the table below along with the permitted operand types. The result is the same type as the operands.

Arithmetic Operators

Function	Type	Operator	Permitted Operand Types
Negation	Unary	`-`	any numeric
Addition	Binary	`+`	any numeric
Subtraction	Binary	`-`	any numeric
Multiplication	Binary	`*`	any numeric[1]
Division, Integer	Binary	`\`	any integral[1]
Division, Real	Binary	`/`	`Single`
Modulus	Binary	`Mod`	any numeric[1]
Exponentiation	Binary	`^`	any numeric[2]

Notes:
[1] In BasicX mode, operating on `UnsignedLong` is not supported.
[2] The result will be the same type as the left operand. In BasicX mode, the left operand must be `Single` and the right operand must be either `Single` or `Integer`.

For `Single` operands, dividing by zero produces a special value indicating either positive or negative infinity depending on the sign of the dividend. Dividing 0.0 by 0.0 produces a special value called NaN, representing a value that is "Not a Number". Similarly, performing the Mod operation with `Single` values using a divisor of zero produces a NaN. For all other operand types the result of using a zero divisor for either division or Mod is undefined. The System Library function `SngClass()` returns a value indicating the general classification of a value of type `Single`. See the description of `SngClass()` in the System Library Reference Manual for more information.

2.4.4 Logical Operators

The logical operators are listed in the table below along with the permitted operand types. The result is the same type as the operand(s).

Logical Operators

Function	Type	Operator	Permitted Operand Types
Logical AND (conjunction)	Binary	`And`	`Boolean`
Bitwise AND	Binary	`And`	any integral[1]
Logical OR (disjunction)	Binary	`Or`	`Boolean`
Bitwise OR	Binary	`Or`	any integral[1]
Logical XOR (exclusive disjunction)	Binary	`Xor`	`Boolean`
Bitwise XOR	Binary	`Xor`	any integral[1]
Logical Complement	Unary	`Not`	`Boolean`
Bitwise Complement	Unary	`Not`	any integral[1]

Notes:
[1] If `Option Strict` is enabled, signed types are not allowed.

2.4.5 Comparison Operators

The comparison operators are listed in the table below along with the permitted operand types. The result type is `Boolean`.

Comparison Operators

Function	Type	Operator	Permitted Operand Types
Equality	Binary	=	any
Inequality	Binary	<>	any
Greater Than	Binary	>	`String` or any numeric
Greater Than or Equal To	Binary	>=	`String` or any numeric
Less Than	Binary	<	`String` or any numeric
Less Than or Equal To	Binary	<=	`String` or any numeric

2.4.6 Miscellaneous Operators

The remaining operator to be described is the string concatenation operator, `&`. Both operands must be type `String` and the result will be type `String`. Note that the + operator may also be used for concatenating strings. The sole difference between using & and using + is that the former supports automatic value-to-string conversion while the latter does not.

2.4.7 No "Short Circuit" Evaluation

It is important to note that in ZBasic, as in most Basic dialects, every term in an expression is always evaluated irrespective of the intermediate results. This is a technical detail that is significant only when an expression contains function invocations and the act of invoking one or more of the functions involved has "side effects" like modifying a global variable, modifying a parameter passed by reference, or changing the state of the hardware. Consider the evaluation of the conditional expression in the `If` statement below when the value of the variable a is, say, 10.

```
If (a > 3) Or (foo() > 10) Then
    [other statements]
End If
```

When the expression on the left side of the `Or` operator is evaluated the result will be True. Because of this fact we know that the resulting value of the entire conditional expression will also be True – nothing on the right hand side can possibly affect the outcome. Nonetheless, the expression on the right hand side of the `Or` operation will still be evaluated and thus the function `foo()` will be invoked. Some other computer languages, notably C/C++ and Java, implement the concept of "short circuit evaluation". In those languages, the evaluation of an expression stops as soon as the result is known. If that were the case here, the right hand side of the `Or` expression would not be evaluated and, hence, the function `foo()` would not be invoked. To reiterate, ZBasic does not implement short circuit evaluation.

2.5 Statements

Within a subroutine or function you can define variables and use statements to implement the logic required for the functionality of the routine. This section describes the types of statements available. ZBasic statements may be divided into two general categories: simple and compound. An example of a simple statement is the assignment statement where the entire statement is expressed on one line (ignoring possible line continuations). In contrast, a compound statement comprises two or more lines and may contain other statements within it. In many respects, it is convenient to think of a compound statement as if it were a single statement even though it may have many constituent statements.

2.5.1 Assignment Statement

The assignment statement is perhaps the most basic and most often used statement in a program. The syntax of an assignment statement is shown below in two forms, one for assigning a value to a scalar variable and one for assigning a value to an array element.

```
<var-name> = <value>
<var-name>( <index-list> ) = <value>
```

In both cases, the *<value>* element may be a simple value or a complex expression involving one or more other variables, constants, functions, etc. However, the type of *<value>* must match the type of the variable or array element to which it is being assigned. There is no automatic type conversion.

When assigning a value to an array element, you must specify the index or indices of the particular element of interest. For a single-dimensional array, you will specify a single value for the index of the desired element. For multi-dimensional arrays, you must specify a value for each of the indices separated from one another by a comma.

Example

```
Dim i as Integer
Dim ia(1 to 5) as Integer
Dim board(1 to 12, 1 to 12) as Byte

i = 2
ia(i) = (ia(3) + 2) / 4
board(i, 6) = CByte(ia(2))
```

2.5.2 Call Statement

The Call statement is used to invoke a subroutine. The syntax is:

`Call <subroutine-name> ( [<parameter-list>] )`

The optional *<parameter-list>* must contain the proper number of parameters each of the correct type for the subroutine being invoked. If more than one parameter is given each parameter must be separated from the next by a comma.

When this statement is executed the supplied parameters, if any, are pushed on the stack and control is transferred to the first statement of the subroutine. When the subroutine finishes executing control resumes with statement following the Call statement.

Although not recommended, for compatibility with other Basic dialects it is permissible to omit the Call keyword. If this is done the parentheses surrounding the parameter list must also be omitted. Note, particularly, the third example below that seems to violate this rule. However, it does not because a parenthesized expression is, in fact, an expression.

Examples

```
Call PutPin(12, 0)
PutPin 12, 0
Delay (1.0)
```

2.5.3 CallTask Statement

The CallTask statement is used to start a task. See Section 3.5 for more information on using tasks. The basic syntax to invoke a task is:

`CallTask <task-name>, <task-stack>`

In this case, the *<task-name>* element must be the name of a user-defined subroutine that takes no parameters. The *<task-stack>* must be the name of a Byte array that will serve as the stack for the task. For compatibility with BasicX, the *<task-name>* may be enclosed in quote marks.

Example

```
Dim ts1(1 to 40) as Byte

CallTask task1, ts1
```

A task may also be passed parameters when it is invoked. The syntax for doing so is similar to that for invoking a subroutine that requires parameters.

```
CallTask <task-name>( <parameter-list> ), <task-stack>
```

See the discussion of the CallTask statement in the ZBasic System Library Reference manual for more details on the allowed parameter types. This syntax is not supported in BasicX compatibility mode.

Example

```
Dim ts1(1 to 40) as Byte

CallTask task1(&H100), ts1
```

For advanced users, the task stack may also be specified by giving its address explicitly. This topic is discussed in section 3.5.1, Advanced Multi-tasking Options.

2.5.4 Console.Write and Console.WriteLine Statements

These statements (with a syntax more akin to object-oriented methods) are similar to `Debug.Print` but they are limited to displaying one string at a time. They are supported for compatibility with Visual Basic. The syntax of the statements is:

```
Console.Write( <string-expression> )
Console.WriteLine( <string-expression> )
```

The difference between these two statements is that the latter also outputs a carriage return/line feed following the string while the former does not.

Examples

```
Dim i as Integer
Dim s as String

Console.WriteLine(CStr(i))
Console.WriteLine("i = " & CStr(i))
Console.Write("s = ")
Console.WriteLine(s)
Console.WriteLine("")
```

This sequence of statements is written to produce exactly the same result as the sequence of `Debug.Print` statements above. Note how the string concatenation operation is used in the second `Console.WriteLine()` invocation to produce a single string. In the final example, an empty string is output followed by a carriage return/line feed.

Note that there are counterparts to these statements called Console.Read and Console.ReadLine that allow you to retrieve data from the Com1 serial port. However, these are technically functions (since they both return a value) and are therefore not described here. See the ZBasic System Library Reference manual for information on these functions.

2.5.5 Debug.Print Statement

One technique for debugging a program is called "print statement debugging". The idea is that to determine how your program is executing you insert statements into the program that display the values of important variables or simply display a distinctive message so that you know what the program is doing. `Debug.Print` is a special statement intended just for this purpose. The syntax is:

```
Debug.Print [<string-list>][;]
```

The *`<string-list>`* element represents zero or more string expressions separated from one another by a semicolon. Each of the string expressions is evaluated in turn, from left to right, and the string result of each is output to Com1. If the optional trailing semicolon is omitted, a carriage return/line feed will also be output so that the next time something is output to Com1 it will appear on a new line. If you don't want the subsequent output to be on a new line, simply add the semicolon at the end of the list. This is often done when you need to compute several different values to output. You can use a separate `Debug.Print` statement for each value and keep them all on the same output line by ending all but the last with a semicolon. It is permitted, syntactically, to specify an empty string list but still include the trailing semicolon. However, this construction does nothing.

Note that each of the items to be displayed must be a string. You can use the `CStr()` function to produce a string from any value.

Examples

```
Dim i as Integer
Dim s as String

Debug.Print CStr(i)
Debug.Print "i = ";CStr(i)
Debug.Print "s = ";
Debug.Print s
Debug.Print
```

The last example, with an empty *`<string-list>`* and no trailing semicolon, will simply send a carriage-return/linefeed to Com1.

The unusual form of Debug.Print is due to its heritage from Visual Basic. It should probably be called the Print method of the system Debug object but it even departs from the traditional syntax of the methods that are part of object-oriented languages. Nonetheless, it is included for compatibility as well as its utility.

2.5.6 Do-Loop Statement and Variants

This compound statement, briefly mentioned earlier in this document, is the basic repetition construct in ZBasic. The syntax is:

```
Do
    [<statements>]
Loop
```

This construct causes the sequence of zero or more statements to be repeatedly executed. However, execution of the loop may be terminated using an `Exit Do` statement at which point control will transfer to the first statement following the `Loop` statement. Note that the `Do-Loop` compound statement may be nested and the `Exit Do` only terminates the innermost `Do-Loop` that contains it.

Example

```
Do
    <other-statements>
    Do
        <other-statements>
        If (i > 5) Then
            <other-statements>
            Exit Do
        End If
    Loop
    <other-statements>
Loop
```

Here, the `Exit Do` only terminates the inner Do-Loop; the outer one continues to iterate.

There are four other variations on this basic looping concept, all of which involve a condition for continuing the iteration. The syntax of the four variations is as follows:

```
Do While <boolean-expression>
      [<statements>]
Loop

Do Until <boolean-expression>
      [<statements>]
Loop

Do
      [<statements>]
Loop While <boolean-expression>

Do
      [<statements>]
Loop Until <boolean-expression>
```

As you can see, the difference between the four variations is whether the test is at the top of the loop or at the bottom of the loop and, secondly, the logic sense of the condition. Testing the condition at the top of the loop means that the statements within the loop may be executed zero or more times. Testing the condition at the bottom of the loop means that the statements will always be executed at least once.

The difference between using While and Until is nothing more than a logic inversion. The construct `Do While Not <boolean-expression>` is logically equivalent to `Do Until <boolean-expression>`.

There is no fixed limit on how deeply Do loops may be nested. The actual limit is governed by how much memory is available to the compiler. For all practical purposes, there is no limit.

BasicX Compatibility Note

In BasicX mode, the nesting of Do loops is limited to 10 for compatibility.

2.5.7 Exit Statement

The Exit statement allows you to terminate the execution of a loop, a function or a subroutine earlier than it otherwise would. This is most commonly used when a condition is detected by the code that prevents further normal processing. The syntax for the Exit statement is:

```
Exit <exit-type>
```

The *<exit-type>* element may be `Do`, `For`, `Sub` or `Function` but the use of each is restricted to use within a Do-Loop, For-Next, subroutine and function, respectively. Any other use will result in a compiler error.

Example

```
Do
    <other-statements>
    If (i > 5) Then
        Exit Do
    End If
    <other-statements>
Loop
```

When an `Exit Do` is executed within nested Do-Loop statements only the innermost Do-Loop that contains the exit statement will be terminated. Control will be transferred to the first statement following the terminated Do-Loop. The same idea applies to an `Exit For` within nested For-Next statements.

2.5.8 For-Next Statement

The For-Next compound statement is another form of looping that provides controlled iteration using a loop index variable. The syntax for a For-Next loop is:

```
For <var> = <start-expr> To <end-expr> [ Step <step-expr> ]
    [<statements>]
Next [ <var> ]
```

The *<var>* element, referred to as the loop index variable, must refer to a previously defined scalar variable (i.e. not an array element) that is either a numeric type or an enumeration type. The *<start-expr>*, *<end-expr>* and optional *<step-expr>* elements must all be the same type as the loop index variable.

Example

```
Dim i as Integer
For i = 1 To 10
    Debug.Print CStr(i)
Next i
```

When the For statement begins execution, the *<start-expr>* is evaluated and the resulting value is assigned to the loop index variable. Then, before executing any statements contained within the body of the For-Next statement, the *<end-expr>* is evaluated and the value of loop index variable is compared to that value. If the value of the loop index variable is less than or equal to the value of the *<end-expr>* the statements within the body of the For-Next are executed. This is called the "loop entry test" because it controls whether or not the loop statements are executed.

At the bottom of the loop, marked by the `Next` statement, the loop index variable is modified in preparation for the next iteration of the For-Next loop. If the optional `Step` *<step-expr>* is present its value is added to the loop index variable, otherwise the loop index variable is simply augmented by 1. Then control is transferred back to the top of the loop where the loop entry test is performed again.

The logic of the For loop in the example above may be exactly duplicated using other statements as illustrated below. You can see that the For loop allows you to express the same logic more concisely.

```
Dim i as Integer

i = 1
Do While (i <= 10)
    Debug.Print CStr(i)
    i = i + 1
Loop
```

There are several things to note about the For-Next loop. Firstly, it is important to be aware that although the values of the *<end-expr>* and *<step-expr>* elements are used multiple times, the expressions are only evaluated once, when the execution of the For-Next begins. This distinction is only important, of course, if these expressions contain references to other variables that might be modified during loop execution or if they involve function calls. Secondly, because the loop entry test checks to see if the loop index variable is less than or equal to the *<end-expr>* you must choose a data type for the loop index variable that is capable of representing a value that is greater than the value of the *<end-expr>* . Otherwise, the loop entry test will always be true. Consider this errant example:

```
Dim i as Byte

For i = 0 to 255
      <statements>
Next i
```

This loop will never terminate because the value of `i` is always less than or equal to 255 since it is a `Byte` type. Where possible, the compiler will issue a warning if it detects these problematic conditions. Even so, you should develop the habit of considering this potential problem every time you code a For-Next loop.

The third important aspect of the For-Next statement is that if the *<step-expr>* evaluates to a negative value, the sense of the loop entry test changes. In this case, the loop index variable is tested to see if it is greater than or equal to the *<end-expr>* and, if so, the statements of the loop are executed. Also, in this case the caveat noted above about the range of the loop index variable changes. The loop index variable must be capable of representing a value that is less than the value of the *<end-expr>*. When a For loop is used with an unsigned data type, the step value is considered to be negative if the most significant bit of the value is a 1.

One other note: it is permissible for the *<step-expr>* to evaluate to zero. This will cause the For loop to execute indefinitely. The For-Next loop may be terminated at any time by using the `Exit For` statement.

The presence of the *<var>* on the `Next` statement is optional. However, if it is present, it must match the name of the loop index variable of the For loop with which it is associated. There is no fixed limit on how deeply For-Next loops may be nested. The actual limit is governed by how much memory is available to the compiler. For all practical purposes, there is no limit.

BasicX Compatibility Note

In BasicX mode, the For-Next statement is much more restrictive. The loop index variable must be a scalar integral type. The DownTo keyword is not supported. Referring to the loop index variable in a `Next` statement is not supported. The *<step-expr>* is restricted to a constant expression that evaluates at compile time to either 1 or –1. Lastly, For-Next loops may be nested to a maximum depth of 10 for compatibility.

When `Option Strict` is enabled, there are additional restrictions that apply. Firstly, the loop index variable must be local to the routine; it cannot be defined at the module level. Secondly, the loop index variable is not allowed to be used or modified outside of the For-Next loop except that it can be used as the loop index variable in a subsequent For-Next loop. Thirdly, inside the For-Next loop the loop index variable is read-only. Any attempt to modify the loop index variable, or pass it by reference to another routine will result in an error message from the compiler.

One final note: although loop index variables of type `Single` are allowed some experienced programmers advise against doing so. This is due to the fact that not all real numbers can be exactly represented as a `Single` value. Consequently, using a `Single` loop index variable may not produce the expected results. It is often better to use an integral loop index variable along with an auxiliary real variable to accomplish the desired objective.

2.5.9 Goto Statement

The Goto statement allows you to transfer control to a specific point in the sequence of statements that comprise a subroutine or function. The point to which control is transferred is marked by a label statement. The label statement is simply an identifier followed by a colon appearing on a line by itself (except that it may be followed by a comment).

Because it interferes with the normal program flow, the Goto statement can be overused resulting in a program that is difficult to understand and, therefore, difficult to maintain. Some programmers believe that a Goto statement should never be used. Others believe that it is acceptable to use a Goto but only if the alternative code structure is even less palatable. The latter strategy is probably the best to adopt.

Example

```
    Goto doOtherStuff
    <other-statements>
doOtherStuff:
    <other-statements>
```

2.5.10 If-Then-Else Statement

The if-then-else compound statement is the basic decision making construct in ZBasic. In its simplest form, the syntax is:

```
If <boolean-expression> Then
    <statements>
End If
```

The *<boolean-expression>* element is an expression whose value is of type `Boolean`. It most often involves one of the conditional operators that allow you to compare the values of two expressions but it may also simply be the invocation of a function whose return type is `Boolean`.

The *<statements>* element represents zero or more ZBasic statements possibly including other If statements. This allows you to create nested decision-making statements of arbitrary complexity.

Although the construction described above is useful, it is often the case that you want your program to execute a certain set of statements if a condition is true but you want it to execute a different set of statements if the condition is false. The If statement allows this logic using the syntax shown below.

```
If <boolean-expression> Then
    <statements>
Else
    <statements>
End If
```

Sometimes you'll want to test several different conditions and execute a different set of statements in each case. The If statement allows this logic using the following syntax:

```
If <boolean-expression> Then
    <statements>
ElseIf <boolean-expression> Then
    <statements>
Else
    <statements>
End If
```

The `ElseIf` portion of the statement, including the associated statements, may occur zero or more times. The `Else` portion of the statement, along with its associated statements, may occur zero or one times. Note that the logic of the If-Then-Else statement is designed so that at most one set of statements gets executed. This construct represents a series of tests that are performed sequentially. The first such test that produces a Boolean `True` result will cause the statements associated with that test to be executed. If none of the tests produce a `True` result and an `Else` clause exists, the statements associated with the `Else` will be executed. In all cases, after the set of statements is executed, control transfers to the first statement following the `End If`.

If the same expression is being repeatedly tested against different values, it is more efficient to use the Select-Case statement described in Section 2.5.11.

Examples

```
If (i > 3) Then
    Call PutPin(12, zxOutputLow)
Else
    j = 55
    Call PutPin(12, zxOutputHigh)
End If

If i > 3 Then
    Call PutPin(12, zxOutputLow)
ElseIf (i > 0) Then
    j = 0
Else
    j = 55
    Call PutPin(12, zxOutputHigh)
End If
```

Note that the conditional expression is not required to be enclosed in parentheses. Many programmers are accustomed to other languages where they are required and therefore do so out of habit. Others believe that the parentheses improve the readability and use them for that reason. You're free to adopt whichever practice suits you.

One other comment on style is in regard to indentation. The examples used in this document indent the statements within compound statements like If-Then-Else in order to improve readability. The compiler ignores spaces and tabs except to the extent that they separate identifiers, keywords, etc. You're free to adopt any indentation style that you deem appropriate.

There is no fixed limit on how deeply If-Then statements may be nested. The actual limit is governed by how much memory is available to the compiler. For all practical purposes, there is no limit.

2.5.11 Select-Case Statement

The Select-Case compound statement is a multi-way branch statement that can be used in place of an If-Then-ElseIf chain is certain situations. The syntax is shown below.

```
Select Case <test-expr>
Case <case-expr-list>
      [<statements>]
...
Case Else
      [<statements>]
End Select
```

The *<test-expr>* element, known as the selection expression, gives a value that will be tested against the value(s) given in zero or more standard case clauses. Each standard case clause begins with the word `Case` and is followed by a list of one or more expressions, each of which must evaluate to the same type as *<test-expr>*. If multiple expressions are given, they must be separated from one another by a comma. The remainder of the case clause consists of zero or more ZBasic statements. The type of the selection expression may be `Boolean`, an enumeration, any numeric type or `String.` The practical value of using a `Boolean` type is somewhat limited, however – it's simpler to just use an If-Then statement.

There may be at most one default case clause introduced by the keywords `Case Else`. The remainder of the default case clause consists of zero or more ZBasic statements. If the default case clause is present, it must be the final case clause.

The Select-Case statement executes by first evaluating the *<test-expr>*. Then the resulting value is compared with the value of each of the expressions in the *<case-expr-list>* of the first standard case clause, if present. The evaluation of the case expressions and the comparison with the test value is done in order, left to right. As soon a case expression is found whose value is equal to the test value, the statements associated with that case clause are executed and then control transfers to the first statement following the `End Select`. If none of the expressions in the first case clause match the *<test-expr>* value, the process is repeated with the second standard case clause and so on until all of the standard case clauses have been tested. When all of the standard case clauses have been tested without finding a matching expression value, if a default case clause exists the statements associated with it are executed.

There are two special forms of case expressions that may be used in the *<case-expr-list>* of a standard case clause. The first special form is the range expression. This takes the form of two expressions separated by the keyword `To`. Both expressions must evaluate to the same type as *<test-expr>*. The *<test-expr>* value will be deemed to select the case clause if the value is greater than or equal to the value of the expression to the left of the `To` keyword and less than or equal to the value of the expression to the right of the `To` keyword. Effectively, the range expression specifies an inclusive range.

The second special form may be used to implement special test conditions. It has the syntax:

```
Is <conditional-operator> <expression>
```

The *<conditional-operator>* element may be any one of the six conditional operators: =, <>, <, <=, >, and >=. The *<expression>* element must be an expression that evaluates to the same type as the *<test-expr>*. Note that the construction `Is = <expr>` yields the same result as simply specifying the expression value alone.

There is no fixed limit on how deeply Select-Case statements may be nested. The actual limit is governed by how much memory is available to the compiler. For all practical purposes, there is no limit.

Example

```
Select Case i * j
Case 3
    j = 5
    Call PutPin(12, zxOutputLow)
```

```
Case 4, 5 To 20, 27
    j = 1
    Call PutPin(13, zxOutputLow)

Case 3, 100, Is > 200, j
    j = 0

Case Else
    j = -1
End Select
```

In the example above, if the selection expression evaluates to 3 the statements of the first case clause will be executed. The fact that the third case clause also has a case value of 3 is of no consequence. Also note that the case expressions are evaluated every time they are tested. This fact must be kept in mind for two reasons. Firstly, if the case expression contains a variable whose value changes between successive executions of the Select-Case statement (a situation that is strongly discouraged), the case clause that is selected may change even if the selection expression value does not change. Secondly, if any of the case clause expressions involves a function call, the function may or may not be invoked depending on the value of the selection expression and the values of the various expressions in the case clauses preceding it.

BasicX Compatibility Note

In BasicX mode, the use of `String` and `Single` types is not supported nor is the construction `Is` *<op> <expr>*. Moreover, there must be at least one standard case clause.

2.5.12 Set Statement

This statement is only allowed as the first statement of a function that returns an `UnsignedInteger` or `UnsignedLong` type. It must precede all other statements and variable definitions. It is supported for compatibility with BasicX and is required in BasicX compatibility mode but it is otherwise ignored. The syntax is shown below.

```
Set <function-name> = New <type>
```

The *<function-name>* element must match the name of the function containing the `Set` statement and the *<type>* must match the function's type.

Example

```
Function myFunc() as UnsignedInteger
    Set myFunc = New UnsignedInteger

    Dim I as Integer

    <other-statements>
End Function
```

2.5.13 While-Wend Statement

For compatibility with other dialects of Basic, ZBasic includes support for an alternative to the Do While – Loop construct. The syntax is:

```
While <boolean-expression>
      [<statements>]
Wend
```

Note that this compound statement is logically equivalent to the Do While variation of the Do-Loop statement. The one difference is that `Exit Do` cannot be used to terminate a While-Wend statement.

BasicX Compatibility Note

In BasicX mode, the While-Wend statement is not supported.

2.5.14 With Statement

The With statement allows you to use a shorthand notation to refer to some objects. The syntax for the With statement is:

```
With <prefix>
    <other-statements>
End With
```

Between, the `With` and `End With` statements, any reference to an identifier that begins with a period will be treated as if it had the series of characters identified by *<prefix>* immediately preceding the period.

Example

```
tick = Register.RTCTick  ' the long way
With Register
    <other-statements>
    tick = .RTCTick  ' the short way, implies Register.RTCTick
    <other-statements>
End With
```

Note that the entire construct, from `With` to `End With`, is treated much like a compound statement in that it cannot be split across other statement boundaries. It is important to note, however, that this is not a true compound statement with block scoping. Variables and constants defined within a With block are visible to statements that follow it.

In addition to `Register`, other useful *<prefix>* designations are `Console`, `Debug`, `Option`, `Version` and `Module`. Also, the *<prefix>* may specify a portion of a structure member reference, allowing shorthand access to structure members. See Section 3.25 for more information on using structures.

BasicX Compatibility Note

In BasicX compatibility mode only `With Register` is supported.

2.6 Literals

Boolean, numeric and string constant values are often used in programming. These are called literals because the represent the literal value of the number or string that you have in mind as opposed to a variable whose value may change over time.

2.6.1 Boolean Literals

Boolean literals are the keywords `true` and `false`, in upper, lower or mixed case. These literals are of type `Boolean`.

2.6.2 Numeric Literals - Integral Values

A decimal integral numeric literal consists of decimal digits optionally with a leading plus or minus sign to indicate a positive or negative literal value.

Integral literals may be specified in hexadecimal (base 16) by beginning the literal with an ampersand and the letter H (upper or lower case) followed by one or more hexadecimal digits (0-9, A-F, a-f). In BasicX compatibility mode, a trailing ampersand is either allowed, required or disallowed depending on the specific value and whether or not Strict mode is enabled. In native mode the trailing ampersand is always allowed, is never required and has no effect whatsoever on the resulting value.

Integral literals may also be specified in binary (base 2) by beginning the literal with an ampersand and the letter B (upper or lower case) followed by one or more binary digits (0-1). To improve readability, you may also include one or more underscores within the digit string provided that each underscore occurs between two digits. For compatibility with other Basic dialects, an upper or lower case X may be used in place of the radix indicator B. Binary literals are not supported in BasicX compatibility mode.

Examples of decimal, hexadecimal and binary integral literals:
```
124
+16
-357
&HabCd
&H8000&
&B0010_0011
```

2.6.3 Numeric Literals - Real Values

A real numeric literal consists of one or more decimal digits, optionally with a leading plus or minus sign, followed by either a decimal point and one or more decimal digits or the letter `E` followed by one or more decimal digits also optionally prefixed by a plus or minus sign. If the fractional part is present, it may also be followed by an exponent specification. For compatibility with other Basic dialects, a literal that would otherwise be an integral literal will be interpreted as a real literal if it is immediately followed by an exclamation mark. Real literals have the type `Single`.

Examples of real literals:
```
3.14159
+6.02e23
3e10
-300!
```

BasicX Compatibility Note

In BasicX compatibility mode, real literals must either contain a decimal point or have a type designation suffix ! or # to force them to be recognized as type Single. A literal like 12e2 represents the integer value 1200.

2.6.4 String Literals

A string literal consists of zero or more characters enclosed in quotation marks. Note that a string may not be continued on the next line by ending the first line with an underscore. However, you may use the concatenation operator in conjunction with the underscore continuation to span line boundaries. The compiler will combine the operands to the concatenation operator as long as they are both string constants. String literals have the type `String`.

Examples of String Literals:

```
""
"Hello, world!"
"The quick brown fox" & _
" jumped over the lazy dog."
"Hello, ""Joe""!"
```

The third and fourth lines above show how to use the concatenation operator and line continuation to construct longer strings. Note that the underscore must be the last character on the line and that there must be a space or tab character preceding it. The last example shows how to include a quotation mark within a string literal. Two consecutive quote marks are reduced to one in the actual string. If you want two adjacent quote marks in the string, you'll have to double each of them.

2.6.5 Built-in Binary Constants

Although they are technically not numeric literals, ZBasic provides some built-in `Byte` constants that serve the same purpose. The constants begin with the letters `BX` and are followed by exactly 8 binary digits (0-1). There may be an underscore between any pair of binary digits to enhance readability. These constants are of type `Byte` and may only be used where a `Byte` type is allowed. These built-in constants are supported for compatibility with BasicX. It is recommended that new applications use binary literals (described in Section 2.6.2) since they are more generally useful.

Examples

```
Bx0100_1101
BX01_00_11_01
```

2.7 Comments

Comments may be placed on a line by themselves or at the end of a line containing other program text. A comment begins with an apostrophe and continues to the end of the line. A comment may be continued on the next line in the manner described in Section 2.8.

Note: The BasicX compiler does not allow comments to be continued but Visual Basic does. ZBasic allows comment continuation both in native mode and in BasicX compatibility mode.

Examples

```
'This is a comment.
a = 23    ' this is a comment, too
b = 55    ' and because this comment ends with an underscore _
it continues on the next line
```

2.8 Line Continuation and Multiple Statements Per Line

A statement may continued across multiple lines by ending each line except that last with an underscore preceded by at least one space or tab character. Except for spaces, tabs and/or a comment, no characters other than end-of-line characters may follow the underscore for it to be considered a line continuation character. The maximum aggregate size of a line, whether continued across multiple lines or not, is 1000 characters.

In the example below the beginning of the If statement is continued to the following line. This is often useful to help make more complex expressions more readable.

Example

```
If (GetPin(20) = 1) And _
    (GetPin(12) = 0) Then
  Call PutPin(5, 0)
End If
```

While the line continuation capability allows you to create statements that span multiple lines, it is sometimes convenient to place multiple statements on one line. In ZBasic, as in many other Basic dialects, you may accomplish this by using a colon to separate each pair of statements on the line.

Example

```
Dim i as Integer
Dim j as Integer, k as Byte

i = 0 : j = 1 : k = 2
```

2.9 Persistent Variables

You may define variables that are stored in the processor's internal EEPROM, referred to in this document as Persistent Memory. It is called persistent because the values that you store there are retained even if the system is powered down or reset. This characteristic makes persistent variables useful for storing configuration information for your application and other similar information that your application needs to be preserved.

A persistent variable is defined at the module level using the syntax:

{`Public | Private | Dim`*}* `<`*`name`*`> as Persistent <`*`type`*`>`

Using the keyword `Dim` has the same effect as using `Private`. Within a subroutine or function, a persistent variable is defined using the syntax.

`Dim <`*`name`*`> as Persistent <`*`type`*`>`

In both cases, the `<`*`type`*`>` element may be any numeric type (e.g. Byte, Integral, Single, etc.), Boolean or a user-defined type (structure or enumeration). A persistent string must be defined using the bounded string syntax (Section 2.11.1), i.e.

`Dim <`*`name`*`> as Persistent BoundedString(<`*`size-expr`*`>)`

The `<`*`size-expr`*`>` element must be a constant integral expression that specifies the number of bytes to reserve for the persistent string's characters.

Examples

```
Dim kbdAttached as Persistent Boolean
Private signOnMsg as BoundedString(25)
```

It is important to note that the implementation of the PersistentString type is identical to that of the BoundedString type and is therefore not protected from overwriting the boundaries of the data item. To protect against overwriting, it is advisable to explicitly limit the size of the string to be written.

Arrays of persistent variables may be defined as well. To do so, simply add the array dimension list to the variable name in the same manner as for regular variables. If no lower bound is specified, the default array base applies.

Example

```
Dim freq(1 to 10) as PersistentInteger
```

It is important to note that persistent variables are not initialized by the system. They have values based on whatever data happens to be at the Persistent Memory address to which they are assigned. The compiler assigns Persistent Memory addresses in the order that modules are compiled and, within modules, in the order the variables are defined.

To avoid problems of unexpected address order changes, it is highly recommended that all persistent variables be defined in a single module. Also, it is recommended when you add more persistent variables to an existing application that you add them following the definitions of the previously existing persistent variables. Deleting persistent variables or inserting new persistent variables in the midst of existing ones may cause problems because it will change the address to which subsequent variables are assigned.

You'll probably want to build into your application a way to initialize all of your persistent variables to a known state. This initialization only needs to be done once, when the application is first installed (the `FirstTime()` function may be useful for this purpose). It may also be useful, however, to be able to do this at other times as well. Another useful technique is to include a persistent variable whose only purpose is to indicate that the persistent variables have been properly initialized. For this to work, you would need to choose a value that is unlikely to otherwise occur. It may even be advisable to place such a "sentinel" variable at both the beginning and the end of the group of persistent variables, decreasing the likelihood of false positive or false negative indications.

Persistent variables have an associated property named `DataAddress`. The value of this property is the address of the data item in Persistent Memory. The type of the property is `UnsignedInteger` for compatibility with the `PersistentPeek()` subroutine.

Example

```
Dim freq(1 to 10) as PersistentInteger
Dim addr as UnsignedInteger

addr = freq.DataAddress
```

It is possible, also, to use the `DataAddress` property to get the address of an element of a persistent array. To accomplish this, simply add parentheses following the property name and specify the index or indices of the item of interest. The example below will result in `addr` having the Persistent Memory address of the fourth data value of the `freq()` array.

```
addr = freq(4).DataAddress
```

Persistent structures may also be defined, see Section 3.25 for more details. A Persistent variable may also be defined using Based keyword, see Section 3.22 for more details.

For compatibility with BasicX, an alternate syntax is also supported for defining persistent variables as shown below.

```
{Public | Private | Dim} <name> as [New] <persistent-type>
```

Caution: although Persistent Memory data items can be modified, the memory in which they are stored has a write cycle limit of approximately a million writes. Writing to a particular address in excess of this limit may cause the memory to become unreliable. Also, writing to Persistent Memory is much slower than writing to RAM-based variables.

The keyword `New` is optional except in BasicX compatibility mode when it is required. The *`<persistent-type>`* element specifies the type of persistent variable being defined and may be one of the following special types:

```
PersistentBoolean      PersistentByte
PersistentInteger      PersistentLong
PersistentSingle
```

BasicX Compatibility Note

In BasicX mode, all persistent variables must be defined at the module level and neither arrays nor structures of persistent variables are supported. Also, the `DataAddress` property cannot be used to determine the address of a persistent variable.

2.10 Program Memory Data Items

It is often useful to have available initialized arrays of data that are stored in Program Memory. This is advantageous for two reasons. Firstly, you don't need run-time code to initialize the arrays and, secondly, you don't use up the more scarce RAM space for data that seldom, if ever, changes.

You define an initialized Program Memory data item in a manner similar to the way that you define RAM-based data items. There are two differences, however. Firstly, you need to specify a data source from which the array is initialized. Secondly, you don't explicitly specify the array dimensions. Instead, the dimensions are deduced from the content of the initialization data and the lower bound of each index is always 1. The definition syntax for a Program Memory data item is:

```
{Public | Private | Dim} <name> as [New] <progmem-type>(<init-data>)
```

For Program Memory data items defined within a subroutine or function, the Public and Private keywords are disallowed because they would serve no useful purpose.

The supported *<progmem-type>* items are:

One-dimensional types (vector types):
```
ByteVectorData          ByteVectorDataRW
IntegerVectorData       IntegerVectorDataRW
LongVectorData          LongVectorDataRW
SingleVectorData        SingleVectorDataRW
StringVectorData
```

Two-dimensional types (table types):
```
ByteTableData           ByteTableDataRW
IntegerTableData        IntegerTableDataRW
LongTableData           LongTableDataRW
SingleTableData         SingleTableDataRW
StringTableData
```

The types ending with `RW` may be both read and written while the remaining types can read but they cannot have values assigned to them. (Of course, you can still modify the read-only types by using the System Library routine `PutProgMem()` but this is generally only used for special circumstances.)

The initialization data may be provided in two ways. The first way is to provide a file name, enclosed in quote marks, as the *<init-data>* element in the syntax description above. If the filename is not specified using an absolute path (i.e. beginning with the root directory and/or a drive letter), the path prefix (if any) of the current module is appended to the front of the filename. Note, however, that if an include path is specified on the command line, a filename that is specified with a relative path will, instead, be sought in among the directories specified in the include path list. See Section 7.2 for more information on the include path option.

The content of the file should be a textual representation of the initialization data. For integral types, the data values may be expressed in decimal, in hexadecimal (using a &H or &h prefix), or in binary (using a

&B or &b prefix). In the latter case, an underscore may exist between any pair of digits. For real types, the data values may be expressed in integral, decimal or scientific notation format. See the `ValueS()` System Library routine for a description and examples of the acceptable formats of values. For string types, the data values should be zero or more characters enclosed in quotation marks. To include a quotation mark in the string it must appear twice in succession.

For the one-dimensional types, one or more values may be specified per line. When multiple values are given per line they must be separated by a comma and/or white space (space or tab characters). The first value on a line may be preceded by white space. Following the last value on a line, there may be a comma and/or white space and/or a comment (introduced by an apostrophe). The number of elements in the vector will be exactly the number of properly formatted data values in the file.

For the two-dimensional types, the values for each row of the table must be placed on a separate line. The column values on each line must be separated by a comma and/or white space. A comment may follow the last column value on a line. The number of valid column values must be the same for each row.

For either type, the initialization data file may contain blank lines and lines containing only a comment optionally preceded by white space. For the `Byte` types, data values may also be specified using a quoted string. In this case, the ASCII value of each character of the string is used as a data value. As usual, a quote may be included in the string by using two quotes in succession. For the string types, values may be specified by concatenating strings and/or byte values by separating each pair of components with a plus sign. The examples below include samples of each of these special cases.

The second method to provide initialization data is to use an in-line initializer that consists of a pair of curly braces bracketing the initialization data itself. The form of the in-line initializer data is essentially the same as the content of the initialization file described above but appearing between curly braces directly in your source file.

Here is an example of the content of an initialization file for a `ByteVectorData` type:

```
' this is a data file
&H55
2     ' comment
      3

4, 5
   ' another comment
5,
&Haa
```

Below is an example of the initialization data for a `SingleTableData` type:

```
.30103,           3.14159 ' log of 2 and pi
-200.,      1e05
+6.02E+23   100
```

Here are examples of in-line initializers for one-dimensional and two-dimensional types.

```
Dim d1 as ByteVectorData({ 20, &Hff, &H20, "row" })

Dim strList as StringVectorData({
      "alpha", "bravo", "charlie", "delta", "echo", "fox" + &H5f + "trot"
})

Dim tbl as New SingleTableData({
'     column 1          column 2
      .30103,           3.14159
      -200.,            1e05
      +6.02E+23,  100
})
```

The values specified in an in-line initializer may be literal constants as shown above or they may be named constants that are visible within the module. For example,

```
Const cval as Byte = &H20
Dim d1 as ByteVectorData({ 20, &Hff, cval, "row" })
```

Program Memory data items have an associated property named `DataAddress`. The value of this property is the address of the data item in Program Memory. The type of the property is `Long` for compatibility with BasicX and the `GetProgMem()` subroutine.

Example

```
Dim addr as Long

addr = tbl.DataAddress
```

It is possible, also, to use the `DataAddress` property to get the address of a particular Program Memory data item. To accomplish this, simply add parentheses following the property name and specify the index or indices of the item of interest. The example below will result in `addr` having the Program Memory address of the second data value of the first row of the table.

```
addr = tbl.DataAddress(2, 1)
```

Caution: Program Memory data tables are arranged in memory in row-major order, i.e. the column values for the first row, followed by the column values of the second row, etc. This is a direct result of scanning the initialization data row by row. When you index a data table, you must specify the column index first and the row index second. This is backward with the respect to the way matrices are often visualized, i.e. (row, column). This strategy was adopted to maintain compatibility with BasicX. See Section 3.18 for more information on array data order.

Note that the `UBound()` function is useful with Program Memory data items to determine the dimensions of the vectors and tables. `LBound()` will always return 1 since initialized Program Memory data items are always 1-based.

Program Memory variables may also be defined using a syntax similar to that used for defining RAM-based variables, using the keyword attribute ProgMem preceding the type name. For example,

```
Dim d1(1 to 20) as ProgMem Byte
```

This defines and reserves space for an array of bytes in Program Memory. Variables defined in this way will be zero-filled. Strings in Program Memory may be defined as well using the bounded string syntax. In this case, the string will have an initial value representing an empty (zero length) string.

```
Dim ps as ProgMem BoundedString(15)
```

Program memory structures may also be defined, see Section 3.25 for more details. A Program Memory variable may also be defined using Based keyword, see Section 3.22 for more details.

Caution: although Program Memory data items can be modified, the memory in which they are stored has a write cycle limit of approximately a million writes. Writing to a particular address more than this may cause the memory to become unreliable. Also, writing to Program Memory is much slower than writing to RAM-based variables.

BasicX Compatibility Note

In BasicX mode, Program Memory string types are not supported nor are any vector types other than the `Byte` types. Also, in-line initializers are not supported, the `DataAddress` property cannot be used to determine the address of an individual Program Memory data element, and quoted strings cannot be used to specify data values. Finally, Program Memory data items may only be defined at the module level.

You may completely omit the initialization data from the definition of a Program Memory data item, including the parentheses that normally enclose it. If you do this, you must use the `Source` method to specify the initialization data as shown below. This alternate initialization mechanism is supported for backward compatibility with BasicX and is not recommended for new applications. Note, particularly, that this somewhat odd construction involving a `Call` does not produce any run-time executable code. It is merely a signal to the compiler to read the initialization data from the specified file.

```
Dim d1 as New ByteVectorData

Sub Main()
      Dim b as Byte

      ' specify the initialization data
      Call d1.Source("mydata.txt")

      b = d1(2)
End Sub
```

Only one method of specifying the initialization data can be used for any particular Program Memory data item. Attempting to specify the initialization data multiple times will result in a compiler error even if the data supplied in the multiple cases is identical.

2.11 String Types

ZBasic supports several variations of the fundamental type `String`. You may define a string variable thusly:

```
Dim msg as String
```

The amount of space required for this variable definition and the maximum size of the string that it can represent varies depending compiler command line options and Option Directives. By default, the maximum string size is 255 characters. See Section 3.26 for more information on the implementation details of the various string data types.

2.11.1 Bounded Strings

A bounded string is nothing more than a way to specify a string having a maximum length that may be different than the default string length. A bounded string is defined using the syntax:

{`Public | Private | Dim`*} <name>* `as` *[*`New`*]* `BoundedString(`*<size-expr>)*

When defining a bounded string, you replace the *`<size-expr>`* with a constant integral expression specifying the number of bytes to allocate for the string's characters.

Examples

```
Const slen as Integer = 7
Dim msg as BoundedString(15)
Dim msg as BoundedString(slen + 2)
```

The first definition will create a string variable that can hold up to 15 characters; the second will hold 9 characters.

For compatibility with BasicX, the alternate syntax shown below is also supported. New applications should use the definition syntax given above since it allows the use of an expression to specify the length.

```
{Public | Private | Dim} <name> as New BoundedString_<length>
```

2.11.2 Fixed-Length Strings

For compatibility with BasicX, ZBasic supports fixed-length strings. These are similar to bounded strings but with two important distinctions. Firstly, the string size is constant and equal to the specified fixed size. If a string value is assigned that has fewer characters than a fixed-length string variable's specified size, the remaining characters will be filled with spaces. Secondly, if a string value is assigned having more characters than a fixed-length string's specified size, the excess characters will be discarded.

A fixed-length string is defined using the syntax:

```
{Public | Private | Dim} <name> as [New] String * <size-expr>
```

The keyword `New` is optional except in BasicX mode where it is required for compatibility reasons. When defining a fixed-length string the *<size-expr>* should be a constant integral expression specifying the number of bytes to allocate for the string's characters.

Example

```
Dim msg as String * 15
```

This definition will create a string variable that always contains exactly 15 characters.

2.12 Variable Initialization

All statically allocated variables are initialized by the system immediately prior to `Main()` beginning execution. For `String` types, this means that the bytes comprising the variable are set to represent an empty string. For all other types, the constituent bytes are set to zero. Variables defined at the module level and those defined using `Static` within a subroutine or a function are statically allocated and are, therefore, initialized.

Dynamically allocated variables are not initialized by the system except for `String` types which are initialized to represent an empty string. Variables defined using `Dim` within a subroutine or a function are dynamically allocated.

When you define a variable you may provide an initial value by adding an equal sign and the desired value following the variable's type. Initialization is not supported for arrays, structures, Based or Alias variables nor for Program Memory or Persistent Memory data items.

Examples

```
Dim count as Integer = 5
Dim str as String = "column"
```

2.13 Type Conversions

The ZBasic language is strongly typed meaning, for example, that it is not allowed to assign the value of a constant, variable or parameter of one type to a variable or parameter of a different type. There are two apparent exceptions to the strong-type regimen. The first exception is with respect to integral numeric literals. An integral literal is considered to have a universal integral type (32-bit internally) so it can be assigned to a parameter or variable of any integral type (`Byte`, `Integer`, `UnsignedInteger`, etc.). Note that the presence of a plus sign or minus sign on a numeric literal does not change this interpretation so it is allowable to assign the value `–1` to an unsigned variable type.

The second apparent exception to the strong typing rules occurs with the System Library routines. Many of these routines will accept two or more data types for some of their parameters. It is as though several different versions of the library routines exist, differing only in the types of the parameters that they accept. This computer science concept is known as polymorphism.

The System Library routines include a set of functions for performing type conversions. The first set, `CBool()`,`CByte()`, `CInt()`, `CUInt()`, `CLng()`, `CULng()`, `CSng()` and `CStr()` allows, with some exceptions, conversion of a value of an arbitrary type to the target type. The second set, `FixB()`, `FixI()`, `FixUI()`, `FixL()`, and `FixUL()` are specifically for converting `Single` values to the target type. The difference between using `FixI()` and `CInt()`, for example, to convert a `Single` value is the rounding method used. The final set of conversion functions, `CType()` and `To`*`<enum>`*`()` is for converting an integral value to an enumeration member. See the ZBasic System Library Reference Manual for more details on these conversion functions.

2.14 Parameter Passing Conventions

When a subroutine or function is defined, part of the definition specifies the parameters and their types that are expected by the routine. These parameters are referred to as the “formal parameters”. When a subroutine or function is invoked, parameters must be provided that match the formal parameters in order, number and type. These parameters are referred to as the “actual parameters” for each invocation.

As discussed earlier, parameters may be passed to subroutines and functions either “by value” or “by reference”. The differences between these two methods are subtle but important. When passed by value, the actual parameter may be an individual value like a constant or variable or it may be an expression. In either case the value of the actual parameter is calculated and the resulting value is passed to the routine. Within the called routine, the passed value may be utilized in any manner; it may even be modified and this modification will have no effect on any of the constituent elements of the passed parameter value. Effectively, the called routine gets its own private copy of the passed value.

When passed by reference, the address of the actual parameter is passed to the called routine. This implies that the actual parameter must be an individual variable or another parameter and not an expression nor a constant. Since the called routine has the address of the actual parameter, it is able to both read from and write to the actual parameter unless otherwise restricted. The ability of a routine to modify a variable passed by reference is often useful, especially in cases where a routine needs to produce multiple values for use by the caller. On the other hand, if used carelessly, it can be the source of errant program operation that is difficult to diagnose.

There are restrictions on whether a particular variable type may, may not, or must be passed by value or by reference. For example, a persistent variable cannot be passed by reference because the called routine is expecting the address of a RAM-based variable. Also, for efficiency reasons some variables are always passed to routines by providing the variable address, even if the definition of the routine specifies the parameter is to be passed by value. In such cases, the compiler treats the parameter as being read-only, effectively enforcing the semantics of pass-by-value. Any attempt to modify a read-only variable in the called routine or to pass it by reference to another routine will be detected and reported as an error by the compiler. Arrays may only be passed by reference and then only if they are RAM-based, single-dimension and have a lower bound of 1.

Allowed Parameter Passing Methods

Actual Parameter Type	Pass By Value	Pass By Reference
Constant or expression, any type	Yes	No
RAM-based variable or array element	Yes[1]	Yes[2]
RAM-based single-dimension array, 1-based	No	Yes
RAM-based single-dimension array, not 1-based	No	No
RAM-based multi-dimensional array	No	No
Persistent variable or array element	Yes	No
Persistent Memory array	No	No
Program Memory array element	Yes	No
Program Memory array	No	No

Notes:
[1] `String` types and structures are read-only within the called routine when passed by value.
[2] The sub-byte types, `Bit` and `Nibble`, cannot be passed by reference. Also, in BasicX mode, `UnsignedInteger` and `UnsignedLong` types are read-only within the called routine when passed by reference.

The table below gives the number of bytes of stack space required to pass different variable types using the two passing methods.

Stack Usage by Parameter Type and Passing Method

Actual Parameter Type	Pass By Value	Pass By Reference
`Boolean, Byte`	1	2
`Bit, Nibble`	1	2[2]
`Integer, UnsignedInteger, Enum`	2	2
`Long, UnsignedLong, Single`	4	2
`String`[1], structure	2	2
Array, any type	n/a	2

Notes:
[1] Persistent strings, Program Memory strings and strings returned by functions all require 4 bytes of temporary data space (local to the caller) plus the 2-byte reference when passed to a routine other than a System Library routine.
[2] Sub-byte types like Bit and Nibble may only be passed by reference if they are byte aligned. See Section 3.24.1 for details.

2.15 Program and Data Item Properties

Most data items, whether located in RAM, Persistent Memory or Program Memory have an associated property called DataAddress which evaluates to the address of the data item. The DataAddress property is applied to a data item by appending it to the data item's name with a period separating them as illustrated by the example below.

```
Dim b as Byte
Dim addr as UnsignedInteger

Sub Main()
  addr = b.DataAddress
End Sub
```

The DataAddress property can be applied to arrays and structures as well. When used with arrays it is best to append it after the array indices, if any. For most data items, the type of the DataAddress property is UnsignedInteger. However, for compatibility with GetProgMem() and other routines related to Program Memory, the type of the DataAddress property for Program Memory data items is Long.

Along similar lines, subroutines and functions have an associated property called CodeAddress whose type is Long. The CodeAddress property is employed in a similar manner as the DataAddress property is as shown by the example below. Of course, use of the CodeAddress property of a subroutine is not limited to the code in the subroutine itself. It can be applied to any subroutine or function that is visible to the code. In short, if you can invoke the subroutine or function, you can also get it address via the CodeAddress property.

```
Dim addr as Long

Sub Main()
  addr = Main.CodeAddress
End Sub
```

Chapter 3 - Advanced Topics

This chapter provides additional technical information on topics that were introduced earlier in this document. Also, some more advanced concepts are introduced.

3.1 Scope and Lifetime

There are two important attributes of variables that have been alluded to in earlier discussion – scope and lifetime. The scope of a variable reflects its visibility in the sense of where it can be accessed by name. A variable defined within a subroutine or function has local scope meaning that it is only directly accessible to code within that routine. A variable defined outside of any routine has module scope if it is declared `Private` and global scope if it is declared `Public`. Module scope means that only routines within that module can access it directly. Global scope means that any routine in the application can access it directly.

These three scoping levels, global, module and local, form a hierarchy that controls the visibility of the variables. Global scope is at the outermost level of the hierarchy, module scope is at the next inner level and local scope is at the next inner level to that. There are additional inner levels of scope created by compound statements, which topic is discussed further below.

At any particular level of the scoping hierarchy, variables that are in the same scope level or farther outward are visible. It is possible to define variables with the same name at different scoping levels. This does not cause a conflict because the compiler resolves a reference to a particular variable name by searching the current scoping level first and then proceeding outward in the hierarchy until the variable name is found or not as the case may be. Variables are said to “hide” same-named variables that exist at outer scoping levels. In most cases, the hidden variables can still be accessed but more information has to be added to the variable name to clarify to the compiler which variable is being referenced. This concept may be clarified by an example.

Example

Module T1:

```
Public i as Integer
```

Module T2:

```
Private i as Integer

Sub foo()
    Dim i as Integer

    i = 5     ' this refers to the locally defined variable
    t2.i = 5  ' this refers to the private variable at the module level
    t1.i = 5  ' this refers to the public variable in module T1
End Sub
```

The second and third references to the variable `i` in the example above are qualified by the addition of the module name containing the definition of the desired variable. You may add module qualification to any module level and global level variable reference if you wish but it is generally only done when required to resolve the reference to the intended variable.

Although the preceding discussion focused on variables, the same scope concept applies to all identifiers – variables, constants, subroutines and functions. A local constant named `count` will hide a module level variable of the same name. Because the inadvertent hiding of identifiers is a common cause of programming errors the compiler, by default, issues a warning about the hiding. The warning can be disabled if desired. See Section 7.2 for specific information on various compiler options.

The second important attribute is lifetime. This concept refers to how long storage space is reserved for a variable. For variables defined at the module level, the lifetime is indefinite. They exist as long as the program is running. For variables defined within a routine, the lifetime normally begins when the routine begins execution and it ends when the routine finishes execution. Because these variables are dynamically created and destroyed, they are referred to as dynamic variables. This is in contrast to the module level variables which are static variables – they exist for the duration of the program's execution.

Sometimes, it is convenient to have a variable that is visible only to the routine in which it is defined but which is also static. ZBasic supports this concept by allowing the use of the keyword `Static` in place of the keyword `Dim` normally used in a variable definition within a routine. The `Static` keyword tells the compiler to allocate space for the variable alongside the module level variables but since it is defined within a routine, i.e., it has local scope, only the code in that routine can directly access the variable.

Example

```
Private Sub mySub()
      Dim var1 as Integer
      Static var2 as Integer

      [other code here]
End Sub
```

The difference between `var1` and `var2` is that space is allocated on the stack for `var1` when the routine begins executing while the space for `var2` exists as long the program is executing. Another difference is that dynamically allocated variables like `var1` have an undefined value immediately after they are created, you have to add code to initialize them. By default, the compiler will issue a warning if you write code that uses the value of a dynamically allocated variable before it is initialized. Note that `String` variables are a special case in that they are automatically initialized to a zero length.

In contrast, statically allocated variables like `var2` and all module level variables are initialized to zero just before the `Main()` begins running. Each time `mySub()` is invoked, the value of `var1` is undefined but the value of `var2` is whatever was assigned to it last. A consequence of this difference comes into play if `mySub()` is recursively invoked. Each invocation of `mySub()` will have its own private version of `var1` but they will all share the same `var2`.

BasicX Compatibility Note

In BasicX mode, variables cannot be defined as `Static`.

For all compound statements (If-Then, Do-Loop, For-Next, Select-Case and While-Wend), you may define additional variables within the body of the compound statement. When this is done, those variables will only be directly accessible to statements within the compound statement, including any nested compound statements. This is another example of local scope described above. If a variable so defined has the same name as a variable defined in an enclosing compound statement, in the routine itself, or at the module level, the newly defined variable obscures the same-named variable defined at the outer level rendering it inaccessible by normal means.

Example

Module Test:

```
Dim i as Integer

Sub Main()
```

```
    Dim i as Byte, j as Byte

    i = 44
    Test.i = 55
    For j = 0 to 1
        Dim i as Byte

        For i = 1 to 3
            Dim i as String

            i = "Hello"
            Debug.Print i
            Debug.Print CStr(Main.i)
            Debug.Print CStr(Test.i)
        Next i
     Next j
End Sub
```

The first `Debug.Print` will display the string `"Hello"` because it refers to the variable `i` that is defined in the body of the innermost For loop. The second `Debug.Print` will display the value 44 because it refers to the variable `i` defined at the outermost level of the subroutine using the subroutine name qualifier. The third `Debug.Print` will display the value 55 because it refers to the module-level variable `i` using the module qualifier. Within the body of the innermost For loop, there is no way to access the loop index variable because it is hidden by the local definition. There is no qualifier that can be added to a variable reference to resolve to outer block scope levels except the block formed by the routine definition itself.

Since ZBasic implements true block scoping, one advantage to using variables defined within compound statements is that, in addition to the restricted visibility, the stack space used by the variables can be reused by local variables defined in subsequent compound statements.

Example

```
If (j > 5) Then
    Dim i as Byte

    i = 12
    Call PutPin(i, zxOutputLow)
Else
    Dim s as Single

    s = 3.14159
    debug.print CStr(s)
End If

For k = 1 to 10
    Dim b as Byte

    b = GetPin(13)
    Debug.print CStr(b)
Next k
```

In this example, the three variables `i`, s and b share the same stack space. That works because none of them are “active” at the same time.

BasicX Compatibility Note

In BasicX mode, block scoping of variables is not supported. It is permitted to define variables within a compound statement but the effect is the same as if they were defined at the beginning of the routine.

3.2 Enumerations

In some situations it is convenient to be able to refer to the values of a variable by a name rather than by a numeric value. An enumeration type essentially allows you to define a new data type and name the set of values for that type. The syntax for defining an enumeration is:

```
[Public | Private] Enum <name>
      <member-name> [= <constant-expr>]
      ...
End Enum
```

In this syntax, `<member-name>` is an identifier that names a member of the enumeration. The optional `<constant-expr>` represents a value that you want to be associated with that member name. For any member that does not have an explicit member value specified, a member value will be automatically assigned that is one larger than the preceding member or zero for the first member.

The ellipsis in the syntax above indicates that there may be zero or more additional member definitions. Member names must be unique within the set of members for each enumeration. A particular member name may, however, be used in multiple enumerations. See the discussion below for information about how ambiguity is resolved.

If neither `Private` nor `Public` is specified on an enumeration definition, the enumeration is public. An enumeration may be defined at the module level or it may be defined within a subroutine or function, either at the outer level or within any inner block. In the latter case, the `Public` and `Private` keywords have no useful purpose and are therefore not allowed.

Examples

```
Enum Pet
      Dog
      Bird
      Snake
End Enum
```

In this case, the members will be assigned values of 0, 1 and 2 respectively.

```
Enum Mammal
      Cat
      Dog
      Elephant = 5
      Horse
End Enum
```

Here the members will have the values 0, 1, 5 and 6 respectively. Note that if explicit values are specified, they must be larger than the value assigned, explicitly or implicitly, to the preceding member. The value associated with enumeration members is unsigned.

After an enumeration has been defined, the enumeration name may be used as a `<type>` in a variable, constant or structure definition. Enumerations may also be used as the `<type>` in the formal parameter list of a subroutine or function definition. Note, however, that a Public subroutine or function cannot be defined with a parameter that is a Private Enum.

Example

```
Public animal as Mammal

animal = Elephant
animal = Mammal.Dog
```

In the second assignment, the enumeration name is used as a qualifier on the enumeration member name. This is always allowed but it is only required when ambiguity exists. See the discussion below for additional information on resolving ambiguity.

The numeric value of a member may be obtained by using System Library type conversion functions. Continuing the example from above:

```
Dim i as Integer, j as Integer
...
i = CInt(animal)
i = CInt(Mammal.Dog)
```

It is also possible to convert an integral value to an enumeration member. There are two ways to accomplish this. The first, and recommended, way is to use the System Library function `CType()`. This function takes two parameters, the first being the integral value to convert and the second being the name of the enumeration.

```
animal = CType(3, Mammal)
```

The conversion will be performed even if the value specified does not actually correspond to any member of the enumeration so this type of conversion must be used carefully.

Wherever it is used, an enumeration name may be qualified with the module name containing the enumeration. Assume that the enumeration defined above exists in a module named `Test`.

```
animal = Test.Mammal.Dog
animal = CType(3, Test.Mammal)
```

Qualification using the module name is only necessary in unusual cases but it is well to remember that it is allowed for the situations where you need it.

For an enumeration defined within a subroutine or function, the enumeration may be qualified with the subroutine/function name, optionally preceded by a module name qualification. For example, if the enumeration `Mammal` is defined in the subroutine `Main()` contained in the module `Test`, the following constructions are permitted within the `Main()` subroutine.

```
animal = Main.Mammal.Dog
animal = Test.Main.Mammal.Dog
```

When resolving a reference to a member name, the following order is used. If the member reference occurs within a subroutine or function, the name is first checked against the set of members of all enumerations defined within that subroutine or function. If only one enumeration has a matching member name, no ambiguity exists. If no match was found, the name is next checked against the set of members of all enumerations defined to be private to the module. If only one enumeration has a matching member name, no ambiguity exists. If no private enumerations have a matching member, the public enumerations of all modules are checked next. Again, if only one enumeration has a matching member name, no ambiguity exists. This search order may be overridden by qualifying the member name with the name of the enumeration to which it belongs. The reference may be further qualified by adding the module name. This allows access to a public enumeration that is being hidden by an enumeration that is private to the module.

The only operations that can be performed on enumeration variables are comparison using relational operators, assignment and type conversion.

BasicX Compatibility Note

In BasicX compatibility mode, enumerations may only be defined at the module level and qualification of an enumeration is not supported. Also, the values of enumeration members are only 8-bits wide in BasicX while they are 16-bit values in ZBasic.

An alternate type conversion method is supported for compatibility reasons but is a bit awkward to use. For each defined enumeration, there is a special System Library conversion function just for that enumeration whose name is `To<`*`enum`*`>` where `<`*`enum`*`>` is replaced with the enumeration name. This special conversion function takes only one parameter, that being the value to convert.

```
animal = ToMammal(3)
```

One drawback to this conversion method is that there is no way to qualify the enumeration with the module name in which it is defined. It is recommended that all new applications use the `CType()` conversion function.

3.3 Serial Channels

All of the ZX series devices support up to 5 serial channels. Channel 1 (COM1) is implemented using the USART hardware on the microcontroller chip. When the ZX begins running, COM1 is configured to run at 19.2K baud, 8 data bits, 1 stop bit. Your program can send and receive data using the default configuration by utilizing the console I/O routines like Console.Write(), Console.Read() and the related Debug.Print command.

The four remaining serial channels are implemented in software. This strategy allows a lot of flexibility in choosing which pins are used for transmission and reception but it also imposes a processing overhead on the system, even when characters are not being actively sent and received. Because of this, only one additional serial channel, COM3, is enabled for use by default. If you wish to utilize additional serial channels (COM4 to COM6) you must call the System Library routine ComChannels() to specify both the number of channels desired and the maximum baud rate that may be used. See the description of the ComChannels() routine in the ZBasic System Library Reference manual for more details.

Since serial channels 3-6 rely on interrupts to achieve the necessary timing for serial I/O, if interrupts are disabled for a substantial fraction of the bit time (the inverse of the baud rate) the integrity of the transmitted or received characters may suffer. Typically, the maximum acceptable interrupt disable time is about 25% of the bit time of the fastest channel. If the fastest channel is running at 9600 baud, the interrupt disable time should be kept below 25µS or so. Many of the I/O routines that utilize Timer1 (e.g. PulseOut) disable interrupts in order to achieve precise timing. Those that do disable interrupts have a caveat to that effect in their respective descriptions in the ZBasic System Library Manual.

Some ZX devices have additional hardware-based serial channels. See the descriptions of DefineCom(), OpenCom() and CloseCom() in the ZBasic System Library Manual for more details on setting up and using a serial channel.

3.4 Queues

A queue is a fundamental data structure that is widely used in computer programs. Its primary distinguishing feature is that data items are extracted from the queue in the same order in which they were inserted. This is called first-in, first-out or FIFO order.

One of the uses for a queue in ZBasic is as a temporary buffer for data going to and coming from one of the serial channels. Another common use is as a medium through which to pass data between tasks. The producer of the data puts data in the queue and the consumer of the data takes it from the queue.

ZBasic queues are of a fixed length, that length being determined at the time that the queue is prepared for use using `OpenQueue()`. If the data producer inserts data faster than the consumer removes it, the queue will eventually become full. Further attempts to add data to the queue will stall until enough data is removed from the queue to make space for the new data.

Although the data in queues is nominally byte oriented, you can put data of any type into a queue. All that is necessary is for the producer and consumer of the data to agree on the nature and meaning of the data. For example, the producer could copy several `Single` data values to a queue and as long as the consumer copies them out to `Single` variables all will be well.

Prior to using a queue it must be initialized by using the System Library routine OpenQueue(). An example of this is shown below.

```
Dim myQueue(1 to 40)

Call OpenQueue(myQueue, SizeOf(myQueue))
```

This call prepares the queue for use by initializing the first 9 bytes of the specified array with queue management data. The remainder of the array is used for the data to be held by the queue, in this case the queue can hold up to 31 bytes of data since it was defined as being 40 bytes long. This implies that the smallest array usable as a queue is 10 bytes.

Note the use of the `SizeOf()` function. Although the second parameter could just as well have been the literal value 40, if you later changed the size of the myQueue array you'd have to also remember to change the second parameter to match. If you made the queue larger, opening the queue with a smaller value would have no ill effect other than wasting RAM. On the other hand, if you reduced the size of the queue array but left the larger value as the second parameter, the data area inside the queue would overlap adjacent variables leading to puzzling results. For this reason it is highly recommend to use SizeOf() in this and other similar situations rather than hard-coded constants. An alternative method is to use a defined constant as shown below.

```
Const myQueueSize as Integer = 40
Dim myQueue(1 to myQueueSize)

Call OpenQueue(myQueue, myQueueSize)
```

Either of these methods of improving the maintainability of your program is acceptable. Which you choose is more of a stylistic issue than a technical one. The code generated by the compiler in the two cases is identical.

There are several System Library routines available for adding data to and extracting data from a queue as well as some querying functions for determining the status of the queue. Once initialized as shown above, the code fragment below will place some data in the queue.

```
Dim punct(1 to 2) as Byte

punct(1) = &H2c
punct(2) = &H20
Call PutQueueStr(myQueue, "Hello")  ' add a string
Call PutQueue(myQueue, punct, 2)    ' add some individual bytes
Call PutQueueStr(myQueue, "world")  ' add another string
Call PutQueueByte(myQueue, &H21)    ' add an exclamation point
```

After the data is in the queue, the following code fragment will extract it and display it on the console.

```
Do While (StatusQueue(myQueue)  ' add a string
    Dim b as Byte
    Call GetQueue(myQueue, b, 1)    ' retrieve a byte from the queue
    Debug.Print Chr(b);
Loop
Debug.Print
```

It is important to note that the queue insertion routines all wait until there is sufficient space available in the queue for the data being inserted before inserting any data. This may lead to deadlock situations, particularly if the size of the data being inserted is larger than the queue's data area.

3.4.1 System Queues

The console I/O routines like Console.Read() and Console.Write() use the input and output queues associated with Com1. Because of their special use in this manner, the queues associated with Com1 are called system queues. The values `Register.RxQueue` and `Register.TxQueue` give the address of the system input queue and system output queue respectively. The function `CByteArray()` can be used to convert these values to a reference to a Byte array allowing them to be passed to the queue-related routines. The example code below shows how to determine if there is any data available in the system input queue using this technique.

```
If StatusQueue(CByteArray(Register.RxQueue)) Then
    b = Console.Read()
End If
```

The system queues are initially set to be small pre-defined queues. If your code opens Com1, the queues that you provide with the OpenCom() invocation become the system queues until you subsequently close Com1. At that time, the system queues will revert to the pre-defined queues.

Of the two pre-defined queues, the output queue is the smallest, having space for only a few data bytes. Because of this, if you send a fairly long string to the output queue using Console.Write() your application will experience a delay until the string's characters can all be transferred to the output queue. If this causes a problem in your application you can define a larger queue to be used as the output queue. The example code below does so while retaining the pre-defined input queue.

```
Private sysOutQueue(1 to 40) as Byte

Call OpenQueue(sysOutQueue, SizeOf(sysOutQueue))
Call OpenCom(1, CLng(Register.Console.Speed), _
      CByteArray(Register.RxQueue), sysOutQueue)
```

For native mode devices (e.g. the ZX-24n) you can specify the default sizes for the system queues using the directives `Option TxQueueSize` and `Option RxQueueSize`.

3.5 Multitasking

The concept of multitasking is a very powerful one although it can be, at first, somewhat confusing. In a single task system, there is only one program and it runs continuously until it is terminated. During the course of execution of the program it may perform several different functions, e.g. checking the keyboard to see if a character is available, updating the display unit, etc. The fundamental idea behind multitasking is that these activities are divided into related groups, each of which is called a task. Each task executes for a period of time and then the next task executes. There are two basic types of multitasking: cooperative and preemptive. In cooperative multitasking it is left up to each task to determine when to suspend itself and allow the next task to run. In preemptive multitasking a part of the operating system called a task scheduler determines, generally, how long a task is allowed to run before switching to the next task.

The multitasking in the ZX microcontrollers is, fundamentally, preemptive multitasking. The task scheduler switches tasks on each RTC tick (approximately 1.95mS). A task can, however, lock itself to avoid (with some exceptions) giving up execution. This capability should be used sparingly because it defeats the equitable sharing of processing resources.

It is not necessary, of course, to structure your program to utilize the multitasking capabilities. If there is only one task defined (the subroutine Main()) the task scheduler will simply allow it to run continuously. If your application might benefit from being divided into tasks, it is simple to define them. Each task has two core elements: a task main routine and a task stack. The task main routine is the (usually parameterless) subroutine with which the task begins execution. The task stack is a portion of RAM set aside exclusively for the task. The example below shows how to define a task stack and how to activate a second task.

```
Dim taskStack(1 to 80) as Byte

Sub Main()
    CallTask "MyTask", taskStack
    Do
        Debug.Print "Hello from Main"
        Call Delay(1.0)
    Loop
End Sub

Sub MyTask()
    Do
        Debug.Print "Hello from MyTask"
        Call Delay(2.0)
    Loop
End Sub
```

This simple program has two tasks: Main and MyTask. The Main task is created automatically for you and its task stack is automatically allocated all of the remaining User RAM after explicitly defined variables are allocated. In contrast, additional tasks such as MyTask have to be explicitly invoked using the `CallTask` statement and each task's stack must also be explicitly allocated, for example by defining a `Byte` array as shown above. A task is said to be "active" if it has been invoked by `CallTask` and has not yet terminated. Both of the tasks above never terminate so they are always active. The example below shows a situation where a task terminates.

```
Dim taskStack(1 to 50) as Byte

Sub Main()
    CallTask "MyTask", taskStack
    Do
        Debug.Print "Hello from Main"
        Call Delay(1.0)
    Loop
End Sub

Sub MyTask()
    Dim i as Integer
    For i = 1 to 5
        Debug.Print "Hello from MyTask"
        Call Delay(2.0)
    Next i
End Sub
```

In this example, the task `MyTask` will output the message 5 times and then terminate. After the task terminates, its task stack could be used for another task. It is important to note that tasks cannot use the same task stack if they will be active at the same time.

The examples above were chosen because they clearly illustrate, when executed, that the separate tasks experience interleaved execution. A more realistic example is shown (necessarily incompletely) below.

```
Dim taskStack(1 to 50) as Byte

Sub Main()
    CallTask "MyTask", taskStack
    Do
        <other-important-stuff>
    Loop
```

```
End Sub

Sub MyTask()
    ' set the interval timer to 5 seconds
    Call SetInterval(5.0)
    Do
        WaitForInterval()
        <stuff-to-do-periodically>
    Loop
End Sub
```

In this partial example, a task is set up to perform some important activity periodically. If this were part of a system to control the chlorine level in a pool, `MyTask` might read a transducer that indicates chlorine level and, if it's below a certain level, cause a known amount of chlorine solution to be injected into the circulation stream.

One important aspect of creating and using tasks is the allocation of a task stack. The task stack has two components: a task control block and the portion used as an execution stack. The task control block is a fixed size (see Section 3.27 for details) and resides either at the beginning (for VM mode devices) or at end (for native mode devices) of the task stack. The remainder of the task stack is dedicated to the execution stack for the task. The execution stack is used for local variables in the task routine, parameters passed to other subroutines and functions that might be invoked as well as local variables used in those routines, and space required for expression evaluation.

If a task stack is allocated that is much larger than is actually necessary, User RAM is wasted since no other task can use the excess space. On the other hand, if a task stack is smaller than required, data items preceding or following the task stack in memory will be overwritten. This will generally cause your application to misbehave and may result in the processor resetting.

This begs the question, of course, of what is the optimum task stack size. Unfortunately, this is not as easy to answer as it might seem. The difficulty in answering this question lies in the fact that stack use is dynamic and may depend on external factors such as a signal applied by an external device or a user pressing a key. Not only that, the stack use may depend on the order or the timing of such external events. Further, recursive invocation of routines adds to the dynamic nature of stack use and is impossible to account for (in most cases) using any kind of static analysis.

For VM mode ZX devices (e.g. the ZX-24a), the ZBasic compiler automatically estimates the minimum task stack size. Information about the estimate, including the stack use of individual routines, is displayed in the .map file (assuming that one is generated as is the default). If the size of the stack that you have allocated is smaller than the estimated size plus a safety margin, the compiler will generate a warning indicating the deficiency and indicating the minimum allocation required. By default, the safety margin is 10 bytes but you may specify a larger or smaller safety margin (including zero) using Register.StackMargin.

For native mode ZX devices (e.g. the ZX-24n), the minimum task stack size cannot be determined at compile time. To assist you in determining the appropriate task stack size a special System Library function is provided that determines, at run time, how much unused space exists in the task stack. By exercising your application and periodically checking the amount of unused task stack space exists, you may be able to determine empirically a suitable minimum task stack size. See the description of System.TaskHeadRoom()

3.5.1 Advanced Multi-tasking Options

For advanced users, a task's main routine may also be defined with parameters. This may be useful to provide information for the task to perform its function rather than doing so using global variables. Parameters for the task are specified in a comma-separated list enclosed in parentheses just as they are for a subroutine invocation. The number and types of the parameters specified must match those of the task being invoked.

For special circumstances, it is possible to specify the task's stack by giving its address as an integral value. Generally, it will also be advisable to specify the size of the stack since the compiler will be unable to deduce the size. Unless the task stack size is specified or can be deduced, no stack overrun checking can be performed. For native mode ZX devices (e.g. the ZX-24n) the task size must be explicitly specified or it must be determinable by the compiler at compile time, otherwise, the compiler will issue an error message.

The syntax for these advanced options is described in the ZBasic System Library Reference manual. Also, see Section 3.27 for additional information on task management.

3.6 Semaphores

In computer science a semaphore is a mechanism used to control access to a shared resource, the idea being to prevent more than one actor from attempting to use, modify or update a resource simultaneously. As a physical analogy, consider a long single-lane tunnel on a roadway. A driver waiting to enter at one end would like to be certain that no car enters the tunnel from the opposite end while he is enroute. A semaphore indicating that a vehicle is in the tunnel would, if properly observed, help prevent simultaneous use of the shared resource. Clearly, however, if not all drivers understand the meaning of the semaphore or if a driver ignores the semaphore an accident is likely to occur. So it is with semaphores in a computer program.

In a single task system there is no need for semaphores because there are not multiple actors to coordinate. However, in a multitasking system (or a multiprocessor system) because there are multiple actors there is a need to coordinate access to shared resources such as a serial channel, a timer a data structure, etc. Whether or not you need to use a semaphore in your program depends on how it is structured and what resources are shared between tasks.

The essential element of a semaphore is known in computer science as an *atomic test and set* operation. The basic idea is that a method is needed to test a Boolean variable to see if it is already set true and if it is not set then set it to true. The adjective *atomic* in this case refers to the fact that the testing phase and the setting phase are indivisible. That is, once a task begins the process of testing, no other task is able to begin testing until the first task has completed the test and set. This prevents what is called a *race condition*.

The atomic test and set operation in ZBasic is provided by the System Library routine `Semaphore()`. To implement a semaphore you must define a `Boolean` variable and ensure that it is set to `False` initially. Then, whenever a task wants to use the shared resource it must first call Semaphore() passing the previously defined variable as a parameter. If the semaphore is already set, the function will return `False` indicating that the resource is busy. If the semaphore is not already set, the function will set it to `True` and return `True` indicating that the semaphore has been successfully obtained. Once the task is finished with the resource, it must set the semaphore variable back to False again so that the next user may successfully acquire a semaphore on the resource. The example code below illustrates the sequence.

```
Dim serSem as Boolean

serSem = False

' wait until we get the semaphore
Do While (Not Semaphore(serSem))
Loop

' now we can use the controlled resources
[add code here]

' finished with the resources, release the semaphore
serSem = False
```

3.7 Timers

The ZX series CPUs both incorporate several timers, the actual number depending on the underlying CPU. On all ZX models, Timer0 and Timer2 are 8-bit timers while the remaining timers are 16-bit. The table below summarizes timer usage.

Timer Usage by CPU Type

Underlying CPU	RTC	I/O	Serial	PWM	InputCapture	OutputCapture
mega32	Timer0	Timer1	Timer2	Timer1	Timer1	Timer1
mega644	Timer0	Timer1	Timer2	Timer1	Timer1	Timer1
mega128	Timer0	Timer1	Timer2	Timer1/3	Timer1/3	Timer1/3
mega1281	Timer2	Timer4	Timer0	Timer1/3	Timer1/3	Timer1/3
mega1280	Timer2	Timer4	Timer0	Timer1/3/4/5	Timer1/3/4/5	Timer1/3/4/5

The RTC Timer is used to generate interrupts to manage the real time clock (RTC). The I/O Timer is used for several I/O functions that require accurate timing. The Serial Timer is used to implement the "software UART" for the serial channels Com3 to Com6. When a specific timer is not being used by the system, you are free to use that timer in your program as you wish. To do so, you'll have to read the Atmel datasheet for the ATmega32, ATmega644, ATmega128, ATmega1280 or ATmega1281 microcontroller to determine how to program the timers and then use the built-in registers to read and write the timer registers. Further discussion of that topic is beyond the scope of this document.

3.8 Built-in Variables

The set of pre-defined registers comprises two sub-groups: actual CPU registers and control program variables. All of these built-in variables must be referenced using the `Register` prefix or within a `With Register` compound statement.

Example

```
Dim tick as Long

tick = Register.RTCTick
```

3.8.1 CPU Registers

ZX-24, ZX-40, and ZX-44 Registers

The pre-defined CPU registers for the ZX-24, ZX40 and ZX-44 match those available on the ATmega32 CPU and all are `Byte` values. Discussion of the use of these registers is beyond the scope of this document. See the Atmel documentation for a full description. Also, be advised that modifying some of these registers may severely alter the behavior of the control program, possibly even rendering it inoperable.

The register value may be accessed by using the register name, from the table below, prefixed by the keyword `Register`, e.g. `Register.PortC`.

CPU Registers for mega32-based ZX Models

ACSR	EECR	OCR1AL	PORTD	TCCR2	UBRRH
ADCH	EEDR	OCR1BH	SFIOR	TCNT0	UBRRL
ADCL	GICR	OCR1BL	SPCR	TCNT1H	UCSRA
ADCSRA	GIFR	OCR2	SPDR	TCNT1L	UCSRB
ADMUX	GIMSK	OSCCAL	SPH	TCNT2	UCSRC
ASSR	ICR1H	PINA	SPL	TIFR	UDR
DDRA	ICR1L	PINB	SPMCR	TIMSK	WDTCR
DDRB	MCUCR	PINC	SPSR	TWAR	

DDRC	MCUSR	PIND	SREG	TWBR
DDRD	MCUCSR	PORTA	TCCR0	TWCR
EEARH	OCR0	PORTB	TCCR1A	TWDR
EEARL	OCR1AH	PORTC	TCCR1B	TWSR

Some CPU registers are 16-bit values that have register names in the table above corresponding to the high and low bytes. For convenience of access, the following `UnsignedInteger` register names are available. When reading and writing to these registers, the control program ensures that the high and low bytes are accessed in the correct order as required by the CPU. Note that UBRR cannot be supported because the high and low bytes are not adjacent in the I/O address space.

```
ADC     EEAR    ICR1    OCR1A    OCR1B   SP      TCNT1
```

Because the ZX microcontrollers are implemented using a more powerful CPU, some registers available in BasicX don't actually exist on the ZX CPU but there are equivalent registers. For compatibility, both the original register name and the actual register name are supported. For new applications, the new register names should be used.

BasicX Register	New Register
ADCSR	ADCSRA
GIMSK	GICR
MCUSR	MCUCSR
UBRR	UBRRL[1]
UCR	UCSRB
USR	UCSRA

Notes:
[1] The serial port baud rate control register is 16-bits wide in the mega32 CPU. The upper bits are referred to as `UBRRH`.

ZX-24a, ZX-40a, and ZX-44a Registers

The pre-defined CPU registers for the ZX-24a, ZX40a and ZX-44a match those available on the ATmega644 CPU and all are `Byte` values. Discussion of the use of these registers is beyond the scope of this document. See the Atmel documentation for a full description. Also, be advised that modifying some of these registers may severely alter the behavior of the control program, possibly even rendering it inoperable.

CPU Registers for mega644-based ZX Models

ACSR	EEDR	OCR1AH	PIND	TCCR1A	TWAR
ADCH	EICRA	OCR1AL	PORTA	TCCR1B	TWBR
ADCL	EIFR	OCR1BH	PORTB	TCCR1C	TWCR
ADCSRA	EIMSK	OCR1BL	PORTC	TCCR2A	TWDR
ADCSRB	GPIOR0	OCR2A	PORTD	TCCR2B	TWSR
ADMUX	GPIOR1	OCR2B	PRR0	TCNT0	UBRR0H
ASSR	GPIOR2	OSCCAL	SMCR	TCNT1H	UBRR0L
DDRA	GTCCR	PCICR	SPCR0	TCNT1L	UCSR0A
DDRB	ICR1H	PCIFR	SPDR0	TCNT2	UCSR0B
DDRC	ICR1L	PCMSK0	SPH	TIFR0	UCSR0C
DDRD	MCUCR	PCMSK1	SPL	TIFR1	UDR0
DIDR0	MCUSR	PCMSK2	SPMCSR	TIFR2	WDTCSR
DIDR1	MONDR	PCMSK3	SPSR0	TIMSK0	
EEARH	OCDR	PINA	SREG	TIMSK1	
EEARL	OCR0A	PINB	TCCR0A	TIMSK2	
EECR	OCR0B	PINC	TCCR0B	TWAM	

The 16-bit registers for the mega644-based ZX models are shown in the table below.

ADC	EEAR	ICR1	OCR1A	OCR1B	SP	TCNT1	UBRR0

ZX-24p, ZX-40p, ZX-44p, ZX-24n, ZX-40n and ZX-44n Registers

The pre-defined CPU registers for these devices match those available on the ATmega644P CPU and all are `Byte` values. Discussion of the use of these registers is beyond the scope of this document. See the Atmel documentation for a full description. Also, be advised that modifying some of these registers may severely alter the behavior of the control program, possibly even rendering it inoperable.

CPU Registers for mega644P-based ZX Models

ACSR	EEDR	OCR1AH	PIND	TCCR1A	TWAR	UDR1
ADCH	EICRA	OCR1AL	PORTA	TCCR1B	TWBR	WDTCSR
ADCL	EIFR	OCR1BH	PORTB	TCCR1C	TWCR	
ADCSRA	EIMSK	OCR1BL	PORTC	TCCR2A	TWDR	
ADCSRB	GPIOR0	OCR2A	PORTD	TCCR2B	TWSR	
ADMUX	GPIOR1	OCR2B	PRR0	TCNT0	UBRR0H	
ASSR	GPIOR2	OSCCAL	SMCR	TCNT1H	UBRR0L	
DDRA	GTCCR	PCICR	SPCR0	TCNT1L	UCSR0A	
DDRB	ICR1H	PCIFR	SPDR0	TCNT2	UCSR0B	
DDRC	ICR1L	PCMSK0	SPH	TIFR0	UCSR0C	
DDRD	MCUCR	PCMSK1	SPL	TIFR1	UDR0	
DIDR0	MCUSR	PCMSK2	SPMCSR	TIFR2	UBRR1H	
DIDR1	MONDR	PCMSK3	SPSR0	TIMSK0	UBRR1L	
EEARH	OCDR	PINA	SREG	TIMSK1	UCSR1A	
EEARL	OCR0A	PINB	TCCR0A	TIMSK2	UCSR1B	
EECR	OCR0B	PINC	TCCR0B	TWAM	UCSR1C	

The 16-bit registers for the mega644-based ZX models are shown in the table below.

ADC	EEAR	ICR1	OCR1A	OCR1B	SP	TCNT1	UBRR0	UBRR1

ZX-1281 and ZX-1281n Registers

The pre-defined CPU registers for the ZX-1281 and ZX-1281n match those available on the ATmega1281 CPU and all are `Byte` values. Discussion of the use of these registers is beyond the scope of this document. See the Atmel documentation for a full description. Also, be advised that modifying some of these registers may severely alter the behavior of the control program, possibly even rendering it inoperable.

CPU Registers for mega1281-based ZX Models

ACSR	EECR	MCUSR	OCR4AL	PINE	SREG	TCNT2	TWAR
ADCH	EEDR	MONDR	OCR4BH	PINF	TCCR0A	TCNT3H	TWBR
ADCL	EICRA	OCDR	OCR4BL	PING	TCCR0B	TCNT3L	TWCR
ADCSRA	EICRB	OCR0A	OCR4CH	PORTA	TCCR1A	TCNT4H	TWDR
ADCSRB	EIFR	OCR0B	OCR4CL	PORTB	TCCR1B	TCNT4L	TWSR
ADMUX	EIMSK	OCR1AH	OCR5AH	PORTC	TCCR1C	TCNT5H	UBRR0H
ASSR	EIND	OCR1AL	OCR5AL	PORTD	TCCR2A	TCNT5L	UBRR0L
CLKPR	GPIOR0	OCR1BH	OCR5BH	PORTE	TCCR2B	TIFR0	UBRR1H
DDRA	GPIOR1	OCR1BL	OCR5BL	PORTF	TCCR3A	TIFR1	UBRR1L
DDRB	GPIOR2	OCR1CH	OCR5CH	PORTG	TCCR3B	TIFR2	UCSR0A
DDRC	GTCCR	OCR1CL	OCR5CL	PRR0	TCCR3C	TIFR3	UCSR0B
DDRD	ICR1H	OCR2A	OSCCAL	PRR1	TCCR4A	TIFR4	UCSR0C
DDRE	ICR1L	OCR2B	PCICR	RAMPZ	TCCR4B	TIFR5	UCSR1A
DDRF	ICR3H	OCR3AH	PCIFR	SMCR	TCCR4C	TIMSK0	UCSR1B
DDRG	ICR3L	OCR3AL	PCMSK0	SPCR	TCCR5A	TIMSK1	UCSR1C
DIDR0	ICR4H	OCR3BH	PCMSK1	SPDR	TCCR5B	TIMSK2	UDR0

DIDR1	ICR4L	OCR3BL	PINA	SPH	TCCR5C	TIMSK3	UDR1
DIDR2	ICR5H	OCR3CH	PINB	SPL	TCNT0	TIMSK4	WDTCSR
EEARH	ICR5L	OCR3CL	PINC	SPMCSR	TCNT1H	TIMSK5	XMCRA
EEARL	MCUCR	OCR4AH	PIND	SPSR	TCNT1L	TWAMR	XMCRB

The 16-bit registers for the mega1281-based ZX models are shown in the table below.

ADC	ICR5	OCR3B	OCR5A	TCNT3
EEAR	OCR1A	OCR3C	OCR5B	TCNT4
ICR1	OCR1B	OCR4A	OCR5C	TCNT5
ICR3	OCR1C	OCR4B	SP	UBRR0
ICR4	OCR3A	OCR4C	TCNT1	UBRR1

ZX-1280 Registers

The pre-defined CPU registers for the ZX-1281 match those available on the ATmega1280 CPU and all are `Byte` values. Discussion of the use of these registers is beyond the scope of this document. See the Atmel documentation for a full description. Also, be advised that modifying some of these registers may severely alter the behavior of the control program, possibly even rendering it inoperable.

CPU Registers for mega1280-based ZX Models

ACSR	EEARH	MCUSR	OCR4BL	PINJ	SPMCSR	TCNT2	TWCR
ADCH	EEARL	MONDR	OCR4CH	PINK	SPSR	TCNT3H	TWDR
ADCL	EECR	OCDR	OCR4CL	PINL	SREG	TCNT3L	TWSR
ADCSRA	EEDR	OCR0A	OCR5AH	PORTA	TCCR0A	TCNT4H	UBRR0H
ADCSRB	EICRA	OCR0B	OCR5AL	PORTB	TCCR0B	TCNT4L	UBRR0L
ADMUX	EICRB	OCR1AH	OCR5BH	PORTC	TCCR1A	TCNT5H	UBRR1H
ASSR	EIFR	OCR1AL	OCR5BL	PORTD	TCCR1B	TCNT5L	UBRR1L
CLKPR	EIMSK	OCR1BH	OCR5CH	PORTE	TCCR1C	TIFR0	UCSR0A
DDRA	EIND	OCR1BL	OCR5CL	PORTF	TCCR2A	TIFR1	UCSR0B
DDRB	GPIOR0	OCR1CH	OSCCAL	PORTG	TCCR2B	TIFR2	UCSR0C
DDRC	GPIOR1	OCR1CL	PCICR	PORTH	TCCR3A	TIFR3	UCSR1A
DDRD	GPIOR2	OCR2A	PCIFR	PORTJ	TCCR3B	TIFR4	UCSR1B
DDRE	GTCCR	OCR2B	PCMSK0	PORTK	TCCR3C	TIFR5	UCSR1C
DDRF	ICR1H	OCR3AH	PCMSK1	PORTL	TCCR4A	TIMSK0	UDR0
DDRG	ICR1L	OCR3AL	PINA	PRR0	TCCR4B	TIMSK1	UDR1
DDRH	ICR3H	OCR3BH	PINB	PRR1	TCCR4C	TIMSK2	WDTCSR
DDRJ	ICR3L	OCR3BL	PINC	RAMPZ	TCCR5A	TIMSK3	XMCRA
DDRK	ICR4H	OCR3CH	PIND	SMCR	TCCR5B	TIMSK4	XMCRB
DDRL	ICR4L	OCR3CL	PINE	SPCR	TCCR5C	TIMSK5	
DIDR0	ICR5H	OCR4AH	PINF	SPDR	TCNT0	TWAMR	
DIDR1	ICR5L	OCR4AL	PING	SPH	TCNT1H	TWAR	
DIDR2	MCUCR	OCR4BH	PINH	SPL	TCNT1L	TWBR	

The 16-bit registers for the mega1280-based ZX models are shown in the table below.

ADC	ICR5	OCR3B	OCR5A	TCNT3
EEAR	OCR1A	OCR3C	OCR5B	TCNT4
ICR1	OCR1B	OCR4A	OCR5C	TCNT5
ICR3	OCR1C	OCR4B	SP	UBRR0
ICR4	OCR3A	OCR4C	TCNT1	UBRR1

3.8.2 Control Program Variables

Some of the ZBasic register values allow access to certain variables used by the system control program as described below. Unless otherwise indicated, the registers are readable and writable. However, for some of the variables, modifying their values may have undesirable or unpredictable effects on your program. See the descriptions of the individual items for more information.

BasicX Compatibility Note

In BasicX mode, only a subset of the variables described below is available, being limited to those that are also present in BasicX. Those that are available in BasicX mode are specifically identified in their descriptions.

`Register.ResetFlags`

Whenever the processor is reset, the cause of the reset is noted and stored in an internal variable that is available to user programs as `Register.ResetFlags`. The bits of the `Byte` value have the meaning shown in the table below.

ResetFlags Bit Semantics

Reset Source	Hex Value
WatchDog Reset	&H08
Brown-Out Reset	&H04
External Reset	&H02
Power-On Reset	&H01

The value of this register is set when the processor first begins executing the control program. The value is not used by the control program in any manner so you may modify it to suit the needs of your application. Note that in some circumstances, two or more of the bits in the table above may be present.

`Register.RTCDay`
`Register.RTCTick`

These two register values (both available in BasicX compatibility mode) represent the current state of the real time clock (RTC). `Register.RTCTick` is a `Long` value that is incremented on each RTC tick (see `Register.RTCTickFrequency`). After 24 hours of continuous execution, it will reach its maximum value and will then roll over to zero. At the same time, the value of `Register.RTCDay`, type `UnsignedInteger`, will be incremented. Day number zero represents January 1, 1999 (for compatibility with BasicX). When the system is reset or powered up both of these values are initialized to zero.

`Register.RTCFastTick`

This `Byte` value changes twice as fast as `Register.RTCTick`. For most purposes, this value should be considered to be read-only. Changing it will affect the accuracy of the RTC and may interfere with normal task switching.

`Register.RTCStopWatch`

This `UnsignedInteger` value is incremented on each RTC tick (see `Register.RTCTickFrequency`), the same as `Register.RTCTick`. However, you may reset this value to zero at any time to facilitate simpler elapsed time calculations without affecting the RTC's timekeeping. This register variable is available in BasicX compatibility mode.

`Register.RTCTickFrequency`

This read-only `UnsignedInteger` value indicates the number of RTC ticks that will occur per second. For all ZX models currently available, the RTC tick frequency is 512Hz.

Register.SeedPRNG

This register, having type `Long`, represents the “seed” value used by the built-in pseudo-random number generator. With a given seed value, the random number generator will always return the same sequence of values. Usually, you wouldn’t want this type of repeatability but for some purposes it is useful. See the descriptions for the System Library routines `Rnd()` and `Randomize()` for more details.

Register.Timer0Busy
Register.Timer1Busy
Register.Timer2Busy
Register.Timer3Busy
Register.Timer4Busy
Register.Timer5Busy

These `Boolean` values indicate when the processor’s built-in timers are being used. For example, depending on the model, either Timer0 or Timer2 is used by the Real Time Clock. Your code can pass one of these register values as the parameter to the `Semaphore()` function in order to get exclusive access to the corresponding timer. See Section 3.7 for more information on Timer use.

Register.TimerSpeed1
Register.TimerSpeed2

These two registers, both Byte values, represent the Timer pre-scaler value used by several System Library routines. Setting the value of these registers other than by direct assignment will produce undefined results. See Section 3.7 for more information on Timer use.

Register.TaskMain
Register.TaskCurrent

These `UnsignedInteger` registers contain the address of the task control block for the `Main()` task and the current task respectively. The values, which are read-only, can be passed to the various task management functions by using the System Library function `CByteArray()`. See Section 3.27 for more details on Task Management.

Register.StackMargin

This `Byte` value specifies how close to the end of the stack the stack pointer for a task may approach before triggering a stack fault. The default value is 6. See Section 3.11 for more information on Run Time Stack Checking. This built-in is useful only for VM code devices such as the ZX-24.

Register.FaultType
Register.FaultData
Register.FaultData2

These values give information about the last detected system fault. `Register.FaultType` is a `Byte` value that indicates the fault type. `Register.FaultData` and `Register.FaultData2` are `UnsignedInteger` values that provide additional data about the fault. See Section 3.11 for more information on Run Time Stack Checking.

Register.RxQueue
Register.TxQueue

These UnsignedInteger registers contain the address of the queues currently associated with Com1. These queues used for Console.Read(), Console.Write() and related routines. The values, which are read-only, can be passed to the various queue functions by using the System Library function CByteArray(). See Section 3.4.1 for more information on the system queues.

Register.Console.EOL

This Byte value represents the character that the system will recognize as the end-of-line character. It is initially set to the value of a line feed character (&H0a). See the discussion of Console.ReadLine() in the ZBasic System Library Reference for more information on how it is used.

Register.Console.Echo

This Boolean value, initially set to True, controls whether characters received by Console.Read() and Console.ReadLine() are echoed back to the sending device. See the descriptions of these two functions in the ZBasic System Library Reference for more information.

Register.Console.Speed

This UnsignedInteger value gives the default console speed. This is the baud rate for which Com1 is configured when the system begins running. The value is read-only.

Register.SignOn

This Boolean register contains the flag that controls whether or not the ZX issues a sign-on message when it starts up after reset. It is a Persistent Memory value that may also be set or cleared by using the Option SignOn directive. See Section 2.3.1 for more information.

Register.CodeSize
Register.RamSize
Register.RamStart
Register.RamUsed
Register.PersistentSize
Register.PersistentStart
Register.PersistentUsed

These system values may be useful for diagnostic and other purposes. Register.CodeSize is a Long constant that indicates the number of bytes of Program Memory consumed by your program together with any Program Memory data items that it defines.

Register.RamSize is an UnsignedInteger constant that indicates the total number of bytes of User RAM that is directly available to your program. Register.RamStart is an UnsignedInteger constant that indicates address at which User RAM begins. Register.RamUsed is an UnsignedInteger constant that indicates the total number of bytes of User RAM that your program statically allocates. The difference between Register.RamSize and Register.RamUsed represents the number of bytes that will be allocated automatically for the task stack for the Main() task.

Similarly, Register.PersistentSize is an UnsignedInteger value that indicates the size of Persistent Memory, in bytes, that is available to your program. Register.PersistentStart is an UnsignedInteger value that indicates address at which User Persistent Memory begins and Register.PersistentUsed indicates the number of byte of Persistent Memory actually used by your program. These values are all read-only.

`Register.CPUFrequency`

This read-only `UnsignedLong` value indicates the frequency of the CPU clock, in Hertz. For all ZX models currently available, the CPU frequency is 14,745,600Hz.

`Register.HeapEnd`

This read-only `UnsignedInteger` value indicates the lower bound of the memory allocation heap. The heap grows from the high end of RAM toward the beginning of RAM. As blocks of memory are allocated from the heap (for `String` variables or to satisfy `System.Alloc()` requests), the value of `Register.HeapEnd` will decrease. As blocks of memory are returned to the heap (as `String` variables change or go out of scope, or due to calls to `System.Free()`), the value of `Register.HeapEnd` may or may not increase. The value of `Register.HeapEnd` will not increase until the allocated block of memory closest to its current value is freed. At that time, `Register.HeapEnd` will increase to be near the lowest still-allocated block.

`Register.ExtRamConfig`

This `UnsignedInteger` value indicates the current external RAM configuration. Of course, it is only meaningful on ZX models that support external RAM, e.g. the ZX-1281. Although you may change this value during execution, the configuration of the external RAM interface will not be affected until the next time the ZX resets (i.e. following power up, WatchDog reset, download, etc.).

`Register.FirstTime`

This read-only Boolean value is identical to that which would be returned by the `FirstTime()` function. Note, however, that reading the value via `Register.FirstTime` does not reset the flag like invoking the `FirstTime()` function does.

`Register.UserSP`

This `UnsignedInteger` register contains the value of the stack pointer for the current task at the moment it is referenced. This value may be useful in estimating the stack usage for a task. For native mode devices (e.g. the ZX-24n) the value of Register.UserSP is identical to Register.SP.

Caution: modifying this value will probably cause your program to malfunction.

`Register.SP`

This `UnsignedInteger` register contains the value of the hardware stack pointer of the underlying processor at the moment it is referenced. For VM code devices, there are few, if any, practical uses for this value. See, instead, Register.UserSP. For native mode devices, the value of Register.SP is identical to Register.UserSP and indicates the value of the current task's stack pointer at moment it is referenced.

Caution: modifying this value will probably cause your program to malfunction.

3.9 Built-in Constants

The compiler has several built-in constants. These constants may be used in conditional directives and they may also be used as if they were constants defined by the normal means. The entries in the table below that begin with `Option` typically derive their value from the corresponding Option directives, the corresponding compiler command line options or their respective default values.

Name	Type	Description
`Module.Name`	String	The name of the module being compiled.
`Option.Strict`	Boolean	Indicates if Strict mode is in effect.
`Option.AllocStr`	Boolean	Indicates if dynamic string allocation is in effect.
`Option.StringSize`	Integer	Gives the default string size for static string allocation.
`Option.Base`	Integer	Gives the array base value for the current module.
`Option.ExtRamEnabled`	Boolean	Indicates if external RAM use is enabled.
`Option.HeapLimit`	UnsignedInteger	Indicates the specified heap limit.
`Option.HeapSize`	UnsignedInteger	Indicates the specified heap size.
`Option.Language`	String	Specifies the language mode that is in effect.
`Option.MainTaskStackSize`	String	Indicates the specified size for the main task stack.
`Option.TargetCode`	String	Specifies the type of code being generated (“ZVM” or “Native”).
`Option.TargetDevice`	String	Specifies the ZX device for which the code is being compiled.
`Option.CPUType`	String	Specifies the CPU used on the target device.
`Option.CodeLimit`	UnsignedLong	Specifies the code size limit (zero if none is specified).
`Pin.GreenLED`	Byte	Gives the pin designator for the green LED (if available).
`Pin.RedLED`	Byte	Gives the pin designator for the red LED (if available).
`System.JumpBufSize`	Integer	Gives the size required for a buffer used for SetJmp().
`System.MinQueueSize`	Integer	Gives the minimum size for an array to be used for a queue.
`Version.Major`	Integer	Gives the major portion of the compiler version number.
`Version.Minor`	Integer	Gives the minor portion of the compiler version number.
`Version.Variant`	Integer	Gives the variant portion of the compiler version number.

With regard to the built-in constants referring to the version number, consider the version number v1.2.0. In this version number, the major value is 1, the minor value is 2 and the variant is 0.

Additionally, for each target device there exists a set of built-in pin name constants of the form `<port>.<bit> where <port>` is a letter (case insignificant) referring to an I/O port and `<bit>` is a digit in the range 0-7 referring to a bit of the port. The value of the pin name is of type `Byte` and may be the pin number for the target device corresponding to that port bit or it may be an encoded port/pin designator (as controlled by `Option PortPinEncoding`). For example, when the target device is ZX24, the pin name `C.7` has the value 5 if `Option PortPinEncoding` is off.

3.10 Exception Handling

ZBasic implements a simple but effective form of exception handling using a concept borrowed from the C language. In the normal execution of a program, the call-return process is orderly and rigid. A routine can only return directly to the routine that called it. This forms a natural hierarchy that works well in most cases. However, it is sometimes the case that your program will detect a set of circumstances that preclude normal operation. In such cases, it is useful to be able to discard the normal call-return hierarchy and return directly to some distant caller, sending back a value to indicate the nature of the conditions that required the exceptional action.

The simple example below shows how to utilize the exception handling mechanism.

```
Dim jmpBuf1(1 to System.JumpBufSize) as Byte

Sub Main()
    debug.print "start test"

    ' initialize the jump buffer
    Select Case SetJmp(jmpBuf1)
    Case 0
        ' control came back from SetJmp()
        debug.print "calling foo()"
        Call foo()
        debug.print "normal return from foo()"
    Case 1
        ' control came back from LongJmp()
        debug.print "SetJmp() returned 1 via LongJmp()"
    End Select

    debug.print "end test"
End Sub

Sub foo()
    debug.print "in foo()"
    Call bar()
    debug.print "returning from foo()"
End Sub

Sub bar()
    debug.print "in bar()"
    Call LongJmp(jmpBuf1, 1)
    debug.print "returning from bar()"
End Sub
```

Running this program will result in the following debug output:

```
start test
calling foo()
in foo()
in bar()
SetJmp() returned 1 via LongJmp()
end test
```

Note that the last lines of neither `foo()` nor `bar()` were executed nor was the line after the call to `foo()` in `Main()`. If you comment out the call to `LongJmp()` in `bar()`, the program will produce the following debug output:

```
start test
calling foo()
in foo()
in bar()
returning from bar()
returning from foo()
normal return from foo()
end test
```

Note that a call to `LongJmp()` generally should not utilize a value of zero as the second parameter. Doing so will make it appear as though the original `SetJmp()` call is returning.

The jump buffer may also be a local variable if desired. The example code below is a modified version of the previous example showing how the jump buffer is passed down the hierarchy as a parameter. This technique may be used to create generalized subroutines that might return to one of several places depending on how it was called.

```
Sub Main()
    Dim jmpBuf1(1 to System.JumpBufSize) as Byte

    debug.print "start test"
    ' initialize the jump buffer
    Select Case SetJmp(jmpBuf1)
    Case 0
        ' control came back from SetJmp()
        debug.print "calling foo()"
        Call foo(jmpBuf1)
        debug.print "normal return from foo()"
    Case 1
        ' control came back from LongJmp()
        debug.print "SetJmp() returned 1 via LongJmp()"
    End Select

    debug.print "end test"
End Sub

Sub foo(ByRef jb() as Byte)
    debug.print "in foo()"
    Call bar(jb)
    debug.print "returning from foo()"
End Sub

Sub bar(ByRef jb() as Byte)
    debug.print "in bar()"
    Call LongJmp(jb, 1)
    debug.print "returning from bar()"
End Sub
```

3.11 Run Time Stack Checking

On VM mode devices (e.g. the ZX-24a), the control program on the ZX implements optional run-time stack overflow detection. You can enable and disable the checking at any time using the System Library subroutine `StackCheck()`. There is a small performance penalty for the stack check although the checking is only done after operations that add data to the stack.

When stack checking is enabled the stack pointer is compared against the "end of stack" value in the current task's Task Control Block less the current setting of `Register.StackMargin`. If the stack pointer exceeds the limit a stack fault is generated and the processor is reset. When the processor begins running again `Register.ResetFlags` will indicate that a WatchDog reset occurred and `Register.FaultType` will have the value 1 indicating a stack fault condition occurred. Also, `Register.FaultData` will contain the address of the Task Control Block of the task that was running when the stack fault was detected. `Register.FaultData2` will contain the address of the instruction that was executing at the time of the fault. Note that `Register.FaultType` is a persistent value. If you add code to your application to respond to the fault condition you'll want to reset `Register.FaultType` to zero after responding to it. `Register.FaultData` and `Register.FaultData2` are also persistent but it will usually not be necessary to reset its value.

3.12 Conditional Compilation Directives

The ZBasic compiler supports conditional compilation. This means that you can add conditional constructs to your code to specify that a portion of the code should be or should not be processed by the compiler. This is useful in order to create source code that can be compiled in different ways. Such flexibility can be used to create special versions of your application for different markets or for different customers, etc. It also provides a fast way to logically remove blocks of code from your program while leaving the source code intact so that it can be easily restored.

A key element of conditional compilation is the ability to define special identifiers and to give them values. These identifiers can then be used in conditional expressions that control whether or not a block of code will be processed normally by the compiler or ignored completely. The compiler supports the definition of conditional identifiers on the command line but you can also define them in your source code as well using the following syntax:

```
#define <identifier> [ [=] <expression> ]
```

Here, `<identifier>` is the name of the conditional identifier that you want to define. You may also give it a value, represented by `<expression>`, which may be an integral or string type. If you do not specify a value, a default value of 1 is used. The expression may include literals or identifiers previously defined using `#define` and some built-in constants (see Section 3.9).

Examples

```
#define EXPERIMENTAL
#define Version 23
#define Greeting = "Hello"
```

If you attempt to define a conditional identifier that is already defined, you will get an error message to that effect. If you want to redefine a conditional identifier you must first “undefine” the existing one using the directive:

```
#undef <identifier>
```

If the specified identifier is not actually defined, no error message will be issued so you may freely use this directive to ensure that no definition exists prior to defining a conditional identifier. Note that undefining an identifier that was defined on the command line only has effect in the current module. All other modules will see the original value.

Once you have defined your conditional identifiers, you may use them in conditional directives that are similar to If statements. The first two forms presented below are complementary.

```
#ifdef <identifier>
<other-text>
#endif
```

```
#ifndef <identifier>
<other-text>
#endif
```

The first form specifies that if the given `<identifier>` is defined, the compiler should process the text up to the matching `#endif` but if the `<identifier>` is not defined, the compiler should ignore the text up to the matching `#endif`. The second form has the opposite effect.

Example

```
#ifdef EXPERIMENTAL
    Call TestSetup(i)
#endif
```

This allows the subroutine call to be compiled into the application if `EXPERIMENTAL` is defined, otherwise it is left out.

As you might have already guessed, the conditional syntax also allows an `#else` clause.

```
#ifdef EXPERIMENTAL
    ver = "X006"
#else
    ver = "V1.4"
#endif
```

An alternate form of conditional directive allows you to specify an expression involving conditional identifiers and integer or string literals, the Boolean value of which determines whether the code within the conditional block is processed normally or not.

```
#if <expression>
<other-text>
#elseif <expression>
<other-text>
#else
<other-text>
#endif
```

The `<expression>` element may be any legal ZBasic expression involving constants, conditional identifiers and ZBasic operators. The usual type-compatibility rules apply, e.g., you cannot add an integral value and a string value.

The `#elseif` clause in the conditional construction may appear zero or more times. The `#else` clause may appear at most once. The `<other-text>` element represents arbitrary text and may contain other conditional constructs. There is no practical limit on the nesting of conditionals.

Examples

```
#if Version >= 23
    #ifdef EXPERIMENTAL

    ' prepare the external circuitry and activate it
    Call TestSetup(i)
    Call PutPin(12, j)
    #endif
#endif

#if (Version >= 23) And (EXPERIMENTAL <> 0)
    ' prepare the external circuitry and activate it
    Call TestSetup(i)
    Call PutPin(12, j)
#endif
```

The two examples above have the same effect.

Conditional identifiers may be used in definitions, statements, and expressions as if they were constants defined using the `Const` definition (but the converse is not true).

```
#define Version "V1.0"
#define arraySize 26

Dim myArray(1 To arraySize) as Byte

debug.print  Version
```

One implication of this is that adding a definition of a conditional identifier may result in a compiler message related to a `Const` definition warning about duplicate definitions. Conditional identifiers defined using a compiler option are visible in all modules while those defined in a particular module are only visible in that module. Also note that conditional identifiers are essentially module-level constants. This is true even if they are defined in a procedure. A consequence of this is that in spite of whether a conditional identifier is defined inside a procedure or not, its value is visible to all subsequent conditional expressions in the module.

BasicX Compatibility Note

Conditional constructs are not supported in BasicX compatibility mode.

3.13 Error Directive

Sometimes it is useful to be able to purposely generate an error message in order to remind yourself of some detail or condition that requires attention. Often, this is used in conjunction with conditional directives to point out that an incompatible set of conditions exists.

The form of the error directive is shown below.

```
#error <message>
```

All of the characters beginning with the first non-white space character following `#error` up to the end-of-line will appear as the error message. If the last character on the line is an underscore and is preceded by a space or tab character, it is treated as a continued line and all of the characters on the next line up to the end-of-line will also be part of the error message.

Example

```
#if PLATFORM = "alpha"
Const Frequency As Single = 42.347
#elseif PLATFORM = "beta"
Const Frequency As Single = 45.347
#else
#error No platform specified.
#endif
```

3.14 Notice Directive

This directive is similar to the #error directive discussed in the previous section in that it adds a string to the error output. However, its used does not increase the error count like #error does.

The form of the error directive is shown below.

```
#notice <message>
```

3.15 Include Directive

You may use an include directive in a source file to cause another source file to be compiled as well. The form of the include directive is:

```
#include "<filename>"
```

The `<filename>` element is the name of the file that you want to have compiled. If the filename is not specified using an absolute path (i.e. beginning with the root directory and/or a drive letter), the path prefix (if any) of the current module will be appended to the front of the filename. Note, however, that if an include path is specified on the command line, a filename specified with a relative path will, instead, be sought in among the directories specified in the include path list. See Section 7.2 for more information on the include path option.

Note that the effect of the include directive is no different than if you had instructed the compiler directly to compile the file. It is compiled as a separate module.

BasicX Compatibility Note

Include directives are not supported in BasicX compatibility mode.

3.16 Using Conditional Directives in Project and Argument Files

The ZBasic compiler optionally supports the use of the conditional directives, as described in Section 3.12, in project and argument files. Because conditionals are introduced with a # character and that same character is recognized as a comment introduction character in project and argument files, support for conditionals in these files is not enabled by default.

You may enable support for conditionals in project and argument files using the compiler option `--allow-conditionals`. If this option appears as the first line in a project or argument file, conditionals will be enabled for that file and for all subsequently processed project and argument files. If the `--allow-conditionals` option appears on a line other than the first line, conditionals will be enabled for subsequently processed files but not for the the file in which the option appeared. Alternately, conditionals may be enbled only for a specific project or argument file by placing the special comment `#!allow-conditionals` as the first line of the file.

When creating conditionals in project and argument files, you may utilize symbols previously defined via command line options as well as the built-in constant symbols described in Section 3.9.

3.17 Preprocessor Symbols

The ZBasic compiler supports several "preprocessor symbols" that can be used in your program code. These are special sequences of characters that the compiler recognizes very early in the compilation process. When they are recognized, they are replaced with a specific series of characters, different for each symbol. Conceptually, the replacement process is very much like a "global search and replace" operation in a text editor where context is not taken into account. The supported preprocessor symbols are shown in the table below.

Symbol	Description
`__DATE__`	The month, day and year when compiled, e.g. Oct 21, 2005.
`__TIME__`	The hour, minute and second when compiled, e.g. 14:16:04.
`__FILE__`	The name of the file being compiled.
`__LINE__`	The line number of the file on which the symbol occurs.

Although these are primarily intended as an aid to the testing procedure for the compiler, they may be useful in other circumstances. For example, you can cause the compilation date and time to be put in Program Memory using the following instructions:

```
Private Const compDate as String = "__DATE__"
Private Const compTime as String = "__TIME__"
Private compData as StringVectorData({ compDate, compTime })
```

During the preprocessing, the symbols above will be replaced by their respective character values. Of course, there will need to be some code that refers to `compData`, otherwise the compiler will optimize it out.

One caveat is that, as mentioned earlier, the substitution is done without regard to context. This will be a problem if you attempt to define a variable thusly:

```
Dim a__LINE__b as Integer
```

If this happens to occur on line 23 of the file the compiler will see this as

```
Dim a23b as Integer
```

However, on a different line a reference to a`__LINE__`b will yield a different variable name, probably causing the compiler to complain about use of an undefined variable. Since the preprocessor symbols are case sensitive but ZBasic identifiers are not, the simple workaround is to spell the variable name differently, e.g.

```
Dim a__line__b as Integer
```

3.18 Array Data Order

RAM-based arrays are stored sequentially in memory with the first index varying the fastest and the last index varying the slowest. Consider a two-dimensional array defined as:

```
Dim ba(1 To 3, 1 To 2) as Byte
```

The bytes of this array are assigned addresses in memory sequentially as:

```
ba(1,1) ba(2,1) ba(3,1) ba(1,2) ba(2,2) ba(3,2)
```

Whether this constitutes row-major order or column-major depends on whether you consider the first index of a two-dimensional array to be the column and the second index the row or vice versa. To a large extent, this is a non-issue as long as you remember that the first index varies the fastest.

When thinking about Program Memory data tables, on the other hand, the perception does matter because the initializer data is arranged in rows and columns and you need to know how to get the array element that you want. To do so, always specify the column index first and the row index second. This order was adopted to maintain compatibility with the BasicX compiler.

3.19 Recursion in Subroutines and Functions

A subroutine or function may be invoked recursively. This means that among the statements in the routine there is one or more that invokes the same subroutine or function again, either directly or indirectly. Clearly, the recursive invocation must be conditional so that at some point the recursion ceases.

In Section 2.3.2, Defining Functions, an example function was given for computing factorial. Here is the same function written using recursion.

```
Function Factorial(ByVal val as Integer) As Integer
   If (val > 1) Then
       Factorial = Factorial(val - 1) * val
   Else
       Factorial = 1
   End If
End Function
```

Although it may look confusing at first, the logic is fairly simple. The idea is based on the fact that the value of N factorial is equal to N times the factorial of (N-1). That logic is directly expressed in the second line of code. Note that the identifier `Factorial` is used in two distinctly different ways in the second line of the function. On the left side of the equal sign, the identifier `Factorial` refers to the return value variable while that on the right side of the equal sign is a recursive invocation of the `Factorial` function itself. The compiler is able to distinguish the two uses by the presence of parentheses following the name. Since the return value variable can never be an array, the two types of references are easily distinguishable.

The negative aspect of recursion is that it can consume a large amount of stack space. Each time a function or subroutine invocation is performed, the processor allocates additional stack space. The extra space is used to hold some tracking information to allow the processor to return to executing the code that immediately follows the invocation. Also, stack space is required for any parameters that are passed to the subroutine/function and for any variables that are defined inside the subroutine/function. Lastly, in the case of a function only, stack space is required to hold the return value from the function. Clearly, `Factorial(10000)` would require more stack space than is available even if all the RAM were dedicated to the stack.

BasicX Compatibility Note

The BasicX compiler does not allow direct recursive invocation of a function but the BasicX mode of the ZBasic compiler does.

3.20 Using Default Parameter Values in Subroutines and Functions

ZBasic supports the designation of default values for parameters to subroutines and functions. This saves time when typing statements and makes the code easier to read when a particular routine is usually invoked with one or more parameters having the same constant value. Specification of the default value is done by adding an equal sign and an constant expression following the type specification of the formal parameter in the routine definition as illustrated in the example below. In the first call to the subroutine `foo`, the second parameter is omitted so the compiler automatically adds the specified default value for the second parameter.

```
Sub Main()
  Call foo(3)
  Call foo(3, 5)
End Sub

Sub foo(ByVal size as Byte, ByVal cnt as Byte = 1)
End Sub
```

If any parameter has a default value specified, all parameters following that parameter must also have a default value. Also, only parameters that are passed ByVal may have a default value. Some parameter types, like arrays and structures for example, are always passed ByRef even if they are defined as ByVal and therefore cannot have a default value specification.

3.21 Aliases

Occasionally, it is useful to be able to access a variable or parts of a variable as different types at different times. Although this can be accomplished by using the System Library routines `BlockMove()` or `RamPeek()`/`RamPoke()` it is simpler and more efficient to use the concept of an alias. Simply stated, defining an alias tells the compiler to generate code to access a variable or part of a variable as if it were a different type. To be clear, no new data space is allocated by defining an alias. It simply provides a different way of accessing previously defined space.

The syntax for defining an alias is similar to that for defining a variable. For example, the syntax for defining an alias at the module level is shown below.

```
{Public | Private | Dim} <name>[(<dim-list>)] As <type> Alias <var-ref>
```

As with normal variables, `Dim` has exactly the same effect as `Private`. Within a subroutine, a function or any block structure, a local alias may be defined using the syntax shown below.

```
Dim <name>[(<dim-list>)] As <type> Alias <var-ref>
```

In both cases the *`<var-ref>`* element is the name of a RAM variable or the name of another alias optionally including a parenthesized set of one or more constant index expressions. The parenthesized index list is only allowed, of course, if the referent item is an array.

Examples

```
Dim ival as Integer, fval as Single
Dim buf(1 to 20) as Byte

Dim b As Byte Alias fval
Dim c As Byte Alias buf(2)
Dim c2(1 to 3) As Byte Alias buf(3)
Dim bval(1 to 5) As Byte Alias ival
```

The first alias definition allows you to read/write the least significant byte of the `Single` value `fval`. The second definition allows direct access to the second byte of the `buf` variable. The third example shows how to define a sub-array within an array. The fourth example shows an alias being defined that spans more than one variable. Although the compiler allows this form its use is discouraged because the effect depends on the order in which the compiler chooses to allocate data items.

Recursive alias definitions are not allowed; an error message will be issued by the compiler when a recursive definition is detected. Note that is not allowed to define an alias that is a `String` type. You may, however, define an alias that overlays a `String` variable although this is not often useful.

One interesting use for an alias is when your application requires that a series of data items be arranged in a particular order in memory. Consider a situation where, for whatever reason, it would be convenient to have an `Integer` value, a `Byte` value and a `Single` value that are guaranteed to be arranged in sequence in memory. This can be accomplished with the definitions shown below.

```
Dim host(1 to 7) As Byte
Dim ival As Integer Alias host(1)
Dim bval As Byte Alias host(3)
Dim fval As Single Alias host(4)
```

This technique may be useful, for example, for reading and writing data packets to an external device.

One aspect of using aliases that requires careful thought and possibly some experimentation is that an alias of a fundamental type (e.g. `Byte`, `Integer`, etc.) must be defined so that it aligns on a byte boundary. If the target variable for the alias is also a fundamental type this will not be an issue because the fundamental types are always byte-aligned. On the other hand, sub-byte types may or may not be byte-aligned so defining an alias to a sub-byte type may result in a compiler error message indicating that it is not byte-aligned. See Section 3.23 for more information on this topic.

BasicX Compatibility Note

Aliases are not available in BasicX compatibility mode.

3.22 Based Variables

Based variables are a very powerful tool and their use is recommended for advanced programmers only. If used carelessly or without a complete understanding of their characteristics they may cause your program to malfunction in ways that are quite difficult to diagnose.

A based variable is similar to a procedure parameter that is passed ByRef in the respect that no space is allocated for the data item. Rather, the location (i.e. the addess) of the based variable is specified using an integral expression that can be constant or computed at run-time. The effect that can be achieved using a based variable is similar to using an alias but a based variable is even more powerful because of the ability of the address to change at run time.

The syntax for defining a based variable at the module level is shown below.

```
{Public | Private | Dim} <name>[(<dim-list>)] As <type> Based <base-expr>
```

As with normal variables, `Dim` has exactly the same effect as `Private`. Within a subroutine, a function or any block structure, a local based variable may be defined using the syntax shown below.

```
Dim <name>[(<dim-list>)] As <type> Based <base-expr>
```

In both cases the `<base-expr>` element is an integral expression that gives the base address of the variable. Some examples will help clarify the concept.

```
Dim bv as Byte Based &H100
```

This defines a Byte variable whose address is a constant value.

```
Dim addr as Integer
Dim bv as Byte Based addr
```

This defines a Byte variable whose address is given by the value of the Integer variable `addr`.

```
Dim addr as Integer
Dim sel as Byte
Dim fv as Single Based addr + CInt(sel * 3)
```

This defines a Single variable whose address is given by the value of an expression.

```
Dim addr as Integer
Dim bv as Byte Based addr.DataAddress
```

This defines a Byte variable whose address is constant - the same as the address of the variable `addr`. Except for one important aspect, this has exactly the same effect as the following code.

```
Dim addr as Integer
Dim bv as Byte Alias addr
```

The difference between an alias and a based variable is how they are handled by the compiler's optimizer. Normally, when a variable's value is referenced, the compiler's optimizer attempts to deduce the variable's value at that point and, if it is more efficient to do so, the compiler will generate code using the deduced value instead of the variable's value in memory. However, if a variable is an Alias or has an Alias that refers to it, the compiler may not attempt to make this optimization.

In contrast, the compiler does not attempt to determine if a based variable might be occupying the same space as a normal variable. Consequently, if a variable's value is changed by assigning to a based variable that occupies the same space, the compiler may generate code that is incorrect. To circumvent this potential problem, you may instruct the compiler not to make any assumptions about the value of a variable by using the `Volatile` attribute as in the following example.

```
Dim Volatile addr as Integer
Dim bv as Byte Based addr.DataAddress
```

When defining an array variable that is Based, you may omit the `<dim-list>` element. In this case, the compiler will assume that the array is one-dimensional, that its lower bound is 1 and that its upper bound is a large value.

```
Dim addr as UnsignedInteger
Dim ba() as Byte Based addr
```

It is also permissible to define a based Persistent Memory or Program Memory variable. This is accomplished by using the normal syntax for defining a persistent/program memory variable and appending the `Based` keyword and address expression.

```
Dim persVar as Persistent Integer Based &H300

Dim progVar as ProgMem Single Based &H1000
```

BasicX Compatibility Note

Based variables are not available in BasicX compatibility mode.

3.23 Based Procedures

Like based variables, based procedures are a powerful tool intended to be used by advanced programmers who fully understand their nuances. No actual code space is consumed by a based procedure. Rather, declaring a based procedure simply tells the compiler how to generate an invocation of the procedure once its address is known.

The syntax for declaring a based subroutine or based function is shown below.

```
Delare Sub <name> ( [<parameter-list>] ) Based <addr-expr>
Declare Function <name> ( [<parameter-list>] ) As <type> Based <addr-expr>
```

As with based variables, the `<addr-expr>` giving the address of the procedure to be invoked must have an integral type and can be either constant or non-constant (i.e., computed at run time). The based procedure declaration may be placed inside a subroutine or function in which case the declaration is private to that routine. At the module level, the `Declare` keyword may be proceeded by `Public` or `Private` and if neither is specified, the declaration will be public by default.

When using a based procedure, you must be very careful to be certain that the declaration matches the actual procedure that exists at the address that is specified. If the address given is not the beginning of a procedure that is the same type and has the same number and type of parameters, the result is unpredictable but will likely cause your program to malfunction.

BasicX Compatibility Note

Based proceduresles are not available in BasicX compatibility mode.

3.24 Sub-byte Types

In addition to the fundamental data types described in Section 2.2, ZBasic also supports `Bit` and `Nibble` data types. These are referred to as sub-byte types because they occupy less than a whole byte – 1 bit and 4 bits, respectively. In certain circumstances, these types may help reduce the total RAM usage of an application. As compared to packing and unpacking bits in your source code, using these types will be more efficient in both execution time and code space.

`Bit` and `Nibble` types may be used in most places where one of the fundamental data types may be used. You can define arrays of them, you can define `Bit` and `Nibble` constants, you can pass them as parameters and you can define functions returning these types. One limitation is that you cannot pass a `Bit` or `Nibble` variable to a subroutine/function by reference unless the variable is byte-aligned (see further discussion below). This implies, of course, that you may not be able to pass an array of sub-byte types as a parameter since arrays are always passed by reference. The reason for this limitation is that sub-byte variables do not necessarily begin on a byte boundary and there is no way for the called routine to know what the alignment might be. One way to work around this limitation is to define an integral-byte alias to the sub-byte type, pass the alias by reference and then define a new sub-byte alias in the called routine. This works because integral-byte aliases are required to be byte aligned. Note that when passed by value as a parameter, a sub-byte type parameter occupies an entire byte on the stack.

Special type conversion functions, `CBit()` and `CNibble()` are provided to facilitate the use of these types in combination with other types. See the ZBasic System Library Reference Manual for more information on the conversion functions.

In Section 3.21 Aliases, it was mentioned that there may an issue with aliases of a fundamental type overlaying sub-byte types. The issue arises because fundamental data types must be byte-aligned and a particular `Bit` or `Nibble` variable may or may not be byte-aligned. In most cases it will be simpler to define a sub-byte alias to overlay variables of fundamental type. Doing the converse may require some experimentation to achieve the required byte-alignment. The alias defined below may or may not be accepted by the compiler depending on what other sub-byte types are defined at the same scope level and the order in which the are defined.

```
Dim ba(1 to 20) as Bit
Dim bval as Byte Alias ba(3)
```

Internally, the compiler collects together all of the sub-byte types at a given scope level (module, procedure, block) and allocates space for them collectively. The `Nibble` variables are allocated first in the order that they are defined followed by the `Bit` variables in the order that they are defined. The map file will show a special variable with a name like `@BitNib00`. This is the host variable that contains the individual `Bit` and `Nibble` values for a particular scope level.

BasicX Compatibility Note

`Bit` and `Nibble` types are not available in BasicX compatibility mode.

3.24.1 Forcing Byte Alignment

As described earlier, sub-byte type variables are normally sub-allocated within a host variable. One consequence of this space-saving strategy is that a sub-byte type variable is often not aligned on a byte boundary, a circumstance that imposes limitations on the possible uses of the variable. One solution to this problem is to instruct the compiler to align a particular variable on a byte boundary. This is done by adding the ByteAlign attribute to the variable definition as shown below.

```
Dim ByteAlign ba(1 to 20) as Bit
```

Each variable that is so defined will occupy an integral number of bytes of memory. For the example above, three bytes will be allocated even though only 20 of the 24 available bits will actually be used.

Note that the `ByteAlign` attribute may be used with any variable type but it has no effect for those types that are inherently byte-aligned. Also, the `ByteAlign` attribute may be used in a Structure to force sub-byte types to be byte-aligned.

3.25 Structures

It is often useful to define a data type that is a collection of data elements. For example, if you write a program to manipulate dates it may be useful to define a data type that contains “year”, “month” and “day” elements. This is convenient because it allows you to think about or refer to the group of data elements as a single entity rather than as the individual constituent elements.

In ZBasic, as in many other programming languages, such a collection of data items is referred to as a “structure”. A structure is a user-defined data type similar in some respects to an enumeration. As with an enumeration, you define a structure by specifying the consituent elements. The syntax for defining a structure at the module level is:

```
[Public | Private] Structure <name>
      <member-definition>
      ...
End Structure
```

If neither `Private` nor `Public` is specified, the structure definition is public. The ellipsis (…) in the syntax above connotes that there may be zero or more additional member definitions. A structure definition must have least one member and may have a practically unlimited number of members.

A structure may be defined within a subroutine or function, either at the outer level or within any inner block. In this case, the `Public` and `Private` keywords have no useful purpose and are therefore not allowed.

After a structure has been defined, the structure name may be used as a *<type>* in a variable or structure definition. A structure may also be used as the *<type>* in the formal parameter list of a subroutine or function definition. Note, however, that a Public subroutine or function cannot be defined with a parameter that is a Private Structure.

A *<member-definition>* has the same syntax as that used to define a variable. As with an ordinary variable, a member may be a single data element or it may be an array. The syntax for a member definition is given by the two descriptions below – the first being for a non-array member and the second being for a member that is an array.

```
{Public | Private | Dim} <name> As <type>

or

{Public | Private | Dim} <name>(<dim-spec-list>) As <type>
```

As with ordinary variables, `Dim` has exactly the same effect as `Private`, i.e., the member will only be directly accessible to code within the module. In contrast, a `Public` member may be accessed by code outside of the module in which the structure is defined. The names of the members of each structure defined may be any legal identifier however a particular name may be used only once in each structure. The use of a name as a member in one structure does not preclude it from also being used as a member name in a different structure or as a variable, constant, parameter, etc. This circumstance is a result of the ZBasic scoping rules - a structure definition is an independent name scope.

The *<type>* specified for a member may be any of the pre-defined types like `Integer`, `Byte`, `String`, etc. or it may be a user-defined type like an enumeration or another structure. The amount of space required for each member in a structure is the same as for a variable of the same type (but see the discussion below relating to sub-byte types and alignment). It is important to note that recursive structure definitions, with members that are or contain (directly or indirectly) the structure being defined, are not allowed.

Examples

```
Structure MyDate
      Public year as UnsignedInteger
      Public month as Byte
      Public day as Byte
End Structure

Structure MyTime
      Public hour as Byte
      Public minute as Byte
      Public seconds as Single
End Structure

Structure MyTimeStamp
      Public tdate as MyDate
      Public ttime as MyTime
      Private isCurrent as Boolean
End Structure

Sub Main()
      Dim ts as MyTimeStamp

      Call GetTimeStamp(ts.tdate.year, ts.tdate.month, ts.tdate.day, _
            ts.ttime.hour, ts.ttime.minute, ts.ttime.seconds)
End Sub
```

The example above illustrates how members of an instance of a structure are referenced. The variable `ts` is an instance of the `MyTimeStamp` structure. A member is referenced by appending the member name to the variable name, separating them with a period. For members that are structures, members of the subordinate structure are referred to similarly. No spaces are allowed in this construction. In cases where a variable or a member is an array, the index list directly follows the variable/member name, preceding the period.

```
Structure foo
      Dim b as Byte
      Dim ai(1 to 10) as Integer
      Dim ts(1 to 4) as MyTimeStamp
End Structure

Dim yr as UnsignedInteger
Dim i as Integer
Dim f as foo
Dim b as Byte

i = f.ai(b)
yr = f.ts(3).tdate.year
```

The address of a member of a variable that is a structure may be obtained by appending the `.DataAddress` property identifier to the reference or by using the `MemAddress()` function.

```
Dim addr as UnsignedInteger

addr = ts.tdate.DataAddress
addr = MemAddressU(ts.tdate)
```

Structures may be used in Alias and Based variable definitions. However, in these cases the structures may not contain members that are any of the string types. Structures may be passed to subroutines and functions either by reference or by value. If a structure is passed by value, the structure will be read-only within the procedure.

A variable that is a structure may be assigned to another variable that is the same type of structure using the standard assignment operator as long as the structure does not contain any members that have the allocated string type. In contrast, assignment of structures with members that are fixed-length or bounded strings is supported. Likewise, two variables that are the same type of structure may be compared for equality or inequality using the standard comparison operators, = and <>. The equality/inequality test is implemented using a byte-by-byte comparison of the content of two structures. If one or more members of the structure are the BoundedString type, the byte-by-byte comparision may result in a False value even though the strings are identical. This is because the currently-unused portion of the string store may contain byte values that are different between the two instances being compared. Similarly, comparison of structures containing allocated strings, while allowed, is not recommended because of the likelihood of resulting in false negatives.

Structures may contain members that are sub-byte types, `Bit` and `Nibble`. Members that are `Bit` type will be aligned on the next available bit boundary and those that are `Nibble` type will be aligned on the next available nibble boundary. If a member that is not a sub-byte type follows a sub-byte type member, that non-sub-byte member will be aligned on the next available byte boundary. Depending on how you define your structure, this may result in “holes” in the structure layout representing unused bits. The unused bits are generally of no consequence but it is important to note that these “holes” may interfere with the equality/inequality test for structures that are not initialized because the unused bits will have an indeterminate value. You can avoid this potential problem by performing a block initialization on locally defined structures.

Example

```
' this structure contains a "hole", unused bits following the ab member
Structure foo
      Dim b as Byte
      Dim ab(1 to 4) as Bit
      Dim b2 as Byte
End Structure

Sub Main()
      ' this structure is not automatically initialized
      Dim bar as foo

      ' this call zeroes out the entire structure
      Call MemSet(bar.DataAddress, SizeOf(bar), 0)

      ' other code follows
End Sub
```

Another method of creating a user-defined type, compatible with VB6, is also supported. The syntax for a `Type` definition is:

```
[Public | Private] Type <name>
      <member-definition>
      ...
End Type
```

In this case, the *<member-definition>* is the same as for defining a member of a `Structure` except that the visibility attribute (`Public`, `Private`, or `Dim`) is not allowed. The visibility of each member will be the same as the visibility of the `Type` itself.

BasicX Compatibility Note

Structure and Type definitions are not allowed in BasicX compatibility mode.

3.25.1 Structures in Persistent Memory and Program Memory

Structures in Persistent Memory and Program Memory may be defined using the syntax:

```
{Public | Private | Dim} <name> As Persistent <type>
```

or

```
{Public | Private | Dim} <name> As ProgMem <type>
```

where <*type*> is the name of a previously defined structure. The allowable members of a Persistent Memory or Program Memory structure are the intrinsic types `Byte`, `Integer`, `UnsignedInteger`, `Long`, `UnsignedLong`, `Single`, bounded strings, arrays of those types and other structures containing only those types. Arrays of structures in Persistent Memory or Program Memory may be defined by specifying the dimensions in the usual way.

It is permissible to directly assign between any combination RAM-based, Persistent Memory and Program Memory variables defined using the same structure. Similarly, direct comparison between like structures (equality and inequality only) is supported.

3.26 Data Type Implementation Information

This section provides more information on the technical details of the fundamental data types and variants. Although this information is generally not needed to write properly functioning programs it is provided for those who are interested and for the special cases where knowledge of such implementation details may help you implement your application.

To review, the table of fundamental data types introduced earlier in this manual is reproduced here with an additional column indicating the amount of space required for each type.

Fundamental Data Types

Data Type Name	Range of Values	Size in Bytes
`Boolean`	`True, False`	1
`Byte`	0 to 255	1
`Integer`	-32,768 to 32,767	2
`UnsignedInteger`	0 to 65,535	2
`Long`	-2,147,483,648 to 2,147,483,647	4
`UnsignedLong`	0 to 4,294,967,295	4
`Single`	approx. ±1.4e-45 to ±3.4e+38 and 0.0	4
`String`	0 to 255 characters	See 3.26.2

The `Boolean` type, while occupying an entire byte, will always contain one of two values under normal circumstances. The value `True` is represented by the value 255 and the value `False` by `0`. If your application has a need for a lot of Boolean variables it may be more efficient to use the Bit data type described in Section 3.22. The primary disadvantage to using the `Bit` type is that it cannot be passed by reference.

The `Single` type is implemented using the data format specified in the IEEE 754 standard for single precision floating point numbers.

3.26.1 User-defined Type Details

The user-defined types in ZBasic are enumerations and structures. Enumerations are implemented using a two-byte value to represent the enumeration member value. Structures are laid out in memory exactly corresponding to the order in which the members are defined in the structure definition. Bit and Nibble type members are aligned on bit and nibble boundaries, respectively. All other member types are aligned on a byte boundary. Due to these alignment rules and depending on the specific structure definition, there may be unused bits and/or nibbles within the structure. The number of bytes consumed by a structure variable is the sum of the sizes of the members including the unused bits.

3.26.2 String Data Type Details

For the string data types, the storage requirements and implementation details vary depending on compiler command line options, Option Directives and the type of string. For the `String` data type, if `Option AllocStr` is enabled (as it is by default for ZBasic modules) each string variable requires four bytes of storage in User RAM space plus additional space allocated from the dynamic memory allocation heap to hold the characters of the string. This string storage strategy is called dynamic string allocation because the space to hold the string's characters is dynamically allocated and will grow and shrink as need be to accommodate the string value assigned to it. One additional advantage to using dynamically allocated strings is that the string storage will never be overrun because it is changed dynamically and limited to the maximum size of 255 characters automatically.

The 4 bytes of User RAM for a dynamically allocated string variable are used in the following manner:

Dynamically Allocated String Storage Layout

Offset	Length	Description
0	1	The current string length, in bytes.
1	1	A marker identifying the string location: `&He0` – RAM (allocated from the heap) `&He2` – Program Memory `&He3` – Persistent Memory `&He4` – RAM (not allocated) `&He5` – RAM (not allocated, limited to 2 chars max.) `&He6` – RAM (pointer refers to a statically allocated string)
2	2	For type `&He5`, the one or two characters of the string. For type `&He6`, the address of a statically allocated string variable (see below). For all other types, the address of the first character of the string.

For statically allocated strings, including Bounded Strings and Fixed Length strings, the space allocated for the variable is used in the following manner:

Statically Allocated String Storage Layout

Offset	Length	Description
0	1	The current string length, in bytes.
1	1	A marker identifying the string characteristics: `&H00` – fixed allocation, variable length `&Hff` – fixed allocation, fixed length
2	N	The characters of the string where N is the defined string length.

For Fixed Length strings the byte at offset zero will be constant and the byte at offset 1 will have the value `&Hff` to indicate that it is a Fixed Length string.

Fixed-Length String Storage Layout

Offset	Length	Description
0	1	The string length, in bytes.
1	1	A marker identifying the string characteristics: `&Hff` – fixed allocation, fixed length
2	N	The characters of the string where N is the fixed string length.

3.26.3 String Address and String Type

Generally, you needn't be concerned about the technical details regarding strings described in the preceding section. For certain special situations, however, it may be useful directly access a string variable's inner data components. The `StrAddress()` function works with all of the string types described above and will return the address of the first character of the string storage. Note, however, that for dynamically allocated strings if the string's length is zero the returned address will be of no use (it will generally be zero). Moreover, the returned address may be an address in RAM, in Program Memory or in Persistent Memory. Depending on where the string's characters are stored you must use different System Library Functions to retrieve the string's characters. You can use the System Library function `StrType()` to determine the nature of the string's storage. It will return the second byte of the string variable's storage space. See the tables above for the values and their meaning.

BasicX Compatibility Note

In BasicX compatibility mode, neither `StrAddress()` nor `StrType()` is available.

3.27 Task Management

The ZBasic System Library has several routines that are helpful for managing tasks including LockTask(), UnlockTask(), StatusTask(), ResumeTask(), RunTask() and ExitTask(). The latter four routines permit some advanced task management for special situations. One use of these routines is to implement a timeout on a task that is awaiting the completion of an event, e.g. an external interrupt or an input capture. Normally, when a task is set to wait for an event like these it will wait indefinitely.

3.27.1 Task Control Block

Each task has an associated Task Control Block (TCB) - a data area that occupies the first few bytes of the task stack in VM mode (the last few bytes of the task stack in native mode). In most cases, a program will not need to access the contents of a TCB. The information is included here for those rare circumstances when is needed. Use of this information is recommended for advanced programmers only.

When a task is activated its task control block is initialized and then inserted into a circular linked list immediately following the task control block of the then-current task. The table below gives some information on the structure of the task control block. It is important to note that this information is considered implementation detail subject to change as necessary.

Task Control Block Elements (VM Mode Devices)

Offset	Length	Description
0	1	Task status. See `StatusTask()` for details.
1	2	Remaining time to sleep (in RTC ticks).
3	2	Address of next task control block.
5	6	Task context: IP, BP, SP (valid only when not the current task).
11	1	Task control flags (used internally).
12	2	Address of the byte following the end of the task's stack.

Task Control Block Elements (Native mode Devices)

Offset	Length	Description
0	1	Task status. See `StatusTask()` for details.
1	2	Remaining time to sleep (in RTC ticks).
3	2	Address of next task control block.
5	2	Task context: SP (valid only when not the current task).
7	2	Task stack starting address.
9	2	Unused.
11	1	Task control flags (used internally).

Caution: directly modifying the task control block (other than the "time to sleep" value) is strongly discouraged as doing so will probably cause your program to malfunction.

3.28 Dynamic Memory Allocation

The ZX system maintains a dynamic memory allocation heap that is primarily used internally. For example, space is allocated from the heap automatically by the system to hold the characters of String variables. That allocated space is automatically returned to the heap when String variables change or go out of scope.

In some applications, it is useful to be able to allocate a block of memory to use for some period of time (perhaps for a data buffer, for example) and then to deallocate the block when it is no longer needed. The ZBasic System Library routines System.Alloc() and System.Free() provide access to the dynamic memory allocation heap for such purposes. While these routines provide a very useful functionality, they must be used with great care because careless or improper use can result in malfunctioning of the heap. There are two fundamental problems that can arise: heap exhaustion and heap corruption.

If blocks of memory are allocated from the heap and never freed, the heap will eventually become exhausted and subsequent allocation requests will fail. Since String variable types rely on proper heap function, assigning values to String variables will not have the desired effect when the heap is exhausted.

Heap corruption occurs when the heap management data structures are inadvertently modified. Such corruption can result from writing to a previously allocated block after it has been freed or writing to memory outside the bounds of a properly allocated but not yet freed block. The heap may also be corrupted by inadvertent writes to RAM within the bounds of the entire heap such as might result from the careless use of the RamPoke() or BlockMove() routines, by writing beyond the bounds of an array, or overflowing the stack assigned to a task. Heap corruption will also be the likely result of passing an invalid value to System.Free(). The only values that may be safely passed to System.Free() are those that have been returned by System.Alloc() but have not yet been passed to SystemFree(). As a special case, passing the value zero to SystemFree() is benign.

The value Register.HeapEnd, described in this document, may be useful for monitoring the state of the heap. For additional information, see the descriptions of System.Alloc() and System.Free() in the ZBasic System Library Reference manual. Also, for native mode devices (e.g. the ZX-24n) the function System.HeapHeadRoom() may be used to determine, at any time, the amount of unused space that remains in the heap. Moreover, for native mode devices the directive `Option HeapSize` can be used to specify an upper limit on the size of the heap thereby preventing it from encroaching on the task stacks.

Chapter 4 – Special Considerations for Native Mode Devices

With the native mode devices like the ZX-24n, the ZBasic compiler produces native object code that is executed by the microcontroller. This is in contrast to the VM mode devices where the ZBasic compiler produces instructions that are executed by the ZX virtual machine running on the underlying microcontroller. Because the native mode code runs directly on the underlying microcontroller instead of the more controlled virtual machine environment, there are several additional features that you can employ in your application. Moreover, there are several additional issues of which you must be aware as you write the code for your application or when you port your code from a VM mode device to a native mode device.

4.1 Using Inline C and Assembly Code

When compiling ZBasic code for a native mode device, the compiler first produces equivalent C code corresponding to the subroutines, functions and data definitions of each module. Then, the resulting code is compiled into native object code for the underlying processor and linked with the ZX Runtime Library to produce an executable file that can be downloaded into the target device. In some cases it may be useful or desirable to inject sequences of C or native assembly language code directly into the code stream that is produced by the ZBasic compiler.

The mechanism provided for doing this is similar for the two cases – the code lines to be injected are bracketed by lines containing the special directives `#c` and `#endc` or `#asm` and `#endasm`. All of the text lines between the beginning marker and the end marker, but excluding those marker lines, are copied verbatim to the output file corresponding to the module being compiled. It is important to note that no syntax checking or other analysis is performed on the code that is copied so you must be sure that the code is syntactically and semantically correct.

Inline code sequences that occur outside of any subroutine or function are included in the module's output file after the module's data definitions but before any of the generated subroutine/function definitions. Inline code sequences that occur inside of a subroutine or function are placed in the generated code at the corresponding position for the routines to which they belong.

Consider this simple example using inline C code:

```
Dim b as Byte

#c
char ch;
#endc

Sub Main()
   b = 5
#c
   ch = zv_b;
#endc
End Sub
```

Note, particularly, that ZBasic variable names are prefixed by `zv_` in the generated C code. This is done to avoid namespace conflicts that might occur if the ZBasic variable names were used in the generated C code without a prefix. Other elements of the ZBasic program have distinguishing prefixes as well, as shown in the table below.

Accessing ZBasic program elements using the prefixed names in inline C code is fairly straightforward. However, you must keep in mind that the ZBasic compiler may eliminate some variables that you define in your program if they are not used or if the ZBasic compiler's optimizer determines that they aren't needed. If you want to ensure that the ZBasic compiler will not eliminated a variable you've defined you can define it with the Volatile attribute. That attribute is a signal to the ZBasic compiler to not make any assumptions about the value of the variable at any given point in time.

Program Element Prefixes

Element	Prefix
variable	zv_
structure	zs_
structure member	zm_
subroutine or function	zf_
parameter	zp_
function return value	zr_

Accessing ZBasic program elements in inline assembly code is possible but is more complicated partly due to issues regarding the register allocation, code generation and optimization stragegies employed by the compiler that translates the generated C code to native object code, discussion of which is beyond the scope of this document. That said, here is a simple example using inline assembly code.

```
Dim b as Byte
#c
char ch;
#endc

Sub Main()
   b = 5
#asm
   ; save the registers used
   push    r30
   push    r31
   push    r24

   ; load the address of the variable
   ldi     r30, lo8(zv_b)
   ldi     r31, hi8(zv_b)

   ; load the variable's value and save it
   ld      r24, Z
   sts     ch, r24

   ; restore the registers used
   pop     r24
   pop     r31
   pop     r30
#endasm
End Sub
```

It is important to note that inline assembly code must avoid altering certain registers that the compiler expects to remain unchanged. (The same is true, of course, of any assembly language code.) The code in the example above saves registers that it uses on the stack before using them and then restores their values afterward. There is a mechanism to inform the compiler as to which registers are altered in the inline assembly code thus allowing the compiler to automatically perform the save/restore if necessary. Discussion of that mechanism is an advanced topic that is beyond the scope of this document.

4.2 Defining and Using External Subroutines, Functions and Variables

For the native mode devices you may include .c files (C language), .S files (AVR assembly language), .o files (AVR object code) and .a files (AVR object code archives) in your project. When processing the component files of a project, the ZBasic compiler will note the presence of these special files but will otherwise ignore them except for including them in the final build phase. In order to invoke subroutines and functions contained in these special files, you must define them so the ZBasic compiler knows how to check for invocation syntax errors and how to generate code to invoke them.

The syntax for defining external routines is very similar to defining a ZBasic subroutine or function as shown below.

```
Declare Sub <name> ( [<parameter-list>] )
```

```
Declare Function <name> ( [<parameter-list>] ) as <type>
```

Parameters of the external routines may be given a default value as described in Section 3.20. In cases where the external routine name is a ZBasic keyword or is not a valid ZBasic identifier, you can use the `Alias` clause to specify the actual name of the external routine as a quoted string while using the specified *<name>* to refer to it in ZBasic code.

```
Declare Sub <name> ( [<parameter-list>] ) Alias "<ext-name>"
```

```
Declare Function <name> ( [<parameter-list>] ) as <type> Alias "<ext-name>"
```

Note that, with the exception of the String type, ZBasic data types are generally compatible with their similar counterparts in C and assembly language. To pass a string to an external routine, it is recommended that the string's characters be copied to a Byte array and then pass the array by reference to the external routine. Depending on the requirements of the external routine, the length of the string must also be passed or the string's characters must be null-terminated in the Byte array.

Example

```
Declare Sub extSub(ByVal ival as Integer)

Call extSub(5)
```

The `Declare` keyword may be preceded by `Public` or `Private`. If neither is present, the declaration defaults to being public meaning that it will be visible in other modules. The primary use for declaring a private external routine is when you define a routine using inline C or assembly code in the same module.

Similarly, external variables may be declared so that they may be accessed in ZBasic code.

```
Declare <name> As <type> [ Alias "<ext-name>" ]
```

Likewise, for an external array:

```
Declare <name>(<dim-spec-list>) As <type> [ Alias "<ext-name>" ]
```

Making these declarations does not result in any variable space being allocated. It is up to you to provide additional code to make the external variables available. As with subroutine and function declarations, variable declarations may be `Public` or `Private`. In the absence of either attribute, a data declaration defaults to being private.

For external variables that are to be treated as constant, you append the Constant attribute to the declaration. With this attribute, the compiler will generate an error if you attempt to modify the variable or to pass it to a subroutine or function by reference. Note, also, that the external variable should actually be defined with the `const` attribute if it is defined in C code.

Example

```
Declare index as Integer Attribute(Constant)
```

If the external variable declaration has both an Alias and and Attribute specified, they may occur in either order. For compatibility with the Declare statement in Visual Basic, the Alias specification for an external subroutine or function declaration may be placed between the routine name and the opening parenthesis of the parameter list. The form shown above, however, is the recommended syntax because it is more consistent with the other uses of the Alias keyword.

4.3 Defining Interrupt Service Routines

For the native mode devices you may write special-purpose code to service hardware interrupts. This may be useful, for example, to add some interrupt-driven capability to your program that is not directly supported by ZBasic. The syntax for defining an interrupt service routine is similar to that for defining a subroutine, illustrated here by example. Note that the definition of an ISR does not allow parameters but the parentheses are, nonetheless, required.

```
ISR Timer1_CompB()
  ' place the ISR code here
End ISR
```

The ISR name following the `ISR` keyword must be valid for the underlying processor, a list of which is shown in the table below. The ISR names are not case sensitive. More specific details about each interrupt may be found in the datasheet for the corresponding AVR processor. The shaded entries in the table below denote ISR names that may not be used because their functionality is essential to the operation of the ZX infrastructure. For other ISR names, you may define an ISR provided that it does not conflict with an ISR that is needed for System Library routines that you use in the application. For example, you may provide an ISR for the Timer1 Input Capture interrupt as long as you do not use the ZBasic System Library routine InputCapture() in your program. In the table below, the superscript, if any, indicates an ISR that may be automatically supplied by the compiler. The notes below the table describe, in general terms, when they might be automatically supplied. The descriptions of the System Library routines describe with more specificity which ISRs they require, if any, and the conditions under which they will be required.

Available ISR Names by CPU Type

mega644P	**mega1281**	**mega1280**
ADC	ADC	ADC
ANALOG_COMP[1]	ANALOG_COMP[1]	ANALOG_COMP[1]
EE_READY	EE_READY	EE_READY
INT0[1]	INT0[1]	INT0[1]
INT1[1]	INT1[1]	INT1[1]
INT2[1]	INT2[1]	INT2[1]
	INT3[1]	INT3[1]
	INT4[1]	INT4[1]
	INT5[1]	INT5[1]
	INT6[1]	INT6[1]
	INT7[1]	INT7[1]
PCINT0[1]	PCINT0[1]	PCINT0[1]
PCINT1[1]	PCINT1[1]	PCINT1[1]
PCINT2[1]	PCINT2[1]	PCINT2[1]
PCINT3[1]		
SPI_STC	SPI_STC	SPI_STC
SPM_READY	SPM_READY	SPM_READY
TIMER0_COMPA[2]	TIMER0_COMPA[3]	TIMER0_COMPA[3]
TIMER0_COMPB[4]	TIMER0_COMPB	TIMER0_COMPB
TIMER0_OVF	TIMER0_OVF	TIMER0_OVF
TIMER1_CAPT[5]	TIMER1_CAPT[5]	TIMER1_CAPT[5]
TIMER1_COMPA[6]	TIMER1_COMPA	TIMER1_COMPA
TIMER1_COMPB[7]	TIMER1_COMPB[7]	TIMER1_COMPB[7]
	TIMER1_COMPC[7]	TIMER1_COMPC[7]
TIMER1_OVF[5]	TIMER1_OVF[5]	TIMER1_OVF[5]
TIMER2_COMPA[3]	TIMER2_COMPA[2]	TIMER2_COMPA[2]
TIMER2_COMPB	TIMER2_COMPB[4]	TIMER2_COMPB[4]
TIMER2_OVF	TIMER2_OVF	TIMER2_OVF
	TIMER3_CAPT[5]	TIMER3_CAPT[5]

	TIMER3_COMPA	TIMER3_COMPA
	TIMER3_COMPB[7]	TIMER3_COMPB[7]
	TIMER3_COMPC	TIMER3_COMPC
	TIMER3_OVF[5]	TIMER3_OVF[5]
		TIMER4_CAPT[5]
	TIMER4_COMPA[6]	TIMER4_COMPA[6]
	TIMER4_COMPB	TIMER4_COMPB[7]
	TIMER4_COMPC	TIMER4_COMPC
	TIMER4_OVF	TIMER4_OVF[5]
		TIMER5_CAPT[5]
	TIMER5_COMPA	TIMER5_COMPA
	TIMER5_COMPB	TIMER5_COMPB[7]
	TIMER5_COMPC	TIMER5_COMPC
	TIMER5_OVF	TIMER5_OVF[5]
TWI	TWI	TWI
USART0_RX[8]	USART0_RX[9]	USART0_RX[8]
USART0_TX[8]	USART0_TX[9]	USART0_TX[8]
USART0_UDRE[8]	USART0_UDRE[9]	USART0_UDRE[8]
USART1_RX[9]	USART1_RX[8]	USART1_RX[9]
USART1_TX[9]	USART1_TX[8]	USART1_TX[9]
USART1_UDRE[9]	USART1_UDRE[8]	USART1_UDRE[9]
		USART2_RX[9]
		USART2_TX[9]
		USART2_UDRE[9]
		USART3_RX[9]
		USART3_TX[9]
		USART3_UDRE[9]
WDT	WDT	WDT

[1]Used by WaitForInterrupt().
[2]Used by the RTC, cannot be replaced.
[3]Used by OpenCom() for channels 3-6 (software UART).
[4]Used by OpenX10() along with INT0.
[5]Used by InputCapture().
[6]Used by ComToDAC() and DACToCom1().
[7]Used by OutputCapture().
[8]Used by Com1, cannot be replaced.
[9]Used by OpenCom() for channels 2, 7 and 8.

For the form of ISR definition shown in the example above, the compiler takes care of saving the necessary registers upon entry, establishing the standard register state, restoring registers upon exit and executing a “return from interrupt” instruction. For special cases, you may define a “naked” ISR – one in which none of this is done. Using this special form is advised only for advanced programmers, particularly those that understand the nuances of the underlying code.

An example of a naked ISR containing only a “return from interrupt” is shown below. This is commonly called a “stub” ISR. It is important to note that defining a naked ISR with no code statements in it at all results in undefined behavior. At a minimum, you should always include an assembly language “reti” as shown in the example below.

```
ISR Timer1_CompB() Attribute(Naked)
#asm
   reti
#endasm
End ISR
```

On occasion, it is desirable to have the same ISR service multiple interrupts. This can be accomplished by defining an ISR that is aliased to another ISR as shown below.

```
ISR Timer1_CompA()
  ' place the common ISR code here
End ISR

ISR Timer1_CompB() Alias Timer1_CompA
End ISR
```

You may also define a special ISR that serves all interrupts that otherwise have no ISR assigned. The default ISR is defined by using the interrupt name “default” as shown in the example below.

```
ISR default()
End ISR
```

If you do not define a default ISR, the compiler supplies one automatically. The function of the automatically-supplied default ISR is to store in Persistent memory a fault code that indicates that an unhandled interrupt has occurred and then execute a watchdog reset. The Persistent system variable Register.FaultType will have the value 2 to indicate that an unhandled interrupt occurred.

4.4 Executing Blocks of Code Atomically

Compared to writing code for the VM mode devices, when writing code for the native mode devices (e.g. ZX-24n) you must be more aware of the possibility that a sequence of statements may be interrupted either by an Interrupt Service Routine or by a task switch. Unless special techniques are used, another task or ISR may get control at any time in sequence of ZBasic statements – even in the middle of reading or writing the bytes of a multi-byte data item.

If a sequence of statements must be executed without interruption (commonly referred to as a “critical section” or “atomic execution”), the only way to guarantee such atomicity is to disable interrupts before the statement(s) and re-enable interrupts after the statement(s). Clearly, one must do this carefully so that interrupts are disabled for the least time possible in order to avoid missing a frequently occurring interrupt and to ensure timely servicing of important interrupts.

ZBasic supports several different techniques for disabling and re-enabling interrupts for one or more statements. Perhaps the simplest technique, and the one that is recommended for most purposes, is to use the atomic block construction. An atomic block is introduced by the keyword `Atomic` and is terminated by the keyword sequence `End Atomic` as illustrated in the example below.

```
Dim i as Integer

Sub Main()
   Atomic
       i = 200
   End Atomic
End Sub
```

Of course, if this were the entire application there would be no need to be concerned about atomicity of the assignment to the variable `i`. However, if the variable `i` is also read or written by an ISR or by another task, the access to the variable should be protected as shown above.

The second method for guaranteeing atomicity of a sequence of one or more statements is to use the System Library routines `DisableInt()` and `EnableInt()` in matched pairs as shown in the example below.

```
Dim i as Integer

Sub Main()
   Dim stat as Byte
   stat = DisableInt()
   i = 200
   Call EnableInt(stat)
End Sub
```

The disadvantage of using this method is that the compiler does not attempt to verify that the disabling and enabling are properly paired. Consequently, you might inadvertently omit the call to `EnableInt()` which would prevent any further interrupts, effectively disabling serial I/O, the RTC, etc.

A third method of guaranteeing atomicity is to explicitly disable and re-enable interrupts as depicted below.

```
Dim i as Integer

Sub Main()
   Register.SREG = &H00
   i = 200
   Register.SREG = &H80
End Sub
```

Although this technique may be useful in rare cases, its use is strongly discouraged. The primary problem with this method is that it unconditionally re-enables interrupts without regard to whether or not interrupts were enabled beforehand. A related technique using inline assembly language code, having the same disadvantage, is shown below.

```
Dim i as Integer

Sub Main()
#asm
   cli
#endasm
   i = 200
#asm
   sei
#endasm
End Sub
```

4.5 Attributes for Procedures and Variables

This section describes several special attributes that you can apply to variables or procedures with native mode devices to obtain special effects. To apply special attributes, you list the desired attributes (see the table below) in a comma-separated list enclosed in parentheses following the keyword `Attribute`. This entire construction is placed at the end of a normal variable, subroutine or function definition as illustrated in the example below.

Special Attributes

Attribute	Valid For	Description
`Inline`	subroutine function	Instructs the compiler to "inline" the code for the subroutine/function instead of generating a call. Generally, this yields faster execution, sometimes at the expense of larger program size.
`NoInline`	subroutine function	Instructs the compiler not to "inline" the code for the subroutine/function instead of generating a call.
`Used`	subroutine function variable	Instructs the compiler to include the variable or procedure in the executable even if it appears not to be used. This is useful, for example if external C or assembly code needs to use a ZBasic data item or procedure but it is not otherwise used in the program.
`Volatile`	variable	Instructs the compiler to not make any assumptions about the content of the variable. This should be used when another task or an ISR may be modifying the variable.

Example

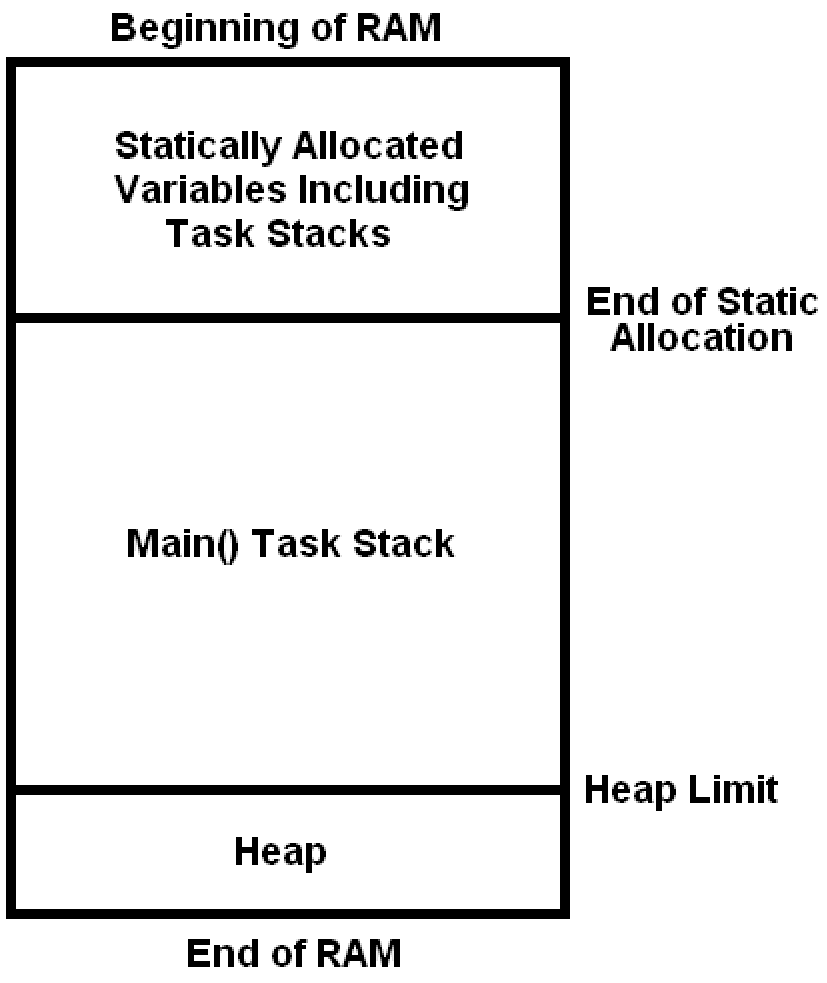
```
Dim index as Integer Attribute(Volatile)

Sub foo() Attribute(Used,Inline)
End Sub
```

4.6 Controlling Task Stack and Heap Sizes

For native mode devices, the compiler currently does not produce an estimate of the minimum size of the stacks for your applications tasks. Consequently, you must determine a suitable size for each task. Generally, the procedure for doing so is to pick an arbitrary size for each task, say 200 bytes, compile your application and then run it. If the chosen task stack sizes were sufficiently large, the application will run normally. If any of task stacks is too small, the application will exhibit erratic behavior, possibly restarting. Assuming that the application runs correctly, you can add code to determine the amount of unused space in each task stack by using the System Library function `System.TaskHeadRoom()`. Depending on the results of this observation, you may choose to increase or decrease the sizes of the various task stacks.

When using `System.TaskHeadRoom()` to determine task stack size, it is important to exercise your application thoroughly to ensure that all execution paths are taken. Moreover, each task's stack must have sufficient space to accommodate the stack requirements of any Interrupt Service Handlers that might be invoked while the task is running.

The diagram below illustrates how RAM is allocated between defined variables, task stacks and the heap.



The static allocation area, located at the beginning of RAM, includes all module-level variables defined by your application and all statically allocated system variables. This includes the task stacks that are statically allocated by your application. The task stack for Main() occupies all of the RAM between the end of the static allocation area and the heap. The stack pointer for all task stacks is initialized to the end of the task stack space and moves toward lower RAM addresses as it is used. By default, the Heap Limit is set at 512 bytes from the end of RAM. For devices with external RAM, the heap is located at the end of external RAM as determined at startup.

The heap will not grow beyond the Heap Limit. When the heap is exhausted, further allocation attempts will fail, possibly resulting in malfunctioning of your application. The heap is used for storing most RAM-based strings (but not BoundedString or fixed-string types) including strings that are returned by functions you define in your application and System Library functions that return strings. Additionally, on devices which have Program Memory in internal Flash memory, when a Program Memory write operation is performed a temporary buffer is allocated from the heap. The size of this buffer is equal to the Flash page size for the microcontroller, typically 256 bytes. This additional temporary use of heap space must be considered if your application performs write operations on Program Memory. Lastly, the System Library function `System.Alloc()` may be used by your application to allocate memory from the heap. All of these uses must be considered when determining how much heap space is needed by your application. The function `System.HeapHeadRoom()` may be helpful for determining the minimum heap size for your application.

If the default heap size is insufficient for your application, you can specify the Heap Limit setting in several different ways, each of which may be useful in different circumstances. Perhaps the simplest method of setting the Heap Limit is to specify the desired size for the Main() task stack. This can be done using the directive `Option MainTaskStackSize` in your Main module or by using the compiler option `––main-task-stack-size`. In both cases, the Heap Limit is set by adding the specified size to the address of the end of the static allocation area.

The second method of setting the Heap Limit is to specify the desired size of the heap using the directive `Option HeapSize` in your Main module or by using the compiler option `––heap-size`. The Heap Limit is set by subtracting the specified value from the address one past the end of RAM. For devices that have external RAM, the end of RAM is determined at startup. If you have external RAM, you may specify that all of the external RAM should be used for the heap by specifying the heap size using the special values 65535 or &HFFFF.

The final method for setting the Heap Limit is to specify an absolute address using the directive `Option HeapLimit` in your Main module or by using the compiler option `––heap-limit`. This method is recommended for use only by advanced programmers who understand completely how memory is allocated in a native ZBasic application.

Chapter 5 - Compatibility Issues

The ZBasic language and compiler were designed to be backward compatible with BasicX and to offer some of the advanced features of Visual Basic along with some capabilities not found in either VB or BasicX. A design goal was that any BasicX program that compiles error-free would be compiled without errors by the ZBasic compiler in BasicX compatibility mode. The converse may or may not be true. If you write a ZBasic program carefully avoiding those features that aren't supported in BasicX, you can expect that it will be compiled error-free by the BasicX compiler. Similarly, you can most likely write a Visual Basic program that can be successfully compiled by the ZBasic compiler if you carefully avoid those ZBasic features that aren't supported in Visual Basic as well as those VB features that aren't supported in ZBasic.

Another design goal was that, with a few unavoidable exceptions, a program that can be successfully compiled by both the BasicX compiler and ZBasic compiler (using BasicX compatibility mode) will exhibit substantially the same behavior when executed on the respective processors. The target behavior is as described in the BasicX documentation. However, in cases where the BasicX documentation is ambiguous, incorrect or incomplete, tests were developed to ascertain the actual behavior. Unless it was impossible, impractical or inadvisable to do so, the empirically determined behavior was implemented in order to maximize backward compatibility.

When porting a BasicX application to ZBasic it may be best to use BasicX compatibility mode at first. Once you get the program working in that mode you can then switch to not using compatibility mode in order to be able to utilize the advanced capabilities of ZBasic. Of course, turning off compatibility mode may cause your program to behave differently requiring some additional modifications to the program.

5.1 Known Differences and Compatibility Issues Between ZBasic and BasicX

- The ZBasic programs are not object code compatible with BasicX. Existing programs will need to be recompiled specifically for the desired ZX target.

- Some BasicX programs rely on I/O pin configuration performed by the Project | Chip dialog in the BasicX IDE. This configuration information is stored in a .prf file rather than appearing directly in the source code so it is easy to overlook. Porting such a program to ZBasic will require the addition of Option Pin or Option Port directives or the addition of equivalent configuration statements to the source code.

- Since the ZX microcontrollers run at twice the clock speed of the BX-24 your code will execute faster. This may reveal latent timing problems that existed in your BasicX application but that never before caused a problem.

- Due to the fact that the CPU runs twice as fast, the time units for `OutputCapture()` and `InputCapture()` are half as long. Depending on the particulars of your application it may work to simply apply the 0.5x scale factor to correct the values. Otherwise, you may be able to choose a different TimerSpeed setting to achieve the result that you need. Also, time divisors, including the serial port baud rate divisor must be twice as large to achieve the same effective rates.

- Also due to the faster clock speed, the SPI interface runs at twice the clock rate. If this is a problem, you may be able to select a different SPI clock multiplier to get satisfactory results.

- Some System Library routines are implemented differently on the ZX than they are on the BX-24. For example, on the ZX series the `ShiftIn()` and `ShiftOut()` System Library routines use Timer1 for speed regulation. This is not the case in BasicX. Also, the `X10Cmd()` routine will not return until the command has been transmitted. In BasicX, the documentation suggests that the routine returns immediately while the transmission is done in the background.

- In BasicX if the `CByte()` function is passed an `UnsignedInteger` argument that is larger than 255, the result is the low byte of the value. This is inconsistent with the handling of other types. In ZBasic, `CByte()` returns 255 when the parameter value is larger than 255 for all parameter types.

- The system-level details of ZBasic are likely to be different than those of BasicX. For example, the system information related to routine invocation, tasks, queues, etc. is probably not the same as that of BasicX so if your program relies on such information it is likely not to work correctly. Similarly, variable space may be allocated differently than it is for BasicX. If your code relies on certain variables being arranged in memory sequentially it may not work correctly. In general, any BasicX code that relies on internal implementation details needs to be examined carefully.

- By default, the ZBasic compiler omits superfluous code and does not allocate space for unused variables. There are compiler options to alter this behavior in case they are needed.

- The precedence of operators is different in ZBasic than it is in BasicX. The default operator precedence in ZBasic matches that of Visual Basic. However, in BasicX compatibility mode, the operator precedence matches that of BasicX. If you compile a BasicX program in ZBasic without using BasicX compatibility mode you may have to add parentheses to some expressions to achieve the intended effect.

- The `Mod` operator in ZBasic is sign-correct. In BasicX, the result of a `Mod` operation on a negative value is (incorrectly) a positive value.

- ZBasic supports `Mod`, multiplication and division operators on `UnsignedLong` types; BasicX does not.

- In ZBasic, enumeration types are represented internally using 16-bit values. In BasicX they are 8-bit values.

- In BasicX, all characters on an input line following a comment character are ignored. In ZBasic, an input line is examined to see if it ends with a continuation character before it is examined for comment characters. This means a line ending with a continuation character that is preceded by a comment character will not compile the same in ZBasic as in BasicX. The workaround is to delete the continuation character.

5.2 Known Differences and Compatibility Issues between ZBasic and Visual Basic

- VB implements run-time checks of various types including variable range overflow, array index bounds checking, stack overflow checking, etc. ZBasic checks string length only for allocated strings and fixed-length strings and has optional run-time stack overflow detection (for VM mode devices only).

- VB does not directly support `UnsignedInteger` and `UnsignedLong` data types. You can add such support to VB using classes but even then the characteristics of objects make them inherently different than using a native type.

- The VB behavior of “sign extending” hexadecimal values in the range &H8000 to &HFFFF (without the trailing ampersand) is not implemented (except in BasicX compatibility mode). This departure is implemented for compatibility with native unsigned types.

- The exponentiation operator in ZBasic is more limited in the types that can be handled as compared to VB. Also, in ZBasic the exponentiation operator is right associative compared to being left associative in VB. Right associativity for exponentiation is both more natural and more common in other languages.

- The only type designation characters recognized in ZBasic are the exclamation mark and pound sign. Both of these characters force type `Single` while in VB they force `Single` and `Double` respectively. ZBasic does not support type `Double`. Inasmuch as ZBasic may support Double types in the future, use of the pound sign type designator is strongly discouraged.

Chapter 6 - The ZBasic IDE

The ZBasic integrated development environment is based on the open-source IDE called SciTE. In addition to the basic text editing capabilities, the IDE has the ability to invoke the ZBasic compiler to compile your application and help you navigate to the lines containing errors. It also has the ability to download the code into the ZX and display any output that results.

6.1 Using the Editor

The source code editor built into the IDE has features designed especially for editing source code. Before those features are described, we will cover the basic editing capabilities that should be familiar to anyone with prior Windows application usage experience.

6.1.1 Basic Editing

As you would expect, typing regular characters (e.g. alphabetic, numeric, punctuation, etc.) causes them to be inserted in the current document at the caret position. If a selection exists at the time, the characters comprising the selection will be first deleted and the typed character will be inserted in its place.

The cursor keys (Up, Down, Left, and Right arrows) move the position of the caret. If the either Shift key is held down when a cursor key is pressed, a selection will be begun if none currently exists and the selection will be extended by characters (for Left and Right) or by lines (for Up and Down). If the Ctrl key is also held down, the Left and Right cursor keys will extend the selection by a word at a time.

While a selection exists, Ctrl+C will copy the selected characters to the clipboard while Ctrl+X will cut the selected characters to the clipboard. Ctrl+V will paste the current clipboard content into the document at the caret position replacing the currently selected characters, if any.

The Home key will move the caret to the first non-space character on the line or, if the caret is already at that position, to the first column of the document. The End key moves the caret to just after the last character on the line whether or not the last character is a white space character. Ctrl+Home moves the caret to the position before the first character of the document while Ctrl+End moves the caret just after the last character of the document.

The PageUp and PageDown keys move backward a forward through the document by pages.

The Backspace and Delete keys remove the character preceding and following the caret position, respectively. Holding down the Ctrl key while pressing Backspace and Delete serve to magnify their effect, deleting to the beginning and end of a word, respectively. Similarly, holding down both the Ctrl and Shift key magnifies the effect even more; deleting to the beginning of the line and the end of the line, respectively.

The editor has the ability to undo (Ctrl+Z) and redo (Ctrl+Y) recent changes. These and other editing commands are available via menu entry as well.

6.1.2 Special Code Editing Features

The specialized “lexers” built into the editor recognize certain aspects of the code in a document based on the “extension” portion of the document’s filename. When a document having a .bas extension is loaded, the editor will apply different styles to various portions of the content. This capability, sometimes called “syntax coloring”, helps programmers more quickly identify the syntactic elements of their program and to recognize when a typing error has been made.

For example, the editor will display Basic keywords (e.g. If, Else, etc.) in a bold blue font. If you type “Ekse” when you intended “Else”, you’ll immediately see that the editor did not display the mistyped word as a keyword thereby giving you immediate feedback that you erred.

6.1.3 Expand/Collapse

Another capability of the editor that is based on its ability to recognize the structure of the code is its ability to expand and collapse parts of the code. If you have a .bas file open that contains an If-Then-Else statement, you'll notice a small minus sign in the margin just to the left of the first displayed column on the line containing the If and the line containing the Else. If you left-click either one of these minus signs, you'll note that the display collapses to hide all of the lines in the corresponding section of code. At the same time, the minus symbol changes to a plus symbol indicating that there are "invisible" lines in the document. Left-clicking on the plus sign expands the corresponding code section. The expand/collapse effect can be invoked also by typing Ctrl+Keypad-* as well.

6.1.4 Auto-Completion

The editor has the ability to save you some typing by presenting a list of words having a prefix that matches a sequence of one or more characters that you have already typed. This ability, called auto-completion, comes in two distinct forms that differ in the origin of the list from which the matching list is built. The first form looks for matches in a list of words collected from the current document. This is useful for quickly inserting a variable, subroutine, or function name or language keyword that is already present elsewhere in your document. The second form looks for matches in a pre-defined list of ZBasic library routines and built-in constants.

The first form of auto-completion is invoked by typing the first few characters of an identifier that you know to be in the current document and then pressing Ctrl+Enter. If one or more identifiers exist having those characters as a prefix, they will be presented in a popup window near the caret. If multiple entries exist, the Up and Down cursor keys will move the selection up and down the list. At any time, you may press Enter to replace the prefix characters that you have already typed with the entry that you have selected. If you don't wish to use any of the items in the list, you may simply continue typing or you may press the Escape key. In either case, the popup window with the matching word list will disappear.

The second form of auto-completion is invoked by typing the first few characters of pre-defined library function or built-in constant and then pressing Ctrl+I. If one or more pre-defined entities exist having those characters as a prefix, they will be presented in a popup window near the caret. As before, if multiple entries exist, the Up and Down cursor keys will move the selection up and down the list. At any time, you may press Enter to replace the prefix characters that you have already typed with the entry that you have selected. If you don't wish to use any of the items in the list, you may simply continue typing or you may press the Escape key. In either case, the popup window with the matching word list will disappear.

In either case after the popup appears you may continue typing characters of the intended word and the list will be refined to include only those entries matching the modified prefix.

6.1.5 Call Tips

Another aid that helps to more quickly compose correct code is the Call Tip feature. When you type the name of a function or subroutine that is known to the system and then type a left parenthesis, a popup window will appear that will contain information about the number and type of parameters expected by that routine. In some cases, there may be multiple ways to invoke the routine, usually because some of the parameters are optional or a given parameter may be of two or more significantly different types (e.g. Single and Integer). In these cases, the Call Tip window will display a dark rectangle containing an upward pointing arrowhead of the left side of the popup. Clicking on the rectangle will display the preceding alternative while clicking to the right of the rectangle will display the next following alternative.

The content of the parameter lists, if any, is intended to remind you of the purpose and allowable types for the parameters. In most cases, the word `Integer` implies that the expected value is the ZBasic type `Integer`. Note, however, that the compiler is often somewhat lenient in its type checking for built-in routines and it may allow, for example, an `UnsignedInteger` to be used instead. The type descriptor

“AnyIntegral” means any integral type such as `Byte`, `Integer`, `Long`, etc. The descriptor “AnyNumeric” includes the same types as AnyIntegral but also includes the type `Single`. The descriptor “AnyType” means just that. Note that the parameter descriptions also include the `ByVal` and `ByRef` keyword. These tell you the parameter passing convention that is used for each parameter.

In some cases, a routine requires a parameter that is an array. In ZBasic code this is indicated by the presence of left and right parentheses following the parameter name. For technical reasons, this is indicated in the Call Tip using left and right square brackets in place of the parentheses.

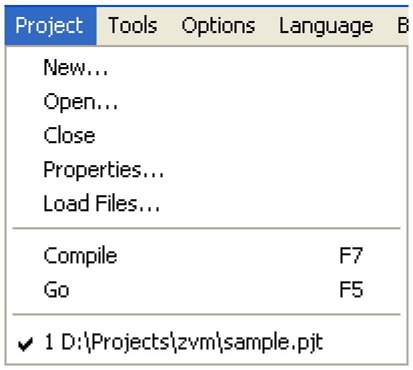
Call Tips may be added for your own routines. In addition to the file describing the system library routines (`zbasic.api`) the IDE will read a file named ZBasicUser.api located in the current user’s “home” directory, if such a file exists. On Windows systems, this is usually the same directory that contains your “My Documents” folder. You can add information for your own Call Tips to ZBasicUser.api following the examples in the `zbasic.api` file. The disadvantage to using this method is that the same Call Tip information will be used for all your ZBasic projects. You can add project-specific Call Tips by creating an API file having the same name as the project file but with an extension of “`.api`”. For example, for a ZBasic project named `C:\test\myproj.pjt` the project-specific API file is named `C:\test\myproj.api`. Although you can create and maintain this file manually, the compiler has the ability to generate the API file each time the compiler is run. See the description of the compiler option `--api` in Section 7.2 for more information.

One item that may need further discussion is the presence of a dollar sign ($) character in the API files. This optional element causes the displayed Call Tip to have a line break at that position (the dollar sign itself is not displayed in the Call Tip text). This will usually only be used when the Call Tip text is longer than a certain number of characters, 75 for example. You may edit the `zbasic.api` file as you wish to add, move or delete the line break characters.

It is important to note that the Call Tips that are presented for a given routine will be the aggregate of the matching routines found in `zbasic.api`, your `ZBasicUser.api` file and the project-specific API file.

6.2 Project Configuration, Compiling and Downloading

The Project menu, depicted below, contains several entries for managing projects including creating a new project, loading an existing project, modifying an existing project, compiling a project and downloading the compiled code for a project.



Most of the dialogs invoked by entries on the Project menu contain a Help button that can be used to obtain more information about using the dialog. When you create a new project using the “New…” menu entry you will have the opportunity to specify the filename for the project file and the folder where it will be created. The dialog also has an option for automatically creating the first module for the new project. If you elect to do this, the filename of the module will be the same as the project filename but with a .bas extension. The newly created file will have an empty `Main()` subroutine to which you may begin adding code.

To add files to the project that is currently loaded, select the “Properties…” entry on the Project menu. The resulting dialog has a button for adding files and one for deleting files. The latter button will be enabled only when one or more entries in the project file list is selected.

The “Load Files…” menu entry is for loading constituent project files into the editor. The resulting dialog presents a list of files that are currently part of the project and you may select one or more from the list to load into the editor. Of course, you may also load files into the editor at any time using the “Open” entry on the File menu.

The “Compile” entry on the project menu will invoke the ZBasic compiler instructing it to process the project file. Any error messages and warnings issued by the compiler will appear in the Output window and the editor cursor will be positioned automatically to the first error or warning. After addressing the problem you can move to the next error using the F4 key or by selecting the “Next Message” entry on the Tools menu. Once the project compiles without errors, you may download the resulting .zxb file by choosing the “Go” entry from the Project menu.

The list on the bottom of the Project menu will contain up to 10 recently used projects. The project currently loaded will have a checkmark beside it on that list.

If you wish to have your project compiled with certain compiler command line options you may manually add those options to the project file. For example, if you wanted to enable warnings about unused parameters you might add the following line to your project file:

```
--warn=unused-param
```

Generally speaking, you’ll want to add such lines at the top of the project file so that they will be in effect for all of the subsequently listed files. See Section 7.2 for more information on compiler options.

6.3 Compiling and Downloading Individual Files

In addition to being able to compile an entire project, the IDE can compile an individual file. If the currently selected document is a .bas file, you may compile that file by using the “Compile” entry on the Tools menu. This method is suitable for small programs that exist in a single file; it’s not necessary to create a project for such a simple application. If the compiler detects any errors, they will be displayed in the Output window that appears near the bottom of the main application window. Note that the first error or warning message will automatically be selected and the cursor will be positioned automatically to the offending line. After you rectify the error, you can quickly move to the next error or warning by pressing the F4 key or by selecting the “Next Message” entry on the Tools menu.

When the file compiles without errors, a download file will have been created in the same directory as the source file but having a .zxb extension. To download this file to the ZX, select the “Go” entry from the Tools menu.

It is important to note that the “Go” entry on the Tools menu will attempt to download a file having the same name as the current document but with a .zxb extension. You may also download a previously compiled file at any time using the “Download…” entry on the File menu.

6.4 Setting Serial Port Options

By default, the serial port used for communicating with the ZX is COM1. The serial port to use may be changed by using the “Serial Port Options…” entry on the Options menu. The combo box on the resulting dialog will contain entries for all of the recognized serial ports on your system

The serial port options dialog has other options that you may wish to modify as well including whether or not a verification pass is performed after the download.

All of the options configurable via the serial port options dialog are stored in the User Options File. You may load this file into the editor using the “Open User Options File” entry on the Options menu.

6.5 Setting Device Options

By default, the compiler generates code for the ZX-24 device. The target device may be changed either by adding a command line option to your project file or by adding a compiler directive to the first module to be compiled (usually the one containing the Main() routine). Alternatively, you may specify the target device by using the “Device Options…” entry on the Options menu. The resulting dialog contains a combobox with entries for the supported devices. This dialog can also be used to update the firmware in your device or, for some ZX models, to set configuration parameters like the Program Memory EEPROM characteristics or external RAM configuration.

6.6 Updating Device Firmware

The VM-mode ZX devices have firmware is designed to be field-upgradeable. This allows the control program to be updated with newer versions as enhancements are made and problems are fixed. An update can be installed in a VM-mode ZX using the “Device Options...” entry on the Options menu. This will bring up a dialog that will allow you to navigate to and select the specially formatted firmware update file (typically with .zvm extension). Prior to using the file’s contents to update the ZX firmware, the file is checked to ensure that it contains a valid update image. Note, particularly, that separate update files are provided for each ZX model. You must be certain that you use the correct update file for the ZX device being updated. The filenames of the update files indicate the target ZX device and the firmware version number. Additionally, the first few lines of the update file contain the same information. The update files are standard text files and may be viewed by any text file viewer.

To perform a normal firmware update the ZX must be powered up and must be connected to your PC by a serial cable. On the 24-pin ZX devices, during the update process the red LED will be illuminated continuously and the green LED will blink at a rate of about twice per second. Note that since the LEDs are also connected to pins 25 and 26, if you have circuitry connected to those pins that would be adversely affected by the LEDs being activated you will want to turn off the check box labeled “Provide visual feedback during update”.

On the other ZX devices no I/O pins are used to indicate that an update is in progress.

6.7 Setting the Debug Output Limit

By default, the debug window will only retain the last 100 lines of output. You can change this by selecting the “Open User Options File” entry from the Options menu. This will display the current contents, if any, of the User Options File. If one does not exist you may insert a line like the one below.

```
debug.line.limit=250
```

This will change the limit to 250 lines. The IDE will need to be exited and restarted after manually modifying the User Options File in this manner in order for the change to take effect.

6.8 Other Configurable Items

There are many aspects of the editor that may be configured. However, discussion of the means of making such changes and the impact of the changes is beyond the scope of this document. Those interested in investigating this topic further are directed to the “ZBasic Help” entry on the Help menu.

Chapter 7 - Compiler Guide

The ZBasic compiler is a modern design that employs some advanced optimization techniques to reduce code and data size while at the same time reducing execution time. The compiler operates in several stages including syntactic analysis, semantic analysis and code generation. During each of the first two phases error messages or warnings may be generated. Even though these are logically separate passes, the resulting error output lists errors by line number to facilitate easier resolution. Error messages may be generated during the code generation phases but this should generally not occur.

Note that the compiler may generate "spurious" error messages due to earlier detected error conditions. Consequently, fixing the cause of some earlier error messages may result in later messages being eliminated on the next compilation.

By default, error messages are routed to the stderr device meaning that they will appear on the console unless redirected to a file. Alternately, the compiler can be directed to output the error messages to a specifically named file. The error messages are formatted in a manner to allow an Integrated Development Environment or advanced text editor to automatically position the cursor at the next error message. The default error messages format is:

```
<filename>:<line-number>:<error-message>
```

where `<filename>` is the full name of the file containing the offending line, `<line-number>` is the line that contained the error (or, in the case of continued lines, the line number of the first continued segment) and `<error-message>` is a description of the detected condition. The `<error-message>` text includes an indication of whether the detected condition is an error condition or just a warning.

There is a class of error messages that reflect the occurrence of a set of conditions that weren't expected and shouldn't normally occur. If the compiler generates such "internal error" messages we would like to know the circumstances that produced them so that we can rectify the problem. There is probably not much that you can do to work around the problem in such cases but it may help to turn off all optimization (see Section 7.2 for details).

The ZBasic compiler is a "console program" meaning that it has no graphical user interface. The compiler is invoked via a command line that specifies the compiler options and the files to compile. One of the compiler options allows you to specify an "arguments file" that contains a list of options and/or files. Another option allows you to specify a "project file". This is similar to specifying an arguments file except that the use of a project file affects some filename defaults as described in subsequent sections.

7.1 Compiler Invocation

The general form of the compiler invocation is:

```
zbasic [<options>] [<files>]
```

where `<options>` represents one or more compiler options and `<files>` represents one or more files to be compiled. Both of these command line elements are optional; if neither is specified, a summary of the command line options is displayed. On the Windows platform, a file specification may contain "wild card" characters. For example, the file specification "`*.bas`" refers to all of the files having the .bas extension in the current directory.

Note that command line options and filenames may appear on the command line in any order and may appear multiple times. The options and files are processed left to right and when a filename is encountered the file is processed (syntactically analyzed) using the options in effect at that point in time. To understand how the intermixing of options and filenames affects the entire process, it is necessary to be aware that certain options are applied to files immediately as the files are encountered while other options are not applied until all files have been processed. If you utilize one of the options in the latter group multiple times, only the last instance will have an effect. In the tables below, the options that are immediately applied to files as they are processed are denoted with an asterisk preceding the description.

After all of the command line options and files have been processed, the compiler proceeds to the next phase – semantic analysis. If no errors are detected and unless directed otherwise, the compiler then proceeds to the final phase – code generation. It is during this last phase that the code output file, optional link map file and optional listing file are produced.

Note that some settings made by command line options may be overridden for individual modules by the presence of option directives in those modules. See Section 2.3.1 for more information on option directives.

All options are case-sensitive. Filenames may be case sensitive depending on the host operating system. Space characters are not allowed within options except within filenames (where supported by the host operating system) and possibly within strings. The command line language of the host operating system most likely requires option values containing spaces to be enclosed in quote marks.

ZBasic Compiler Options

Long Form	**Short Form**	**Description**
`--alloc-str=`*{*`on|off|default`*}*		*Specify the use of dynamic string allocation.
`--allow-conditionals`		*Allow the use of conditionals in project and arguments files.
`--api`*[*`(`*<width>*`)`*][=<file>]*		Request the generation of an API file for the IDE.
`--args=`*<file>*	`-fa`*<file>*	Specify an arguments file to be processed.
`--array-base=`*<value>*		Specify the default array base value.
`--called-by-list`		Request the generation of a “called-by” list for each routine.
`--calls-list`		Request the generation of a “calls” list for each routine.
`--code-limit=`*<value>*		Specify a limit for generated code size.
`--directory=`*<directory>*		*Specify the “current directory”.
`--entry=`*<subroutine>*		Specify an alternate entry point routine.
`--error=`*<file>*	`-fe`*<file>*	Specify an output file for error information.
`--error-format`*[=<fmt-spec>]*		Specify a format for error messages.
`--error-limit=`*<value>*		Specify a limit for error messages.
`--gcc-opts=`*<options>*		Specify options to pass on to gcc.
`--heap-limit=`*<value>*		Specify the heap growth limit.
`--heap-size=`*<value>*		Specify the size of the heap area.
`--help`	`-h`	Display a summary of the invocation syntax.
`--help-all`	`-hh`	Display a longer summary of the invocation syntax.
`--help-optimize`	`-ho`	Display a summary of the optimization options with an indication of the defaults.
`--help-warning`	`-hw`	Display a summary of the warning options with an indication of the defaults.
`--include-path=`*[<path-list>]*	`-I`*[<path-list>]*	*Specify a semicolon-separated list of directories to search for include files.
`--language=`*<language>*	`-l`*<language>*	*Specify the source language variant.
`--keep-files`		Request retention of intermediate files.
`--list=`*[<file>]*	`-fl`*[<file>]*	Request a detailed listing file.
`--main-task-stack-size=`*<value>*		Specify the size of the task stack for Main().
`--map=`*<file>*	`-fm`*<file>*	Specify an output file for the link map.
`--no-code`		Suppress the generation of code.
`--no-map`		Suppress the generation of a link map.
`--optimize=`*<opt-type>*	`-o`*<opt-type>*	*Assert or deassert optimization options.
`--out=`*<file>*	`-fo`*<file>*	Specify an output file for generated code.
`--project=`*<file>*	`-fp`*<file>*	Specify a project to be processed.
`--strict=`*{*`on|off|default`*}*		*Specify the compilation mode.
`--string-size=`*<value>*	`-s`*<value>*	*Specify the default string size.
`--target-device=`*<target>*		*Specify the target for code generation.

`--temp-dir=<`*`directory`*`>`		Specify the directory for intermediate files.
`--verbose`		Display output of the build process.
`--version`		Display the version number of the compiler.
`--warn=<`*`warn-type`*`>`	`-w<`*`warn-type`*`>`	*Enable or disable warnings by type.
	`-D<`*`id`*`>`*`[=<value>]`*	*Define a conditional symbol with an optional value.
	`-U<`*`id`*`>`	*Undefine a conditional symbol.

The options are described in detail in the next section, ordered alphabetically by the long form name followed by those options that have no long form.

7.2 Compiler Options in Detail

`--alloc-str=`*`{`***`On|Off|Default`***`}`*

This option specifies whether dynamically allocated string usage should be on, off or set to the default state for the selected language. See Section 2.3.1 for more information on the default state of this option for the supported language variants.

`--allow-conditionals`

This option activates support for using #if conditionals in project and argument files. Because of the fact that a pound sign introduces a comment in both of these types of files, support for conditionals in them is disabled by default. Alternately, you can enable support for conditionals in specific project or argument files by adding a special comment having the format shown below as the first line of the file.

```
#!allow-conditionals
```

`--api`*`[(<width>)][`***`=`***`<file>]`*

Using this option you can request that the compiler generate an API file for use by the IDE to display "call tips" for the routines in your program. If the =<*file*> is not present, the output is written to a file having the same base name as the first project file specified or, if none, the first module compiled, but with the extension "`.api`". For example, if the project is `myproj.pjt`, the default API file will be `myproj.api`.

The optional <*width*> parameter, which must be enclosed in parentheses, specifies a nominal line width for the generated API definitions. A special line break character is inserted near multiples of the specified width to allow the IDE to display long call tips on multiple lines. The width used if none is explicitly specified is 80.

`--args=`*`<file>`*

You can use this option to specify the name of a file that contains additional compiler options and/or filenames, one per line. The content of the file is processed before processing additional command line options. Such argument files may not be nested but a project file may be specified within an arguments file and vice versa. Argument files may contain blank lines and comment lines (beginning with a pound sign or an apostrophe).

`--array-base=`*`<value>`*

This option specifies base for arrays that do not explicitly specify the lower bound. The specified value must be either 0 or 1.

`--called-by-list`

When this option is specified a “called-by” list is appended to the map file. The called-by list will indicate, for each subroutine or function, each routine that invokes it. This information may be useful in analyzing the impact of changes in your code. If no map file is generated this option is ignored.

`--calls-list`

When this option is specified a “calls” list is appended to the map file. The calls list will indicate, for each subroutine or function, each routine that it invokes. This information may be useful in analyzing the impact of changes in your code. If no map file is generated this option is ignored.

`--code-limit=``<`*`value`*`>`

This option requests that the compiler compare the size of the generated code with the specified limit value. The limit value is specified in decimal, optionally using the suffix K or k to denote a multiple of 1024 bytes. If the code size exceeds the limit an error message will be generated. The default code limit is 0, which value disables the code size checking.

`--directory=``<`*`path`*`>`

This option specifies a directory that should be made the “current directory”. Any filename that has a relative path prefix will be sought relative to the current directory (but see `--include-path`, below). For example, since the filename `ir\test.bas` is a relative filename it will be expected to be in the current directory. In contrast, a file specified with an absolute filename like `c:\projects\ir\test.bas` will not.

`--entry=``<`*`subroutine`*`>`

This option specifies an alternate entry point for the program. By default, the entry point is the subroutine `Main()`.

`--error=``<`*`file`*`>`

This option explicitly specifies the name for the error output file. In the absence of this option, the error output is sent to stderr. Depending on your operating system, you may have the ability to redirect stderr to a file.

`--error-format``[=<`*`format-spec`*`>]`

This option explicitly specifies the format that should be used for outputting error messages. If the equal sign and format specification are missing, use of the default format specification will be resumed. The error format string specifies the string that precedes the descriptive error message. The format string may contain ordinary characters and escape sequences. The supported escape sequences are described in the table below.

Error Format Specification Escape Sequences

Escape	Description
`%f`	The file containing the error.
`%l`	The line number on which the error occurred.
`%%`	A literal percent sign (only needed to disambiguate).

The format specification may be enclosed in matching quote marks or apostrophes but this is necessary only if the format specification contains spaces. The default format specification is `"%f:%l:"`. This causes messages to have the format required by the IDE for error file navigation.

`--error-limit=`*`<value>`*

This option specifies an upper limit on the number of error messages that will be generated. When the limit is exceeded, compilation will cease. The default error limit is 100.

`--gcc-opts=`*`<options>`*

This option allows you to specify additional options to be passed to the gcc compiler and linker when generating code for native mode devices. The option string given will be added to the command lines after the options supplied by the ZBasic compiler and before the filenames. This position allows you to override earlier options or to add to the options or both.

`--heap-limit=`*`<value>`*

This option, useful only for native mode devices, specifies the heap limit, beyond which the heap will not grow. See the discussion in section 4.6 for details on the effect of this option.

`--heap-size=`*`<value>`*

This option, useful only for native mode devices, specifies the size of the heap and, indirectly, the heap limit. See the discussion in section 4.6 for details on the effect of this option.

`--help`

This option causes the compiler to output a summary of the invocation options and then exit.

`--help-all`

This option causes the compiler to output more comprehensive information about the invocation options and then exit.

`--help-optimize`

This option causes the compiler to output information about available optimization flags and default settings, and then exit.

`--help-warning`

This option causes the compiler to output information about available warning flags and default settings, and then exit.

`--include-path=`*`<path-list>`*

With this option you can specify a list of directories in which the compiler will look for files included using the `#include` directive, and also for Program Memory data initialization files. The *`<path-list>`* element consists of zero or more directory names, each separated by a semicolon. If any components of the directory name contains a space, the list may have to be quoted depending on your computer's operating system. Quoting is neither required, nor supported, when this option occurs in a project file or arguments file. Note that the current directory can be made part of the include path in the normal fashion by using a single period to represent it.

Example

```
--include-path=..\includes;.;C:\projects\zbasic\files
```

This specifies an include path with three components. When you use an include directive with a non-absolute path like `#include "lcd.bas"`, the first place that the compiler will look for `lcd.bas` is in the `includes` sub-directory of the parent of the current directory. If it is not found there, the compiler will next look in the current directory because of the presence of the period. Finally, the compiler will look in the directory `C:\projects\zbasic\files`. If the file could not be located in any of the directories of the path, an error message will be issued.

`--keep-files`

This option requests that the compiler not delete the intermediate files that it creates during compilation. This is only useful for native mode devices. Unless otherwise specified (using the `––temp-dir` option), the intermediate files will be created in a subdirectory named `zxTempDir` in the same directory as the project file.

`--language=``<`*`language`*`>`

This option specifies the target language for the modules subsequently processed. The values that may be specified for the `<`*`language`*`>` element are shown in the table below.

Language Option Values

Value	Description
`BasicX`	Compile using BasicX compatibility mode.
`ZBasic`	Compile using native mode (the default).

`--list`*`[=<file>]`*

This option requests that a listing file be generated and specifies the filename for it. The listing file is similar to an assembly language listing, giving detailed information about the code that was generated. If the equal sign and filename are omitted, the listing is output to stdout. Currently, no listing is generated for native mode devices.

`--main-task-stack-size=``<`*`value`*`>`

This option, useful only for native mode devices, specifies the size of the stack for the `Main()` task and, indirectly, the heap limit. See the discussion in section 4.6 for details on the effect of this option.

`--map=``<`*`file`*`>`

This option explicitly specifies the name for the map file. In the absence of this option, the map file name is derived from either the project file, if specified, or the first file compiled. If an earlier or later option specifies no map file should be generated this option is ignored. Currently, no map file is generated for native mode devices.

`--no-code`

This option causes the compiler to omit the code generation step. By implication, no map file will be generated either. This may be useful if all you want is a syntax check.

`--no-map`

This option causes the compiler to omit map file generation.

`--optimize=``<optimization-type>[,<optimization-type>...]`

This option enables or disables specific types of optimizations. In most cases, you'll want to use the default optimization settings. This option is provides for unusual circumstances where more control is needed over the optimizations performed. The optimization types are described in the table below. To disable an optimization type, add the prefix `no-` to the optimization type, e.g. `no-strength-reduction`. All optimization can be turned off using `--optimize=no-optimize`.

Optimization Type	Description
`constant-folding`	Expressions involving constants may be evaluated at compile-time. Note, particularly, that some System Library function invocations having parameters known to be constant may be replaced by the equivalent value. In some cases the same strategy may be applied to user-defined functions.
`constant-propagation`	The use of expressions involving variables known to be constant may be replaced with the constant value.
`expression-order`	Expressions may be rearranged to facilitate additional optimizations. Such rearrangement will never be performed across parenthetical boundaries if the option `preserve-parens` is specified.
`inline`	Small subroutines and functions may be generated in-line instead of generating a routine invocation.
`optimize`	Refers to all affirmative optimization types collectively. The only optimization type not included in this group is `preserve-parens`.
`preserve-parens`	The presence of this options restricts the use of expression optimizations to parenthetical boundaries.
`strength-reduction`	Reduction-in-strength optimizations may be performed, e.g. multiplication by a power of two replaced by left shift.
`string-pooling`	Code size is reduced by detecting identical strings. Each string appears just once in Program Memory but may be referred to in multiple places.
`unreachable-code`	Code that cannot possibly be executed may be eliminated.
`unreferenced-code`	Routines that are not used are not included in the generated code.
`unreferenced-vars`	Variables that are not referenced or are eliminated by optimization may not be allocated space.
`use-identities`	Expression complexity may be reduced by applying algebraic or logical identities. For example, `i * j` can be replaced by `i` if `j` is known to have the value 1.
`useless-code`	Code that is known to have no useful effect may be eliminated. For example, assigning a value to a local variable that is never used is considered useless.

The invocation option `--help-optimize` displays similar information and also indicates which optimizations are on by default.

`--out=``<file>`

This option explicitly specifies the name for the file for the generated code. In the absence of this option, the output file name is derived from either the project file, if specified, or the first file compiled. If an earlier or later option specifies that no code should be generated this option is ignored.

--project=*<file>*

This option specifies the name of a project file. A project file is similar to an arguments file in that it may contain additional compiler options and filenames that are processed before processing subsequent command line options. Additionally, however, several other file names (e.g. map file, output file) are derived from the first-specified project's filename in the absence of other overriding options. Project files may contain blank lines and comment lines (beginning with a pound sign or an apostrophe). Project files may not be nested but an arguments file may be specified within a project file and vice versa.

--strict=*{***On|Off|Default***}*

This option specifies whether strict syntax mode should be on, off or the default state for the selected language. See Section 2.3.1 for more information on the default state of this option for the supported language variants and the effects of strict mode.

--string-size=*<value>*

This option specifies the default string length, in decimal, for statically allocated strings in modules subsequently processed. It may be overridden in any module by the `Option StringSize` directive. See Section 2.3.1 for more information on how this value is used.

--target-CPU=*<target>*

This option is deprecated, use `––target-device` instead.

--target-device=*<target>*

This option specifies the target for which code should be generated. At present, the supported targets are:

```
ZX24      ZX24a      ZX24p      ZX24n
ZX40      ZX40a      ZX40p      ZX40n
ZX44      ZX44a      ZX40p      ZX40n
ZX1281    ZX1281n    ZX-1280    ZX1280n
ZX24e     ZX24ae     ZX128e     ZX1281e
```

If this option is used, it must appear before any modules are compiled. The default target is `ZX24`. The target device may alternately be specified in the first module compiled using the `Option TargetDevice` directive. Doing so will override the specification on the command line.

--temp-dir=*<directory>*

For native mode devices, the compiler generates several intermediate files in the process of compiling an application. Normally, those files are created in a temporary subdirectory of the directory containing the project file and then the directory and its content is deleted when the compilation is completed. The name of the temporary directory is selected to avoid conflicting with any existing files and directories. If the temporary files need to be examined, the `––keep-files` options can be used to prevent them from being deleted in which case the directory in which they will be created will be `zxTempDir`. A separate subirectory will be created in this temporary directory for each project compiled. If the default name of the temporary directory is unsuitable, you may specify a different directory using the `––temp-dir` option. If the specified directory is relative, it is interpreted as being relative to the directory containing the project file.

--verbose

This option, useful only for native mode devices, causes the output from the final build process to be sent to stderr. In the absence of this option, the output from the build process is captured in a file named `build.log` created in the temporary directory. (Use the `--keep-files` option to prevent the build log from being deleted after the build is complete.)

--version

This option causes the version number of the compiler to be sent to stdout. The compiler will then exit.

--warn=*<warning-type>[,<warning-type>...]*

This option enables or disables specific types of warnings. The warning types are described in the table below. To disable a warning type, add the prefix `no-` to the warning type, e.g. `no-unused-param`. All warnings may be disabled en masse using `--warn=no-warnings`.

Warning Type	Number	Description
`array-bounds`	9	Warn about constant indices on arrays being outside of the valid range.
`calltask-byref`	1	Warn about invoking a task that uses ByRef parameters.
`case-overlap`	10	Warn about the same value appearing in more than one case expression or range in a Select Case statement.
`data-range`	3	Warn about data values exceeding the capacity for the specified type.
`for-loop-termination`	11	Warn about a For loop that may not terminate properly.
`hidden-data`	5	Warn about data definitions hiding definitions at outer levels.
`never-returns`	8	Warn that a routine will never return (automatically suppressed for the entry routine and routines invoked using `CallTask`).
`questionable-code`	4	Warn about questionable coding practices.
`structure-compare`	12	Warn about comparison of structures containing allocated strings.
`task-stack-size`	13	Warn about insufficient task stack size.
`undefined-var`	2	Warn about the use of a variable before a value is assigned to it or when at least one path through a function does not set the return value.
`unused-param`	6	Warn about an unused parameter.
`useless-code`	7	Warn about code that will never be executed or has no effect.
`warnings`	1000	Refers to all warning types collectively.

The invocation option `--help-warning` displays similar information and also indicates which warning types are on by default.

-D*<id>[***=***<value>]*

This option defines an identifier that may be used in a conditional construct (see Section 3.12). If no value is specified, the identifier is considered to be an integral identifier and is assigned the value 1. The value may be specified as a decimal value or, when prefixed by `&H` or `0`, as a hexadecimal value. Otherwise, the value is considered to be a string value. The string value may optionally be enclosed in matching quote marks or apostrophes but this is only necessary if the value contains white space or begins with a character sequence that would otherwise indicate that an integral value is being specified. Note that depending on your operating system, using the ampersand or other special characters on the command line may require special quoting or escape characters.

If the identifier already exists, an error message will be displayed. The `–U` option may be used to avoid this situation. Also note that identifiers defined in this fashion are also available to be used in program code somewhat as if they were defined by a `Const` definition. The primary difference is that integral valued identifiers have a universal type that can be used just like a literal integral value.

Identifiers defined by this means are visible in all modules processed after the appearance of the option. Note that within a particular module the identifier may be undefined and re-defined. However, this has effect only within that particular module; the remaining modules will see the value as originally defined.

`-U``<`*`id`*`>`

This option causes the compiler to remove the specified identifier (as defined by `–D`) from the internal symbol table if the identifier exists. If the identifier doesn't exist, the option is silently ignored.

7.3 Error and Warning Messages

The compiler will output messages for detected error conditions and warnings to stderr (by default, the console). The message output may be redirected to a file using the I/O redirection capabilities of the operating system or by using a compiler option to specify the error output file (see **`--error`** above). Error messages relating to problems with compiler options will not be affected by the presence of the error output specification.

For code-related errors and warnings, the message will contain a reference to the filename and line number of the code corresponding to the error or warning. Although the line number given will generally be correct, code involving line continuations may lead to the line number reported being different than the actual physical line number where the offending code appears.

The general format of the error and warning messages may be modified to some extent using the **`--error-format`** option described in the preceding section. This may be useful if you are using the compiler with a different IDE that expects to see error message is a slightly different format for its "jump to the next error" function.

7.3.1 Controlling Warnings

The types of warnings that the compiler will issue may be controlled from the command line using the **`--warn`** option. Most of the warning types are enabled by default. It is desirable in some cases to allow the compiler to generate a certain type of warning message except for specific code sequences. This can be accomplished using the `#pragma warning` compiler directive in your program, the syntax of which is shown below.

```
#pragma warning( <warning id> : <disposition> )
```

The `<`*`warning id`*`>` element is either a numeric value (included as part of the warning message) or one of the mnemonic warning identifiers used with the `–warn` option. The `<`*`disposition`*`>` element must be one of the keywords `On`, `Off` or `Default`, the effect of which is to enable the warning, disable the warning or set it to the default state.

An example of the use of of the warning directive to disable a warning for useless code is shown below.

```
Dim b as Byte
Dim i as Integer

Sub Main()
   b = 3
#pragma warning(7 : Off)
   If (b > 5) Then
       i = 100
   ElseIf (b <= 5) Then
       i = 10
   End If
#pragma warning(7 : On)
   b = 25
```

```
End Sub
```

Because of the immediately preceding assignment, the compiler can deduce the Boolean value of the conditions in both the If and ElseIf statements and will issue a “useless-code” warning indicating that that those conditions are always false. The first warning directive disables that warning and the second one unconditionally enables it. Often, it is undesirable to unconditionally enable a warning after disabling it as was done in this example. To avoid enabling a warning that was not previously enabled, two variations of the warning directive are provided to save the current set of enabled warnings (push) and restore them afterward (pop) instead of unconditionally re-enabling a warning. The syntax for the push and pop warning directives is illustrated in the modified example below.

```
Dim b as Byte
Dim i as Integer

Sub Main()
   b = 3
#pragma warning(push)
#pragma warning(7 : Off)
   If (b > 5) Then
       i = 100
   ElseIf (b <= 5) Then
       i = 10
   End If
#pragma warning(pop)
   b = 25
End Sub
```

Note that it is permissible to combine two or more warning directives by separating the warning specifications using a semicolon. For example:

```
#pragma warning(push; 7:Off; unused-param:On)
```

7.3.2 Internal Errors

The compiler may, in unusual circumstances, generate an Internal Error message. This will happen when it encounters a situation where inconsistencies in internal data structures arise that prevent continued operation. If you encounter an internal error or a program fault, please report it so that it can be investigated to see why the problem occurred. If possible, construct the smallest program possible that still exhibits the problem and submit that program along with the error report. In some cases, you may be able to work around the problem by turning off all optimization although this is not guaranteed to be helpful. Still, it may be worth a try to get you past the obstacle while the problem is being addressed.

Problem reports may be emailed to support@zbasic.net or they may be reported via the Support Forum at http://www.zbasic.net/forum.

Chapter 8 - Downloader Utility

Included as part of the ZBasic package is a standalone command line utility that can download code to the ZX. Like the compiler, it is a console application meaning that it has no graphical user interface. The invocation syntax for the downloader is:

```
zload [<options>] <code-file>
```

where *`<code-file>`* is a file containing object code created by the compiler, usually having a .zxb extension. The code file is in an industry standard format and contains checksums to help detect communication problems.

After the code file is downloaded, the ZX is reset so that it begins running the user program.

The available options for the downloader are described in the table below. All options are case sensitive.

zload Options

Option	Description
`-h`	Display an invocation syntax summary and then exit.
`-c`*`<port>`*	Specify the serial port to use instead of the default port. *`<port>`* must be a decimal value between 1 and 99.
`-v`	After downloading, perform a verification pass.
`-m`	After downloading, remain connected and display any received characters on the console.
`-s`*`[<term-char>]`*	Terminate the monitor mode (`-m`) if a character with the value specified by *`<term-char>`* is received. The value may be specified in decimal or, with a prefix of `0x`, in hexadecimal. If not specified, the termination character is EOT (04).
`-a`	Enter ATN test mode. Used to verify correct connection to the ZX.
`-u`	Download a specially formatted file to update the control program of the ZX. See Section 8.1 for more details.
`-U`	Same as `–u` except that no visual feedback is provided (using the LEDs on the 24-pin ZX devices) during the update.
`-e`	Use a special “emergency update” method. See Section 8.1 for more details.
`-z<id>:<value>`	This option is used to send configuration information to the device.

Some example invocations of zload are:

Download the file `test.zxb` using COM2.
```
zload –c2 test.zxb
```

Download the file `test.zxb` using COM2 with verification.
```
zload –c2 –v test.zxb
```

Download the file `test.zxb` using the default COM port, go to monitor mode until character code 8 is received.
```
zload –s0x08 –m test.zxb
```

Perform ATN testing using COM2.
```
zload –c2 –a
```

The default serial port may be specified by setting an environment variable named `ZLOAD_PORT`. The value of this environment variable should be a series of decimal digits specifying the serial port number (1-99). The method used to set an environment variable varies depending on the operating system. Consult your OS documentation for more information.

Example

```
ZLOAD_PORT=2
```

If no default serial port is specified and the –c option is not specified, COM1 is used by default. Source code is provided in the distribution for the zload utility. It is a fairly simple program written in C targeted to the Windows platform. The utility may be able to be ported to another operating system by a skilled programmer who is familiar with programming for both Windows and the target platform.

8.1 Firmware Updates

The firmware of VM-mode ZX devices (as opposed to native mode ZX devices) is designed to be field-upgradeable. This allows the control program to be updated with newer versions as enhancements are made and problems are fixed. An update can be installed in the ZX using the `–u` option of the zload command line utility and specifying the name of a file to install. The file must contain specially formatted data, the integrity of which is verified before downloading to the ZX. It is important to note that the separate update files are provided for the various models. Be sure that you have the correct update file for your ZX device.

To perform a normal firmware update the ZX must be powered up and must be connected to your PC by a serial cable. An example invocation of zload to perform a normal firmware update is shown below. The example shows the use of the `-c` option to also specify the serial port to use. The file `zx24_1-10-2.zvm` is an example of the specially formatted update file for the ZX-24.

```
zload –c2 –u zx24_1-10-2.zvm
```

On the 24-pin ZX devices, during the update process the red LED will be illuminated continuously and the green LED will blink at a rate of about twice per second. Note that since the LEDs are also connected to pins 25 and 26, if you have circuitry connected to these pins that would be adversely affected by the LEDs being activated you can suppress the activation of the LEDs by using the `–U` option to instead of using `–u`.

It is important to ensure that the update process, once begun, is allowed to run to completion. Powering down the ZX or resetting it during the update may leave it in an unusable state. It is possible that the ZX may no longer be able to properly interact with the zload program to effect a subsequent complete update. In such a case, you may be able to use the special emergency update procedure described below.

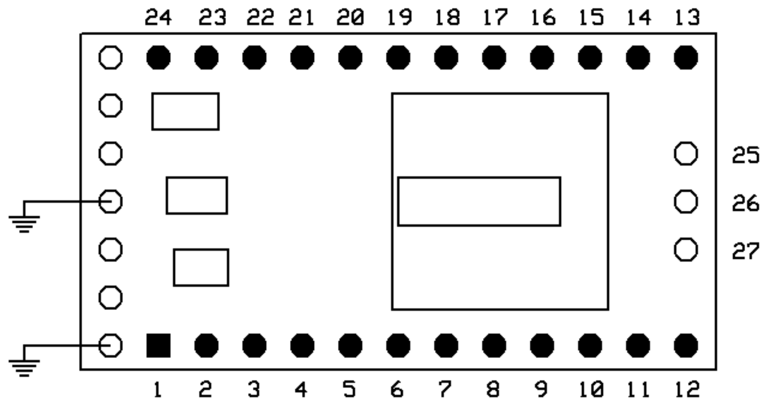
8.1.1 Emergency Update Procedure

To prepare a 24-pin ZX device for an emergency update you must ground the center terminal and the bottom terminal on the left end of the ZX device as shown in the diagram below. This can usually be done with a temporary connection like a test wire with alligator clips or similar connectors. When the ZX is powered up with these connections it enters the emergency update mode instead of operating normally.

Finally, the zload command must be invoked in emergency update mode, an example of which is shown below.

```
zload –c2 –e zvm_1-12.zvm
```

When the emergency update completes successfully the ZX may be powered off and the grounding jumpers may be removed. When it is powered up again it will begin running the newly installed control program normally. Note that the emergency mode may be used any time but because of the extra connections that are required it is primarily intended for extraordinary circumstances when the normal update mode cannot be employed.



24-pin ZX Device Connections for Emergency Update Mode

The emergency update procedure for a ZX-40, ZX-40a, ZX-44, ZX-44a or ZX-1281 is similar to that described above. To prepare for the emergency firmware update, consult the table below for the specific device and ground the indicated pin or pins.

Device	Pins to Ground
`ZX-40, ZX-40a, ZX-40p`	`5, 7 (B.4, B.6)`
`ZX-44, ZX-44a, ZX-44p`	`2, 44 (B.4, B.6)`
`ZX-1281`	`25 (D.0)`
`ZX-1280`	`43 (D.0)`

When the ZX powers up and detects that these pins are grounded, it will enter the emergency update mode and await an emergency firmware download on the serial port. After the firmware update is downloaded using the zload command with the –e option the device should be powered off and the grounding jumpers should be removed. When it is powered up again it will begin running the newly installed control program normally.

8.2 Device Configuration

The device configuration option, -z, can be used to set configuration parameters of the device. Currently, the only parameter that can be configured is the external EEPROM type. This only useful for devices where you can choose which EEPROM to use, e.g. the ZX-40 or ZX-44.

`-z0:`*`<EEPROM type>`*

The *`<EEPROM type>`* element is a 16-bit composite value that specifies characteristics of the attached EEPROM. The least significant byte specifies the “page size” of the EEPROM in terms of 16-byte blocks. For example, a page size of 64 bytes would be specified using the value 4 while a page size of 128 bytes would be specified using the value 8. The least significant bit of the most significant byte specifies whether the EEPROM requires full-page writes. If the EEPROM supports writing less than a full page, this bit should be off.

The *`<EEPROM type>`* element can be specified as either a decimal value or as a hexadecimal value. In the latter case, a prefix of 0x or 0X must be present to indicate that hexadecimal form is being used. The example configuration specifications below all use the hexadecimal form. Note that the values specified in the examples are shown with 4 hexadecimal digits for clarity but leading zero digits may be omitted.

```
-z0:0x0004
```

This specifies an EEPROM with a 64-byte page that supports partial page writes. This is the configuration to use for the Atmel AT25256A. This is the default configuration, ZX devices come pre-configured for this EEPROM.

```
-z0:0x0104
```

This specifies an EEPROM with a 64-byte page that requires full page writes. This is the configuration value to use for the Atmel AT25HP256.

```
-z0:0x0008
```

This specifies an EEPROM with a 128-byte page that supports partial page writes. This is the configuration value to use for the ST Microelectronics M95512.

```
-z0:0x0108
```

This specifies an EEPROM with a 128-byte page that requires full page writes. This is the configuration value to use for the Atmel AT25HP512.

It is important to note that whenever the ZX firmware is updated the default configuration is restored. Therefore, if your chosen EEPROM requires other than the default configuration value you must reconfigure the device after each firmware update.

8.3 Downloader API

In some cases, it may be useful to invoke the downloading functions from another program. For this purpose, a downloader API is provided in the form of a dynamic link library (DLL). The source code for the DLL and the DLL itself may be found in the `src\zload` subdirectory of the directory where ZBasic is installed. Two sample applications, one in C and one in Visual Basic, using the downloader DLL are also available in the same directory sub-tree.

Appendix A - Reserved Words

The list below enumerates the words that are reserved for use as keywords in ZBasic or reserved for possible future use. They may not be used as identifiers in user-written programs.

alias	enum	persistentstructure
and	eqv	persistenttype
array	erase	preserve
as	exit	print
attribute	for	private
based	function	progmem
bit	get	public
boolean	gosub	put
boundedstring	goto	redim
byref	if	register
byte	imp	rem
bytealign	imports	resume
bytetabledata	in	return
bytetabledatarw	input	rset
bytevectordata	int	seek
bytevectordatarw	integer	select
byval	integertabledata	set
call	integertabledatarw	sgn
calltask	integervectordata	single
case	integervectordatarw	singletabledata
ccur	is	singletabledatarw
cdate	lbound	singlevectordata
cdec	let	singlevectordatarw
close	like	sizeof
const	lock	spc
currency	long	static
cvar	longtabledata	step
cverr	longtabledatarw	stop
date	longvectordata	string
declare	longvectordatarw	stringtabledata
defbool	loop	stringvectordata
defbyte	lset	sub
defcur	me	tab
defdate	mod	then
defdbl	module	to
defdec	new	type
defint	next	ubound
deflng	nibble	unlock
defobj	not	unsignedinteger
defsng	open	unsignedlong
defstr	option	until
defvar	or	variant
dim	persistent	vb_name
do	persistentboolean	volatile
doevents	persistentbyte	wend
double	persistentinteger	while
each	persistentlong	with
else	persistentsingle	write
elseif	persistentsingle	xor
end	persistentstring	

Appendix B - ZX-24 Series Hardware Reference

The heart of the ZX-24 is the Atmel AVR ATmega32 microcontroller while the ZX-24a is based on the ATmega644 microcontroller. The ZX-24p and ZX-24n devices are based on the ATmega644P microcontroller. In all cases, the microcontroller chip is accompanied by some support circuitry on a 24-pin module that is simple to connect for operation.

The primary difference between the ZX-24 and the ZX-24a is the amount of RAM available for use by your programs. For the ZX-24 the User RAM comprises 1536 bytes (1.5K bytes) while it is 3.5K bytes for the ZX-24a and ZX-24p. The remainder of the microcontroller's RAM is used by the ZX control program.

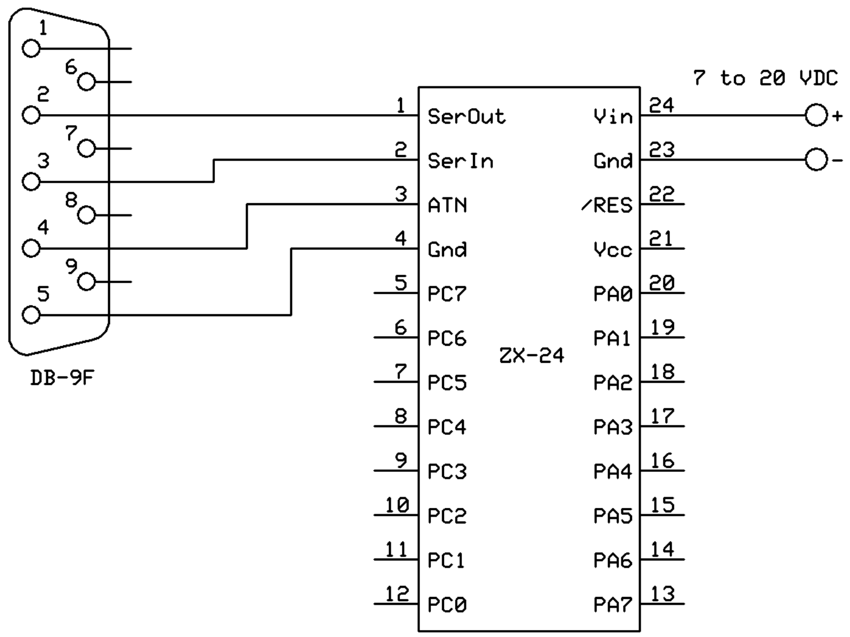
The ZX-24 also provides 992 bytes of internal EEPROM (electrically erasable programmable read-only memory), referred to as Persistent Memory. The ZX-24a and ZX-24p provide 2016 bytes of Persistent Memory available to your program. Data written to Persistent Memory is retained when the ZX is powered down or reset.

The 24-pin ZX models provide additional Flash memory external to the microcontroller chip in which, except for the ZX-24n, your program code is stored. The ZX-24n has the external Flash memory but it is not used because your program's code is stored in internal Flash memory. For all 24-pin ZX models, the external Flash memory is 32K bytes in size.

The ZX-24 and ZX-24a have other resources available to your program including a high-speed serial port, analog-to-digital converters, timers and other sub-systems that may be accessed using routines in the ZBasic System Library. Alternately, some of the resources may be accessed directly using built-in registers. See Section 3.8.1 for more information on this topic.

B.1 External Connections

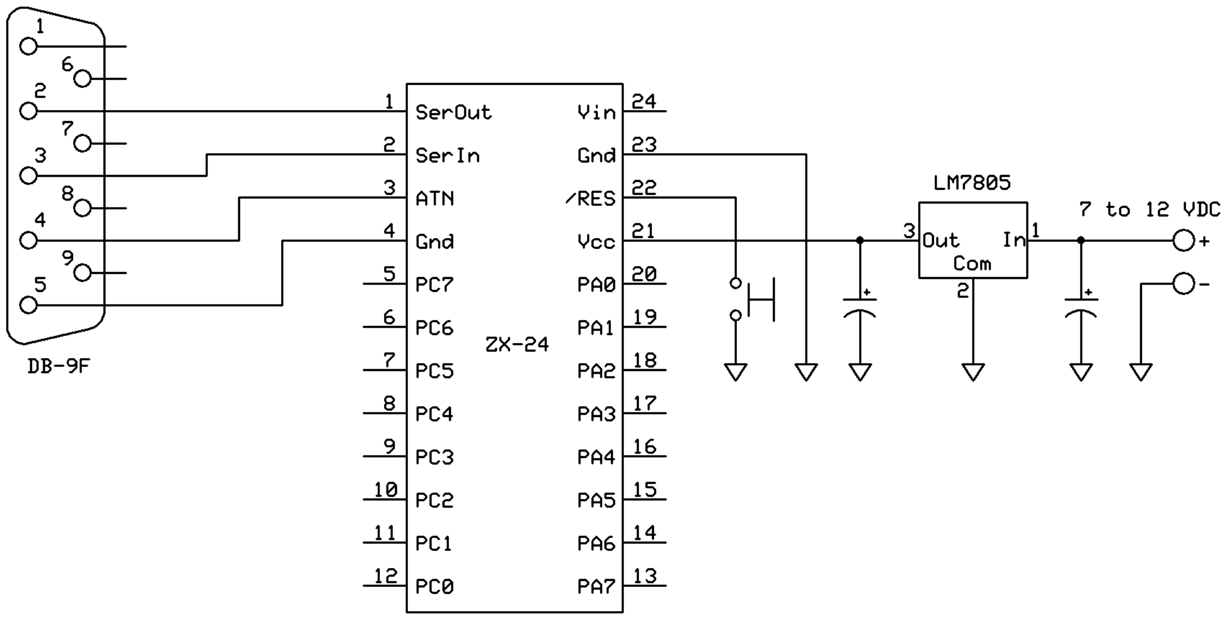
The ZX-24 series devices can be hooked up in several different ways depending on your requirements. The simplest method is depicted below. The connector on the left is the serial connection to your PC for downloading code and transmitting/receiving data on serial channel 1. The connection to pin 3 of the ZX-24 series is only needed for downloading. You may wish to add a jumper in that line so that it may be disconnected for "normal" operation. Some operating systems, notably Windows XP, toggle the DTR line (pin 4 of the DB-9) on the serial ports during boot up. This will cause the ZX to reset on every positive-going transition of the DTR line.



Simple ZX-24 Series Interconnection

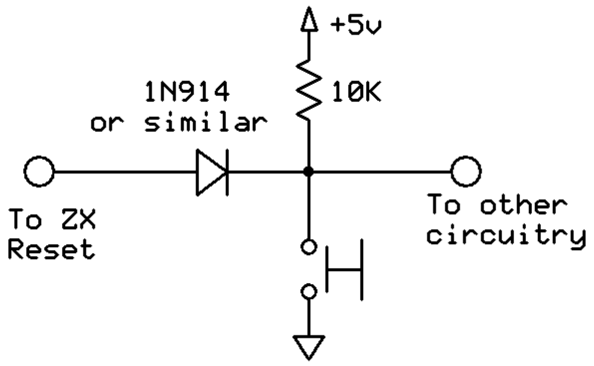
By default, the serial connection to the ZX-24 series devices is 19.2K baud. You can operate the serial channel at higher and lower speeds by explicitly opening the COM1 serial channel. See the description of OpenCom() in the ZBasic System Library Reference manual.

The simple configuration shown above is limited by the capacity of the on-board regulator. Although it will work for simple systems, many users will benefit from powering the ZX-24 series device from an externally regulated source. A suggested connection for such a configuration is shown below. The output of the external regulator can also be used to power additional external circuitry up to the capacity of the regulator. Depending on the power dissipated by the regulator, it may need to be mounted on a heat sink to prevent it from getting too hot.



ZX-24 Series Device Interconnection Using an External Regulator

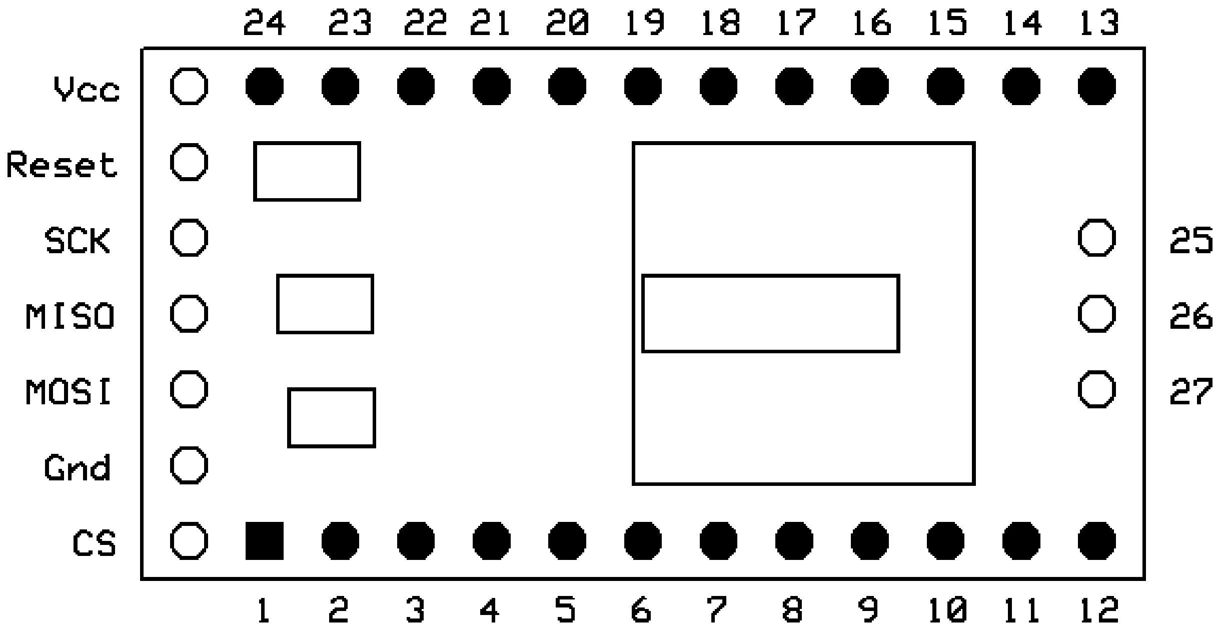
This configuration also shows how to connect a reset switch. A reset switch is sometimes useful, particularly during debugging, to get the ZX to start running its program from the beginning. If the reset is also connected to other external circuitry, you may wish to isolate the ZX from the remaining circuitry so that the on-board resets that occur during downloading are not also applied to the external devices. A simple isolation circuit employing a small signal diode is shown below. You can also implement an isolation circuit using a discrete transistor or an open-collector gate.



Reset Signal Isolation

B.2 Pin Configuration

The ZX-24 series devices have 24 male pins, numbered 1 through 24 in a standard DIP configuration. Additionally, there are 10 other connections that may be made to the ZX-24 series circuit board for expansion and other special purposes. In the diagram below, the male pins are represented by filled circles while the open circles represent solderable connections. The filled square indicates pin 1. The remaining rectangles on the diagram are stylized representations of some of the components on the board and are shown for orientation purposes only. Pins 1 to 24 are referred to as the standard pins while the remaining connections are referred to as the expansion pins.



ZX-24 Series Pin Configuration

B.2.1 Standard Pins

The table below summarizes the functions of the standard ZX-24 series pins. A detailed description of each pin is given following the table. Note that several of the external pins connect to more than one pin of the microcontroller chip. The multiple connections allow such pins to serve more than one function. See the descriptions below and the Atmel documentation for more information.

Standard ZX-24 Series Pins

Pin	Description
1	Serial Output
2	Serial Input
3	ATN
4	Ground (common with pin 22)
5	Port C, bit 7 and Port B bit 1
6	Port C, bit 6 and Port D bit 2
7	Port C, bit 5
8	Port C, bit 4
9	Port C, bit 3
10	Port C, bit 2
11	Port C, bit 1 and Port D bit 3
12	Port C, bit 0 and Port D bit 6
13	Port A, bit 7
14	Port A, bit 6
15	Port A, bit 5
16	Port A, bit 4

17	Port A, bit 3
18	Port A, bit 2 and Port B bit 2
19	Port A, bit 1
20	Port A, bit 0
21	+V out or +V in (regulated)
22	Reset (in and out)
23	Ground (common with pin 4)
24	+V in (unregulated)

Detailed Pin Descriptions

Pin 1, Serial Output

This is the output pin for serial channel 1 and Debug.Print. The voltage swing is approximately 0V to the supply voltage (+5V if using the on-board regulator) and can supply a maximum of 25mA of current. Note that the voltage swing does not meet the specifications for the RS-232 standard but it will be properly recognized by most modern serial interfaces.

This pin should be connected to pin 2 of a DB-9F serial connector.

Pin 2, Serial Input

This is the input pin for serial channel 1. It will tolerate a voltage swing from –15V to +15V allowing it to be connected to an RS-232 output from a serial device.

This pin should be connected to pin 3 of a DB-9F serial connector.

Pin 3, ATN

This input is used for signaling the ZX to enter the download mode. If you don't need to download, no connection is required but you may wish to ground this pin to eliminate the possibility of spurious resets caused by electrical noise.

For downloading, this pin should be connected to pin 4 of a DB-9F serial connector. Note that some operating systems (particularly Windows XP) tend to toggle the DTR serial port line, to which this input is normally connected, during the booting process. Every positive transition on this input will cause the ZX to reset.

Pin 4, Ground

This is the ground connection for the serial port and is common with pin 23. This pin should be connected to pin 5 of a DB-9F serial connector.

Pin 5, Port C Bit 7 and Port B Bit 1

In addition to being a general input or output pin, a signal may be applied to this pin to serve as the clock source for Timer 1. Port C Bit 7 must be an input for this to work properly. Alternately, if no external signal is connected, Port C Bit 7 may be an output and serve as the clock source for the timer.

Pin 6, Port C Bit 6 and Port D Bit 2

In addition to being a general input or output pin, a signal may be applied to this pin to serve as external Interrupt 0. Port C Bit 6 must be an input for this to work properly. Alternately, if no external signal is connected, Port C Bit 6 may be an output and serve as the source for the interrupt.

Pins 7-10, Port C Bits 5-2

These pins may be used as general inputs or outputs.

Pin 11, Port C Bit 1 and Port D bit 3

In addition to being a general input or output pin, a signal may be applied to this pin to serve as external Interrupt 1. Port C Bit 1 must be an input for this to work properly. Alternately, if no external signal is connected, Port C Bit 1 may be an output and serve as the source for the interrupt. Additionally, this pin serves as the SDA line when using I2C channel 0.

Pin 12, Port C Bit 0 and Port D bit 6

In addition to being a general input or output pin, a signal may be applied to this pin to serve as the input for the InputCapture() routine. Port C Bit 0 must be an input for this to work properly. Additionally, this pin serves as the SCL line when using I2C channel 0.

Pins 13-17, Port A Bits 7-3

In addition to being general digital input or output pins, these pins may be used for analog input to the analog-to-digital converter. The analog voltage may be read using the GetADC() routine.

Pin 18, Port A Bit 2 and Port B Bit 2

In addition to being a general digital input or output pin this pin may be used for analog input to the analog-to-digital converter. Alternately, a signal may be applied to this pin to serve as external Interrupt 2. Port A Bit 2 must be an input for this to work properly. Moreover, if no external signal is connected, Port A Bit 2 may be an output and serve as the source for the interrupt. A second alternate use for this pin is as the analog comparator input. See the Atmel documentation for the AIN0 input for more details.

Pins 19-20, Port A Bits 1-0

In addition to being general digital input or output pins, these pins may be used for analog input to the analog-to-digital converter. The analog voltage may be read using the GetADC() routine.

Pin 21, Regulated Voltage In or Regulated Voltage Out

If the on-board regulator is being used (power being fed to pin 24) this pin will output a regulated voltage of about +5 volts. The capacity of the on-board regulator is approximately 100mA. About 60mA of this capacity is used by the on-board devices with all pins in the input state. If any of the pins are outputs, the source current of each output pin must be added to this usage. The balance of the regulator's capacity (very little, in most cases) may be used by external circuitry.

Except for the simplest configurations, it is advisable not to use the on-board regulator and, instead, supply a regulated voltage of 4.5 to 5.5 volts to pin 21. In this case, no connection should be made to pin 24.

Pin 22, Reset

You may apply an active low signal to this pin to reset the processor. Note, however, that the on-board reset circuitry will also pull this line low via an open collector output each time the ATN input makes a positive transition (notably during downloading). If you don't want your external circuitry to receive this reset signal, you must isolate the on-board reset from external devices using a diode, a transistor or an open collector gate. An example circuit is shown in Section B.1.

Pin 23, Ground

This pin, common with pin 4, serves as the reference for the power supply and all I/O pins.

Pin 24, Unregulated Voltage In

To use the on-board regulator, you may supply an unregulated (but filtered) voltage source from 7 to 20 volts DC to this pin. The power supply must be capable of supplying at least 150mA. See the description of pin 21 for more information about the on-board regulator and the limitations of using it. It is important to note that the maximum input voltage must be derated above an ambient temperature of 45°C according to the formula $V_{in} <= 5 + (125 – T_{amb}) / 5.2$. For example, with an ambient temperature of 75°C the input voltage must be kept below 14.6V. For operating in high ambient temperatures it is recommended to use an external regulator with an appropriate heatsink.

Additional Notes:

Each of the I/O pins (5-20, 25-27) can source 20mA or sink 40mA (at Vcc=5V). However, the total source current and the total sink current for all pins may not exceed 200mA. This is further limited by the capacity of the on-board regulator if it is being used. At Vcc=3V, the maximum source and sink currents are one-half of the 5V rating.

All of the I/O pins (5-20, 25-27) have protection diodes built in. If the signals that you connect to these pins go above V+ or below ground, you must include a current limiting resistor to keep the pin current below 20mA.

B.2.2 Expansion Pins

The table below summarizes the functions of the ZX-24 series expansion pins. These “pins” are actually holes on the circuit board that require a soldered or mechanical clip connection to use. A detailed description of each pin is given following the table.

Expansion Pins

Pin	Description
25	Port D Bit 7, Red LED
26	Port D Bit 5, Green LED
27	Port D Bit 4, OutputCapture pin
Vcc	Power, common with pin 21
Reset	Reset, common with pin 22
SCK	SPI Clock
MISO	SPI Master In Slave Out
MOSI	SPI Master Out Slave In
Gnd	Ground, common with pins 4 and 23
CS	Program Memory Chip Select

Detailed Pin Descriptions

Pin 25, Port D Bit 7

This pin drives the red LED and may also be used for PWM generation based on Timer 2. See the Atmel ATmega32, ATmega644 or ATmega644p documentation for more details on the latter function.

Pin 26, Port D Bit 5

This pin drives the green LED and may also be used for PWM generation based on Timer 1. See the Atmel ATmega32, ATmega644 or ATmega644p documentation for more details on the latter function.

Pin 27, Port D Bit 4

This pin is the output point for the standard OutputCapture() routine. It may also be used for PWM generation based on Timer 1. See the Atmel ATmega32, ATmega644 or ATmega644p documentation for more details on the latter function.

SCK, MISO, MOSI

These pins are the inputs and outputs for the SPI bus by which external SPI devices may be connected to the ZX-24/ZX-24a.

Vcc, Reset, Gnd, CS

These pins are used in the manufacturing and testing processes.

Appendix C - ZX-40 Series Hardware Reference

The ZX-40 devices are the 40-pin DIP package versions of the Atmel AVR ATmega32, ATmega644 or ATmega644P microcontrollers that have been programmed with the ZX control firmware. They offer the same capabilities as the ZX-24 series plus access to 7 additional I/O pins. In order to use the ZX-40 series devices you must add several additional components as described below. In each of the diagrams presented below, only a portion of the device's 40 pins is shown. Those that are not germane to the circuit being discussed are omitted for clarity.

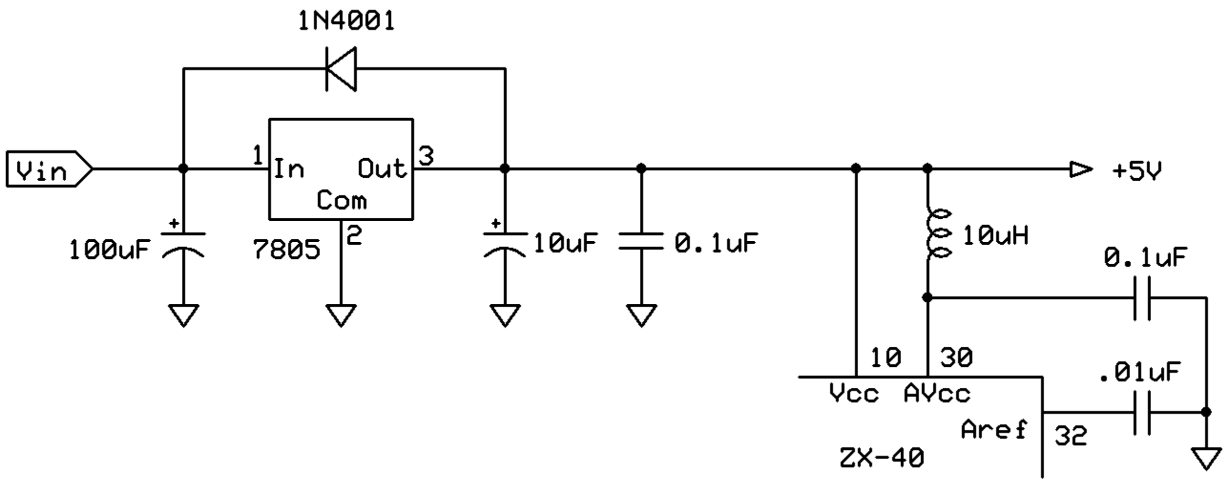
C.1 ZX-40 Series Specifications

The electrical specifications of the ZX-40, ZX-40a, ZX-40p are exactly those of the underlying ATmega microcontrollers. Rather than reproducing them here the reader is directed to the datasheet published by Atmel. It can be obtained from the Atmel website http://www.atmel.com or from the ZBasic website http://www.zbasic.net.

C.2 ZX-40 Series Required External Components

Power Source

The ZX-40 series devices need a regulated voltage source capable of providing at least 200mA of current. The voltage typically used is 5 volts but the device will operate between 4.5 volts and 5.5 volts. A recommended circuit is shown below. A suitable heatsink will probably be required in most cases to keep the regulator IC below its maximum operating temperature. Consult the regulator datasheet for more information.



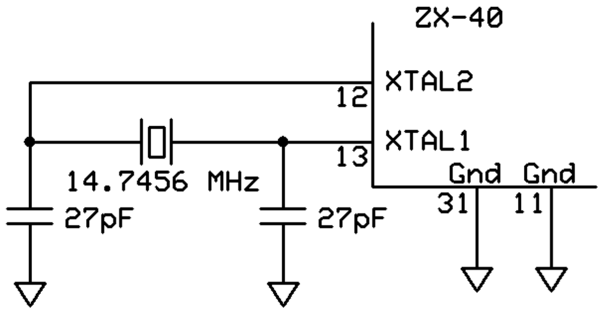
ZX-40 Series Power Source

If you do not plan to use the Analog-to-Digital converter channels you can eliminate the inductor and the two capacitors on the right side of the diagram. In this case, pin 30 would be connected directly to the power source (as are pins 10 and 30) and pin 32 can be left open. Although not shown on this diagram, the ground pins of the ZX-40 (pins 11 and 31) must be connected to the common ground of the system.

Clock Source

The ZX-40 series devices require a clock source running at 14.7456MHz. The easiest way to provide this clock source is to connect a crystal of that frequency to pins 12 and 13 along with the necessary capacitors.

The recommended crystal to use is ECS-147.4-20-4 (Digi-Key #X175-ND). An alternate crystal in a slightly larger package is ECS-147.4-20-1 (Digi-Key #X142-ND). The capacitor value to use depends somewhat on the layout (e.g. stray capacitance). Values in the range of 22pF to 27pF have been used successfully in various situations.



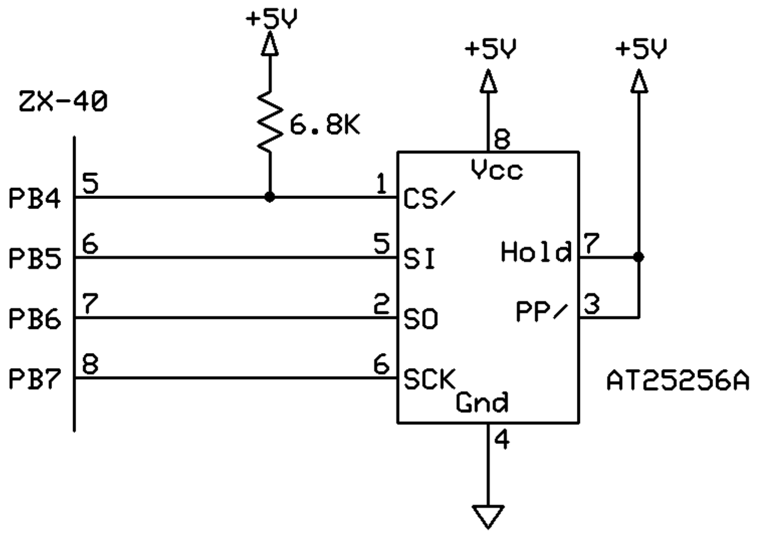
ZX-40 Series Clock Source

If you already have a clock source at the required frequency you can feed that signal directly to pin 13. In this case there would be no connection to pin 12.

Program Memory

The ZX-40 series devices (other than the ZX-40n) require a serial EEPROM in which to store your program's code. (The ZX-40n uses internal Flash memory for Program Memory.) The recommended device to use is the Atmel AT25256A (Digi-Key #AT25256A-10PI-2.7-ND). Note, particularly, that the older part without the A suffix will not work due to its slower speed. An EEPROM capable of operation at 7.5MHz or more is required.

The recommended connection to the ZX-40 series is shown below. The value of the pullup resistor is non-critical. Anything from 4.7K to 22K should work well. Higher values will reduce power consumption somewhat.



ZX-40 Series Program Memory (VM models only)

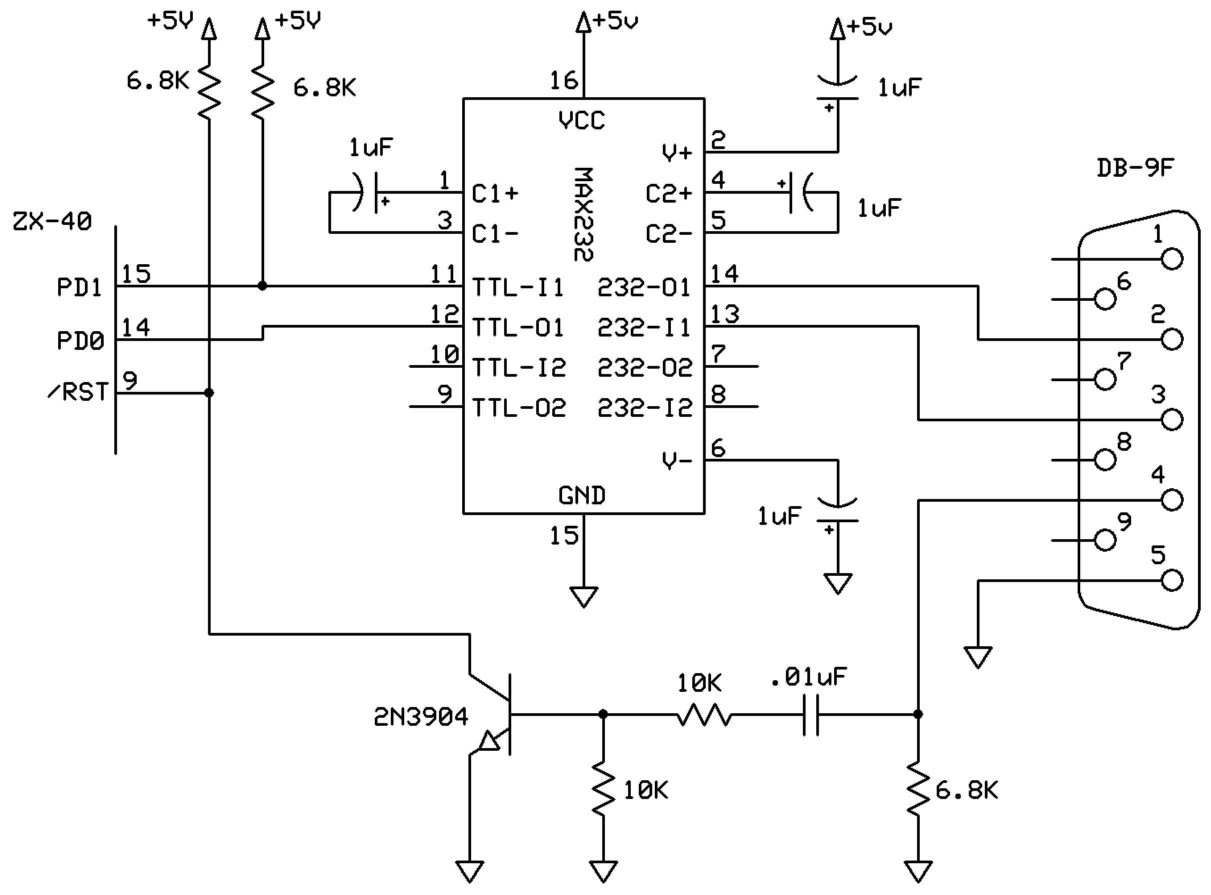
With firmware versions v1.1 and later, the Atmel 25HP512 (Digi-Key #AT25HP512-10PI2.7-ND) can also be used for applications that require larger EEPROM space. The electrical connections for this device are identical to those shown above. However, since this device has different characteristics than the AT25256A, special configuration is required. See Section 8.2 for more information.

Serial Interface

The ZX-40 series devices require a serial interface for downloading code into Program Memory, performing field updates of the control program, and for your program's use via Com1. The recommended serial interface circuitry is shown below. This circuit has two sub components. The Max232 chip functions as an RS-232 level converter that translates the 0-5 volt signals of the ZX to the standard RS-232 voltage levels.

The second component is the ATN circuitry that is used to signal the ZX to go into download mode. The latter circuitry is that portion connected between pin 4 of the DB-9 serial connector and pin 9 of the ZX. You may want to include a jumper in the ATN circuitry so that it can be disconnected when downloads are not required. With the ATN circuitry connected, the ZX will receive a reset pulse on every positive transition of pin 4 (DTR) of the serial connector. If alternate component values are chosen you must ensure that the ZX receives a reset pulse of at least 2uS on every positive transition of the DTR signal on pin 4 of the serial connector.

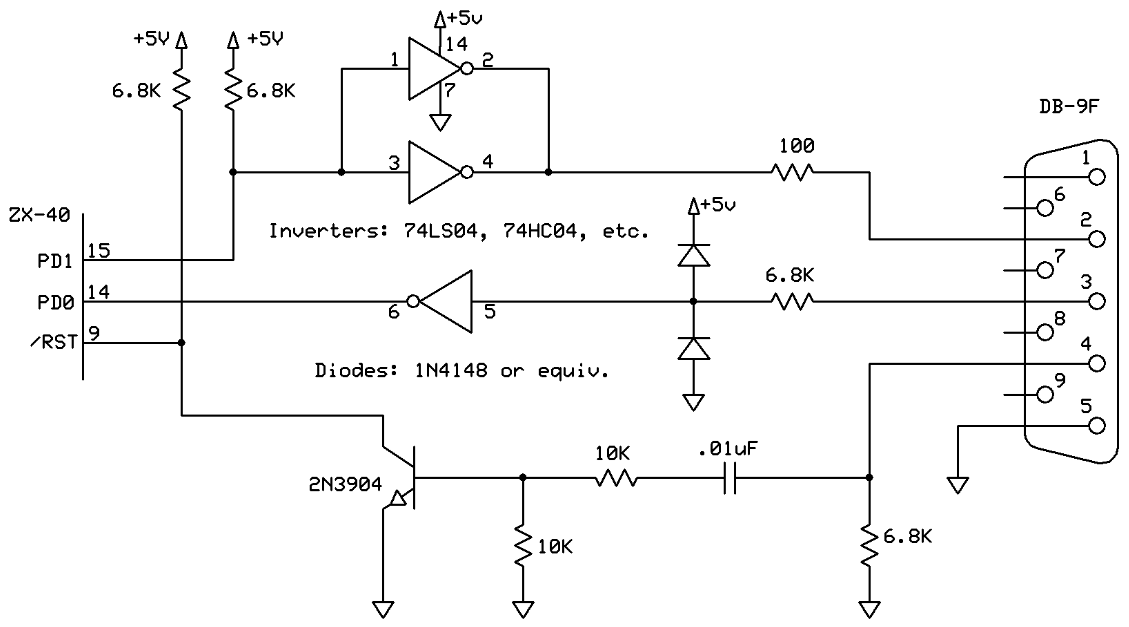
One advantage to using this recommended circuit is that the MAX232 chip includes a DC-DC converter that produces nominally +/- 12V from the 5 volt source. This is particularly useful if your application requires one or more of these voltages. For example, some LCD devices require a low-current negative supply for their backlight circuitry. The positive voltage is available on pin 2 of the Max232 while the negative voltage is available on pin 6. Consult the datasheet for the MAX232 device for information on the current capacity of these supplies.



Recommended ZX-40 Series Serial Interface Circuit

The pullup resistor shown on the ZX's transmit output (pin 15) is needed to ensure that the serial output line stays in the idle state during reset cycles. Its value is non-critical – anything from 4.7K to 22K should work fine. The pullup resistor on the reset input isn't technically required since the chip itself has an internal pullup. The external pullup could be eliminated or could be made larger if desired.

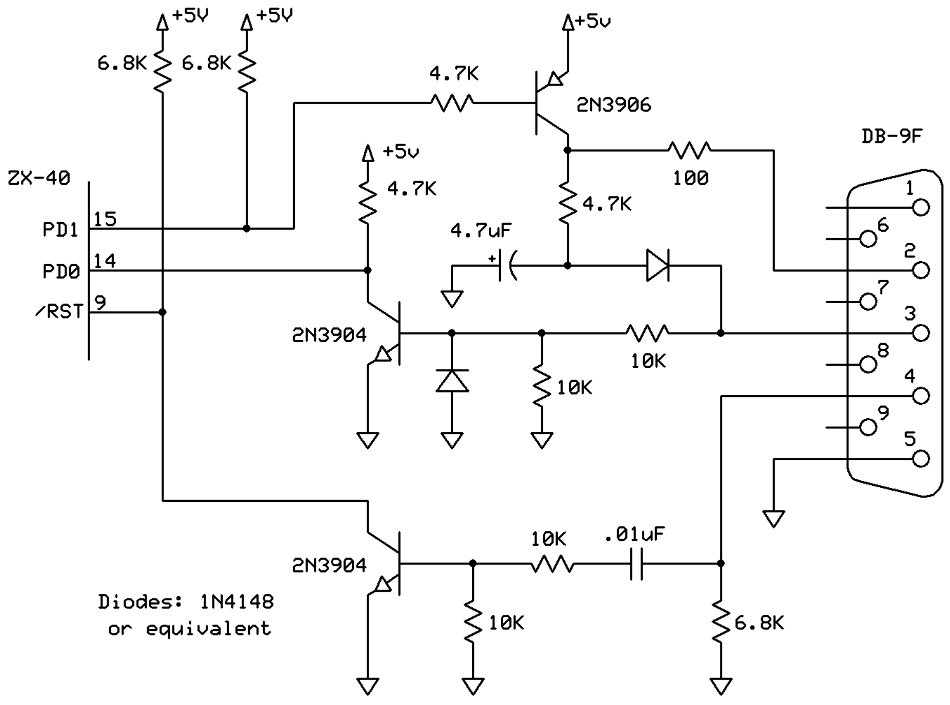
There are other serial interface circuits that would provide the required functionality, two of which are shown below. Both alternate circuits use the same ATN circuitry as the recommended circuit above.



Alternate Serial Interface Circuit #1

The first alternate above is similar to the circuitry used on the ZX-24 series devices. The advantage to this circuit is its simplicity. The disadvantage is that the voltage levels of the serial output signal do not meet the RS-232 standard. Even so, most serial receivers are able to properly interpret the output signal. Note that if the inverters used have protection diodes on their inputs the external diodes shown are not necessary.

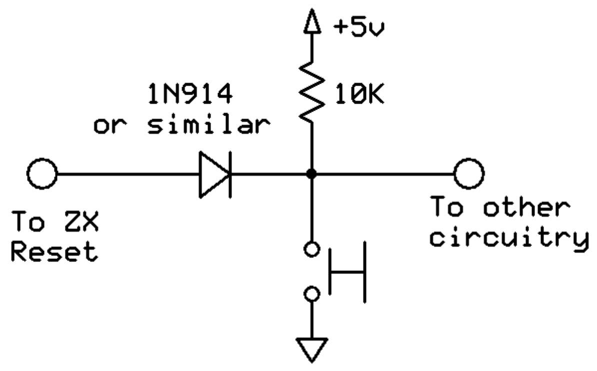
The second alternate serial interface is shown in the diagram below. This circuit produces a serial output signal that meets the RS-232 standard by using the negative voltage on the receive line to provide a reference for the output level. Although the parts count is higher, the circuit uses inexpensive and non-critical components.



Alternate Serial Interface Circuit #2

Reset Circuit

You may also want to incorporate a reset button in your circuitry. If so, a normally open, momentary contact switch may be connected directly between pin 9 of the ZX and ground. If other circuitry in your application also needs the reset signal, you may want to isolate that circuitry from the ATN resets using a buffer gate or a diode. A suggested diode isolation circuit for the reset line is shown below.



Reset Signal Isolation

Appendix D - ZX-44 Series Hardware Reference

The ZX-44 is the 44-pin TQFP package version of the Atmel AVR ATmega32 microcontroller that has been programmed with the ZX control firmware. Similarly, the ZX-44a is the 44-pin TQFP package version of the Atmel AVR ATmega644 microcontroller. They offer the same capabilities as the ZX-24 plus access to 7 additional I/O pins. In order to use the ZX-44/ZX-44a you must add several additional components as described below. In each of the diagrams presented below, only a portion of the ZX pins is shown. Those that are not germane to the circuit being discussed are omitted for clarity.

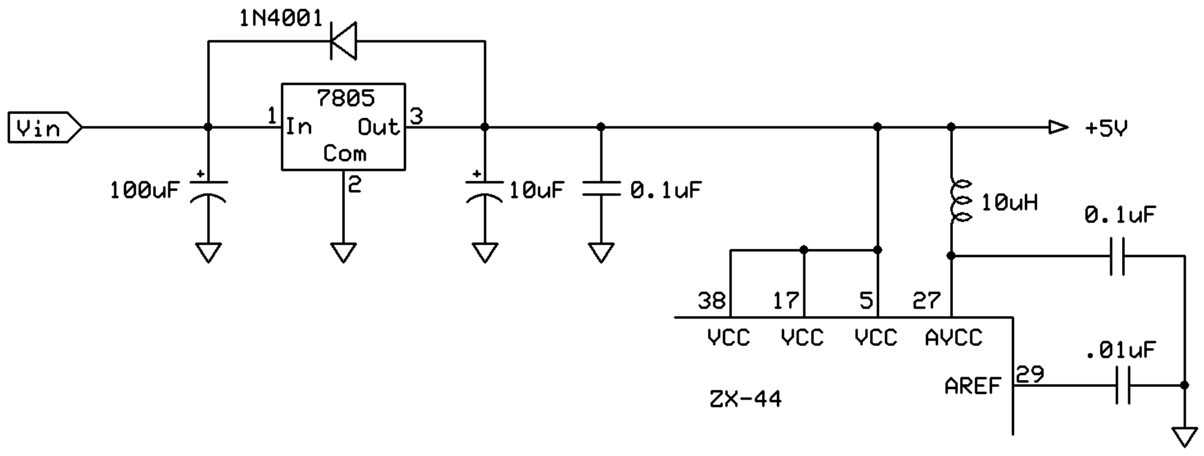
D.1 ZX-44 Series Specifications

The electrical specifications of the ZX-44 are exactly those of the ATmega32. Likewise, the electrical specifications of the ZX-44a are those of the ATmega644. Rather than reproducing them here the reader is directed to the datasheet published by Atmel. It can be obtained from the Atmel website http://www.atmel.com or from the ZBasic website http://www.zbasic.net.

D.2 ZX-44 Series Required External Components

Power Source

The ZX-44 series devices need a regulated voltage source capable of providing at least 200mA of current. The voltage typically used is 5 volts but the ZX-44 series devices will operate between 4.5 volts and 5.5 volts. A recommended circuit is shown below. A suitable heatsink will probably be required in most cases to keep the regulator IC below its maximum operating temperature. Consult the regulator datasheet for more information.



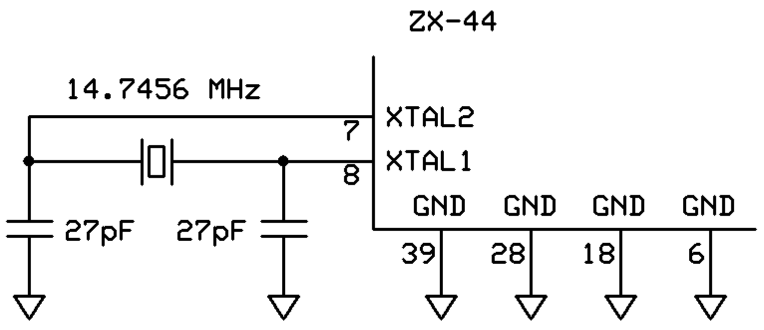
ZX-44 Series Power Source

If you do not plan to use the Analog-to-Digital converter channels you can eliminate the inductor and the two capacitors on the right side of the diagram. In this case, pin 27 would be connected directly to the power source (same as pin 5, etc.) and pin 29 can be left open. Although not shown on this diagram, the ground pins of the ZX-44 (pins 6, 18, 28 and 39) must be connected to the common ground of the system.

Clock Source

The ZX-44 series devices require a clock source running at 14.7456MHz. The easiest way to provide this clock source is to connect a crystal of that frequency to pins 7 and 8 along with the necessary capacitors.

The recommended crystal to use is ECS-147.4-20-4 (Digi-Key #X175-ND). An alternate crystal in a slightly larger package is ECS-147.4-20-1 (Digi-Key #X142-ND). The capacitor value to use depends somewhat on the layout (e.g. stray capacitance). Values in the range of 22pF to 27pF have been used successfully in various situations.



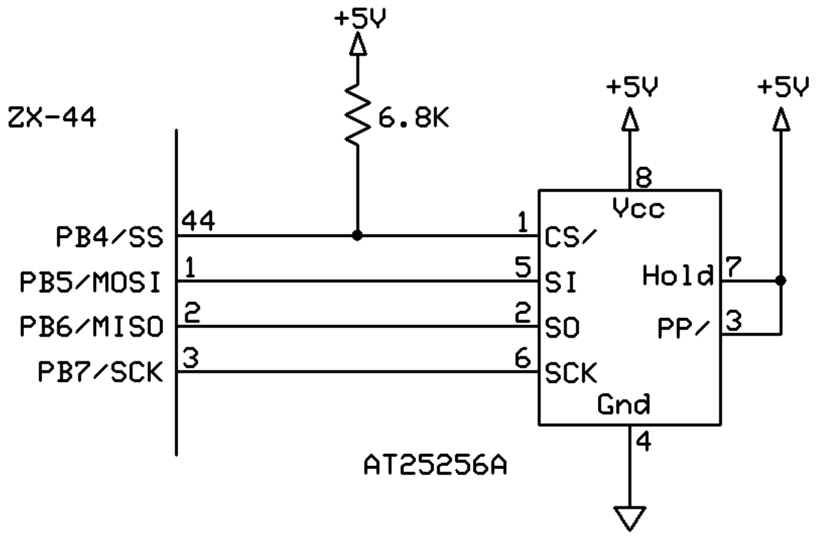
ZX-44 Series Clock Source

If you already have a clock source at the required frequency you can feed that signal directly to pin 8. In this case there would be no connection to pin 7.

Program Memory

The ZX-44 series devices (other than the ZX-44n) also require a serial EEPROM in which to store your program's code. (The ZX-40n uses internal Flash memory for Program Memory.) The recommended device to use is the Atmel AT25256A (Digi-Key #AT25256A-10PI-2.7-ND). Note, particularly, that the older part without the A suffix will not work due to its slower speed. An EEPROM capable of operation at 7.5MHz or more is required.

The recommended connection to the ZX is shown below. The value of the pullup resistor is non-critical. Anything from 4.7K to 22K should work well. Higher values will reduce power consumption somewhat.



ZX-44 Series Program Memory (VM devices only)

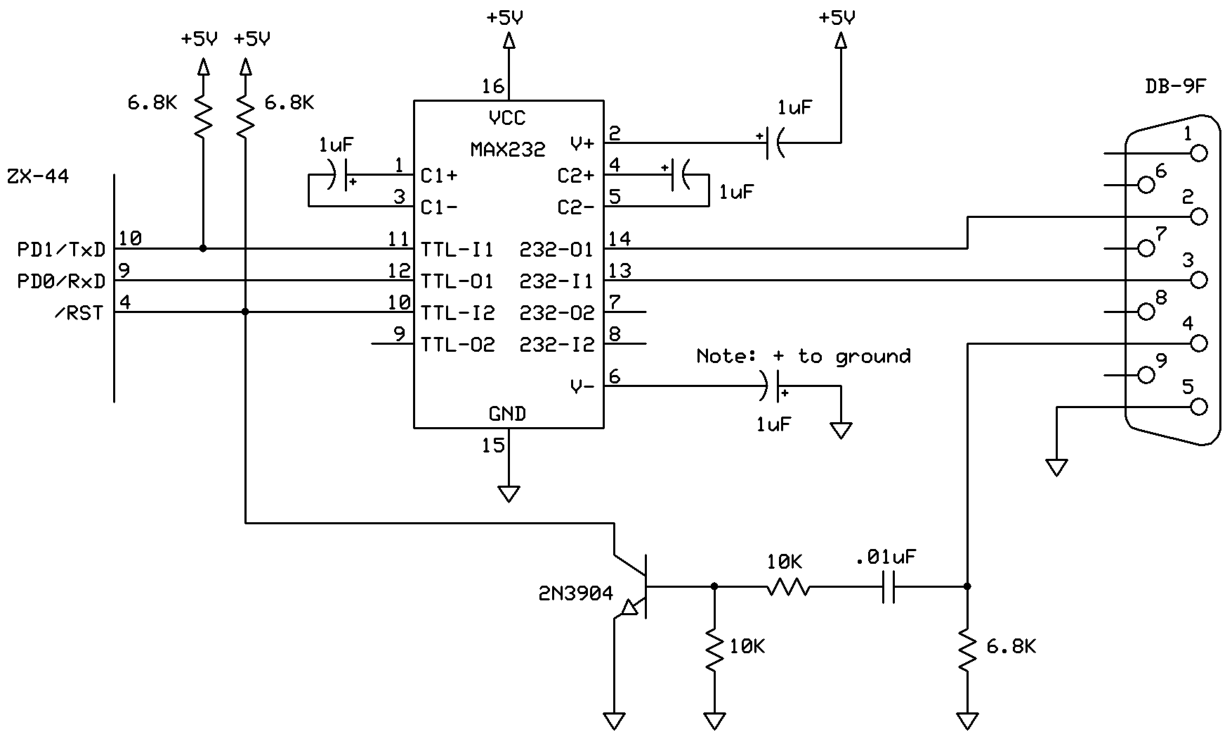
With firmware versions v1.1 and later, the Atmel 25HP512 (Digi-Key #AT25HP512-10PI2.7-ND) can also be used for applications that require larger EEPROM space. The electrical connections for this device are identical to those shown above. However, since this device has different characteristics than the AT25256A, special configuration is required. See Section 8.2 for more information.

Serial Interface

The ZX-44 series devices require a serial interface for downloading code into Program Memory, performing field updates of the control program, and for your program's use via Com1. The recommended serial interface circuitry is shown below. This circuit has two sub components. The Max232 chip functions as an RS-232 level converter that translates the 0-5 volt signals of the ZX to the standard RS-232 voltage levels.

The second component is the ATN circuitry that is used to signal the ZX to go into download mode. The latter circuitry is that portion connected between pin 4 of the DB-9 serial connector and pin 9 of the ZX. You may want to include a jumper in the ATN circuitry so that it can be disconnected when downloads are not required. With the ATN circuitry connected, the ZX will receive a reset pulse on every positive transition of pin 4 (DTR) of the serial connector. If alternate component values are chosen you must ensure that the ZX receives a reset pulse of at least 2uS on every positive transition of the DTR signal on pin 4 of the serial connector.

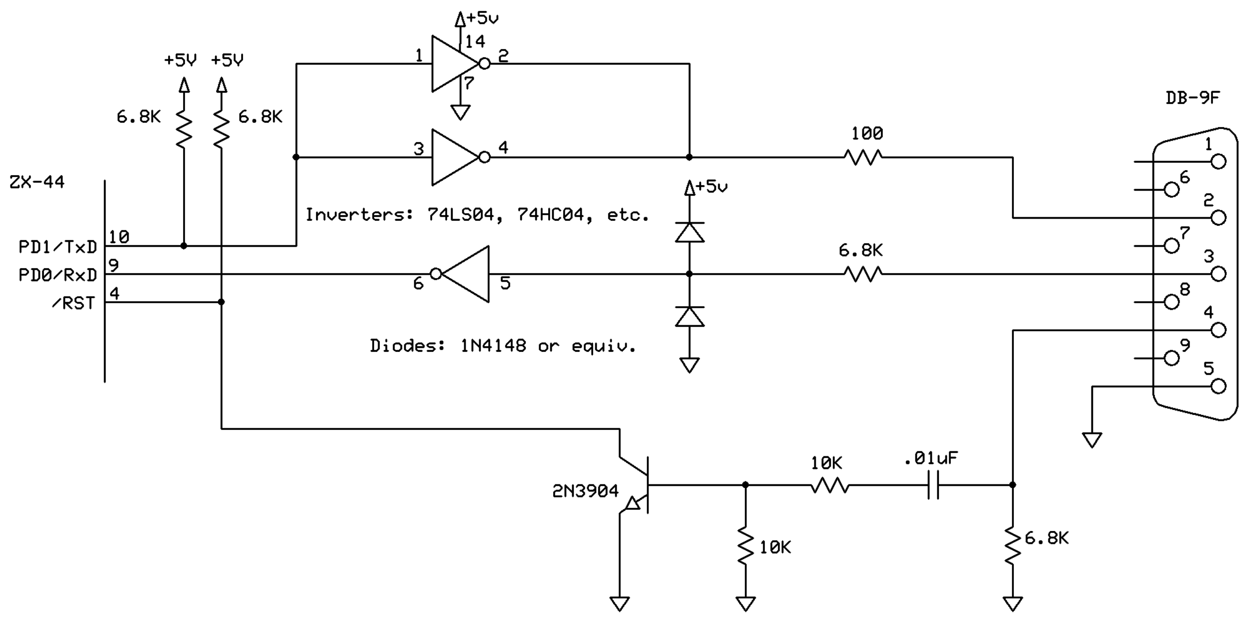
One advantage to using this recommended circuit is that the MAX232 chip includes a DC-DC converter that produces nominally +/- 12V from the 5 volt source. This is particularly useful if your application requires one or more of these voltages. For example, some LCD devices require a low-current negative supply for their backlight circuitry. The positive voltage is available on pin 2 of the Max232 while the negative voltage is available on pin 6. Consult the datasheet for the MAX232 device for information on the current capacity of these supplies.



Recommended ZX-44 Series Serial Interface Circuit

The pullup resistor shown on the ZX's transmit output (pin 10) is needed to ensure that the serial output line stays in the idle state during reset cycles. Its value is non-critical – anything from 4.7K to 22K should work fine. The pullup resistor on the reset input isn't technically required since the chip itself has an internal pullup. The external pullup could be eliminated or could be made larger if desired.

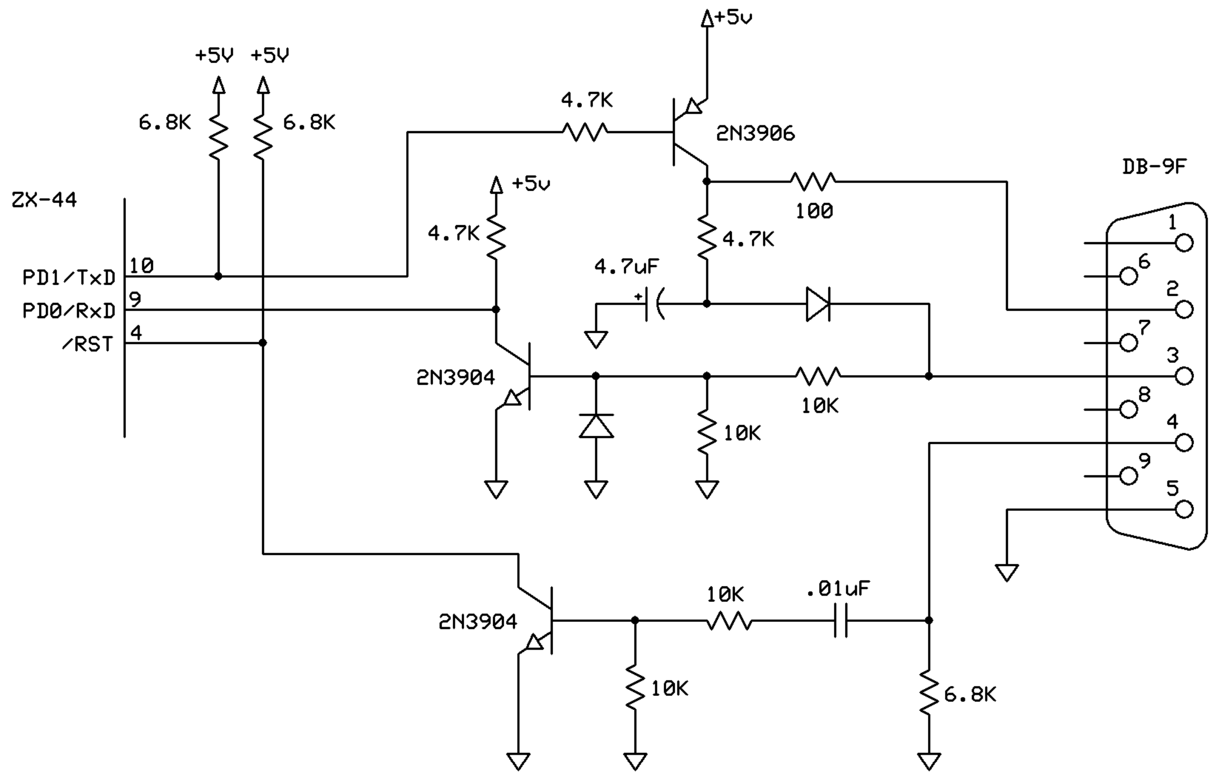
There are other serial interface circuits that would provide the required functionality, two of which are shown below. Both alternate circuits use the same ATN circuitry as the recommended circuit above.



Alternate Serial Interface Circuit #1

The first alternate above is similar to the circuitry used on the ZX-24 series. The advantage to this circuit is its simplicity. The disadvantage is that the voltage levels of the serial output signal do not meet the RS-232 standard. Even so, most serial receivers are able to properly interpret the output signal. Note that if the inverters used have protection diodes on their inputs the external diodes shown are not necessary.

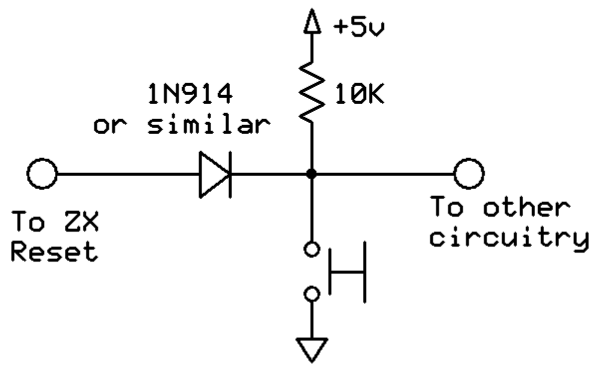
The second alternate serial interface is shown in the diagram below. This circuit produces a serial output signal that meets the RS-232 standard by using the negative voltage on the receive line to provide a reference for the output level. Although the parts count is higher, the circuit uses inexpensive and non-critical components.



Alternate Serial Interface Circuit #2

Reset Circuit

You may also want to incorporate a reset button in your circuitry. If so, a normally open, momentary contact switch may be connected directly between pin 4 of the ZX and ground. If other circuitry in your application also needs the reset signal, you may want to isolate that circuitry from the ATN resets using a buffer gate or a diode. A suggested diode isolation circuit for the reset line is shown below.



Reset Signal Isolation

Appendix E - ZX-1281 Series Hardware Reference

The ZX-1281 series devices are the 64-pin TQFP package version of the Atmel AVR ATmega1281 microcontroller that has been programmed with the ZX control firmware. In order to use the ZX-1281 you must add several additional components as described below. In each of the diagrams presented below, only a portion of the ZX pins is shown. Those that are not germane to the circuit being discussed are omitted for clarity.

E.1 ZX-1281 Series Specifications

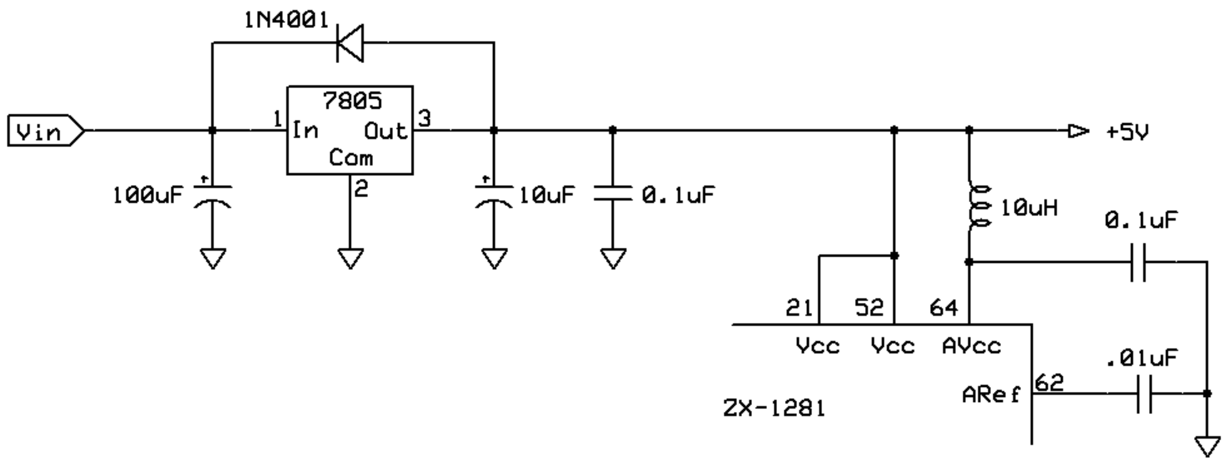
The electrical specifications of the ZX-1281 series devices are exactly those of the ATmega1281. Rather than reproducing them here the reader is directed to the datasheet published by Atmel. It can be obtained from the Atmel website http://www.atmel.com or from the ZBasic website http://www.zbasic.net.

E.2 ZX-1281 Series Required External Components

The circuits described below represent the minimum external circuitry required to operate the ZX-1281 series devices. Depending on your application, you may need additional circuitry to take advantage of the capabilities of the ZX-1281. In contrast to some other ZX devices, the ZX-1281 does not require an external EEPROM for program storage. Rather, the compiled user program is stored in the ZX's internal Flash memory. The maximum user program size is 60K bytes.

Power Source

The ZX-1281 series devices need a regulated voltage source capable of providing at least 200mA of current. The voltage typically used is 5 volts but the ZX-1281 will operate between 4.5 volts and 5.5 volts. A recommended circuit is shown below. A suitable heatsink will probably be required in most cases to keep the regulator IC below its maximum operating temperature. Consult the regulator datasheet for more information.



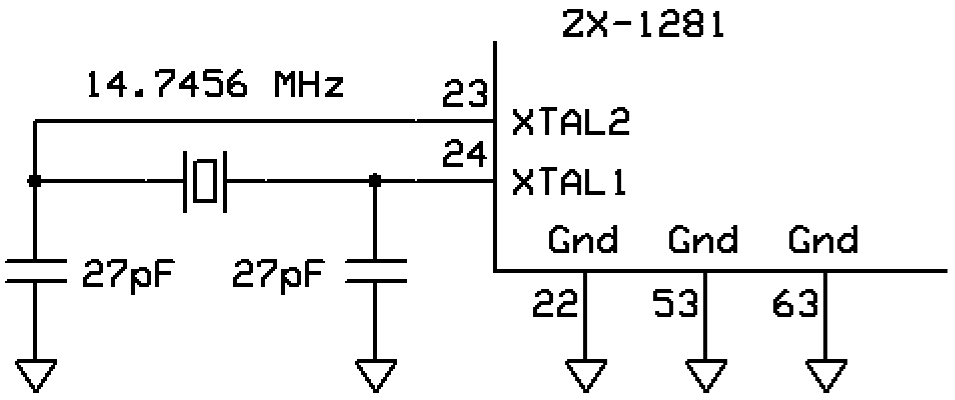
ZX-1281 Series Power Source

If you do not plan to use the Analog-to-Digital converter channels you can eliminate the inductor and the two capacitors on the right side of the diagram. In this case, pin 64 would be connected directly to the power source (same as pin 21, etc.) and pin 62 can be left open. Although not shown on this diagram, the ground pins of the ZX-1281 (pins 22, 53, and 63) must be connected to the common ground of the system.

Clock Source

The ZX-1281 series devices require a clock source running at 14.7456MHz. The easiest way to provide this clock source is to connect a crystal of that frequency to pins 23 and 24 along with the necessary capacitors.

The recommended crystal to use is ECS-147.4-20-4 (Digi-Key #X175-ND). An alternate crystal in a slightly larger package is ECS-147.4-20-1 (Digi-Key #X142-ND). The capacitor value to use depends somewhat on the layout (e.g. stray capacitance). Values in the range of 22pF to 27pF have been used successfully in various situations.



ZX-1281 Series Clock Source

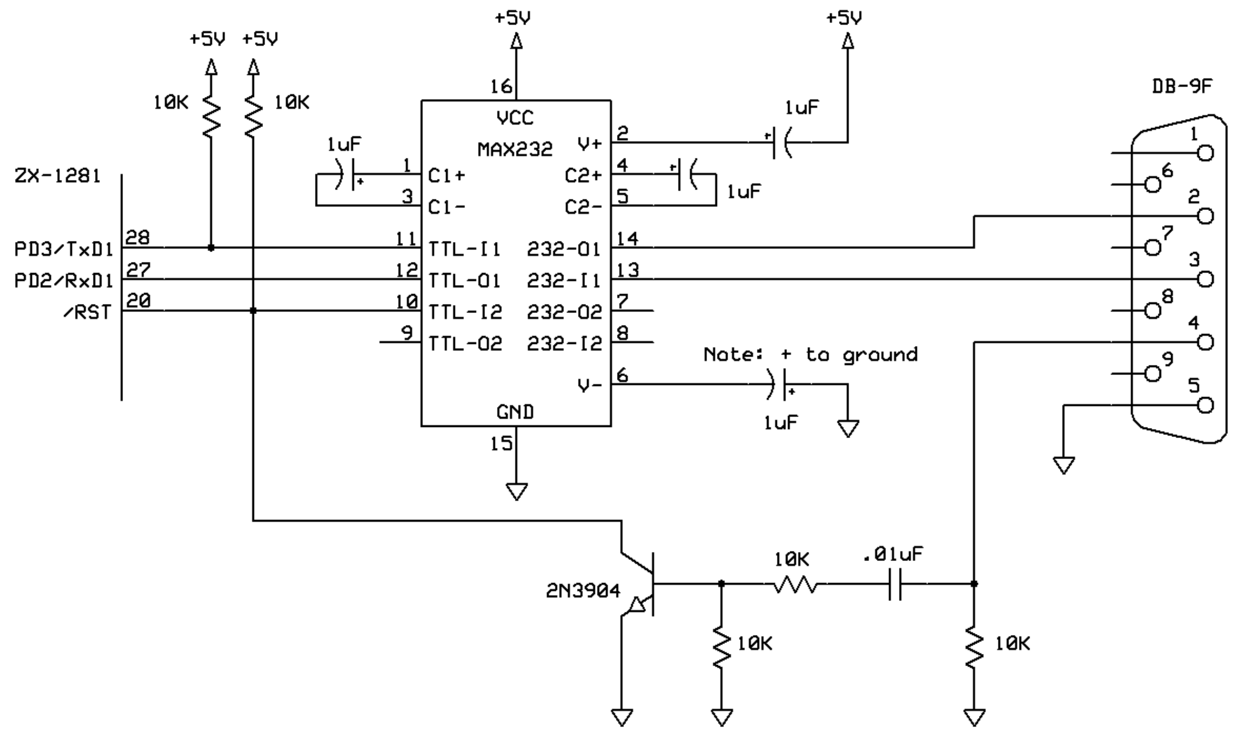
If you already have a clock source at the required frequency you can feed that signal directly to pin 24. In this case there would be no connection to pin 23.

Serial Interface

The ZX-1281 series devices require a serial interface for downloading code into Program Memory, performing field updates of the control program, and for your program's use via Com1. The recommended serial interface circuitry is shown below. This circuit has two sub components. The Max232 chip functions as an RS-232 level converter that translates the 0-5 volt signals of the ZX to the standard RS-232 voltage levels.

The second component is the ATN circuitry that is used to signal the ZX to go into download mode. The latter circuitry is that portion connected between pin 4 of the DB-9 serial connector and pin 20 of the ZX-1281. You may want to include a jumper in the ATN circuitry so that it can be disconnected when downloads are not required. With the ATN circuitry connected, the ZX will receive a reset pulse on every positive transition of pin 4 (DTR) of the serial connector. If alternate component values are chosen you must ensure that the ZX receives a reset pulse of at least 2uS on every positive transition of the DTR signal on pin 4 of the serial connector.

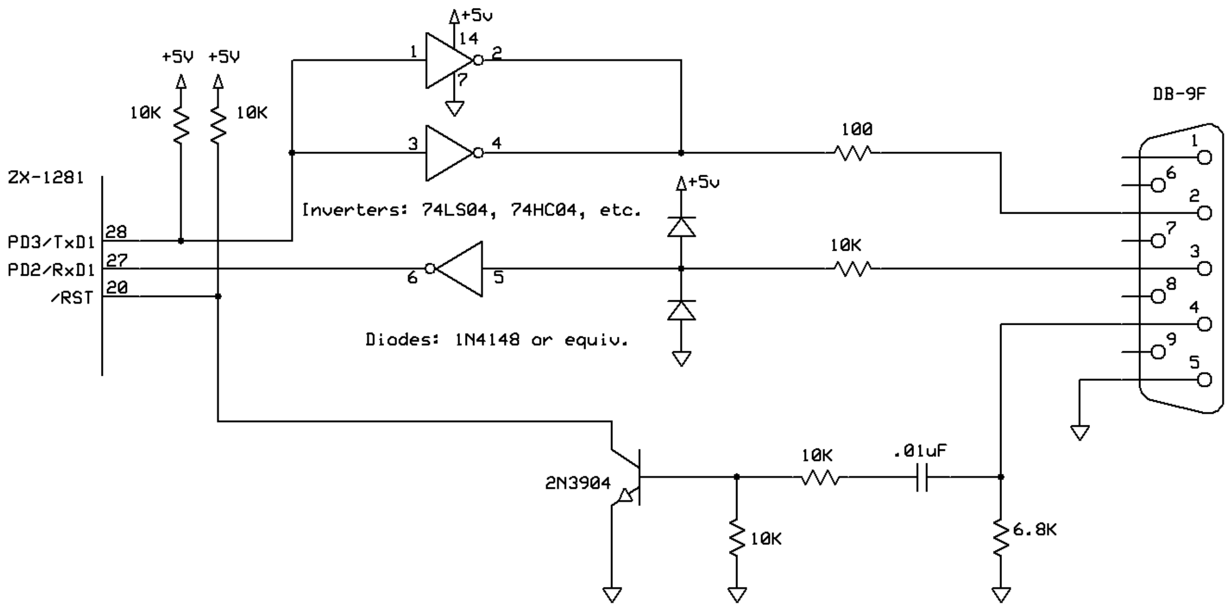
One advantage to using this recommended circuit is that the MAX232 chip includes a DC-DC converter that produces nominally +/- 12V from the 5 volt source. This is particularly useful if your application requires one or more of these voltages. For example, some LCD devices require a low-current negative supply for their backlight circuitry. The positive voltage is available on pin 2 of the Max232 while the negative voltage is available on pin 6. Consult the datasheet for the MAX232 device for information on the current capacity of these supplies.



Recommended ZX-1281 Series Serial Interface Circuit

The pullup resistor shown on the ZX's Com1 transmit output (pin 28) is needed to ensure that the serial output line stays in the idle state during reset cycles. Its value is non-critical – anything from 4.7K to 22K should work fine. The pullup resistor on the reset input isn't technically required since the chip itself has an internal pullup. The external pullup could be eliminated or could be made larger if desired.

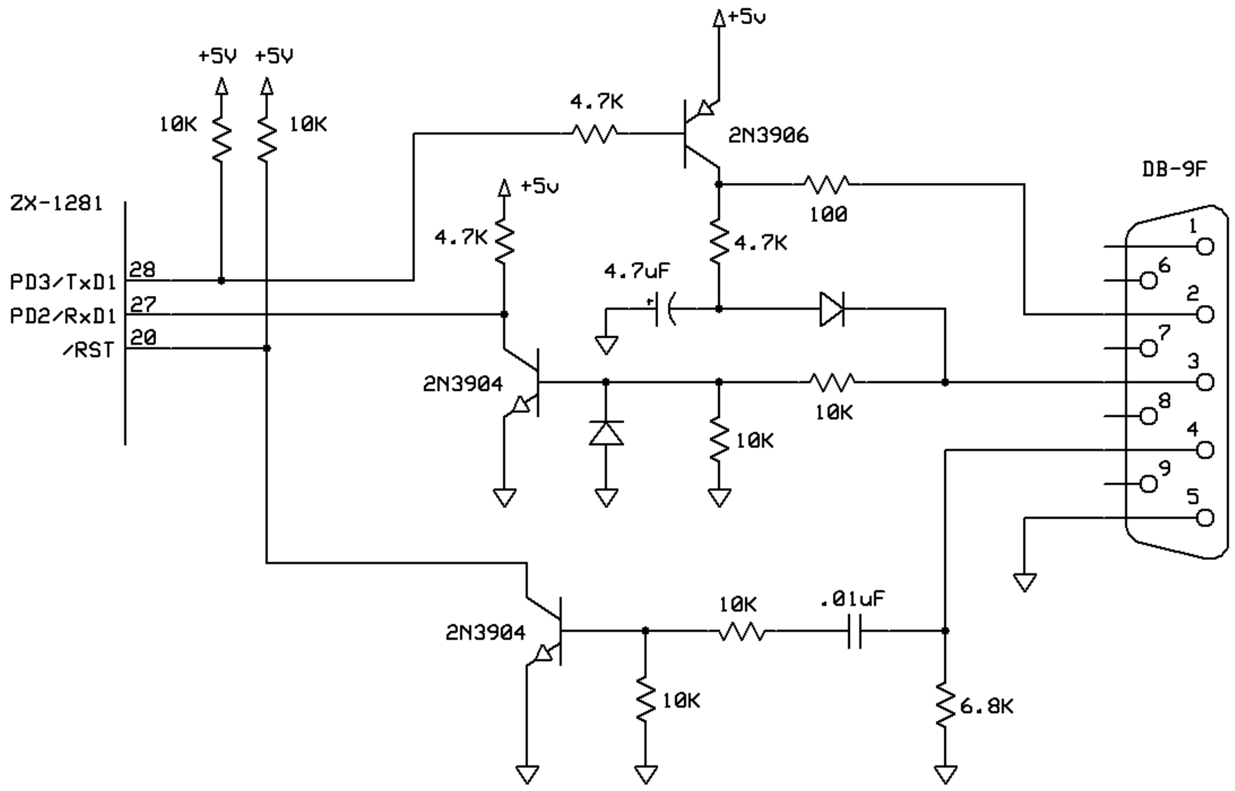
There are other serial interface circuits that would provide the required functionality, two of which are shown below. Both alternate circuits use the same ATN circuitry as the recommended circuit above.



Alternate Serial Interface Circuit #1

The first alternate above is similar to the circuitry used on the ZX-24 series devices. The advantage to this circuit is its simplicity. The disadvantage is that the voltage levels of the serial output signal do not meet the RS-232 standard. Even so, most serial receivers are able to properly interpret the output signal. Note that if the inverters used have protection diodes on their inputs the external diodes shown are not necessary.

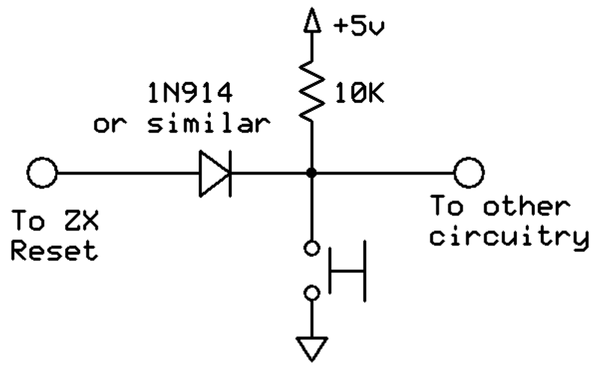
The second alternate serial interface is shown in the diagram below. This circuit produces a serial output signal that meets the RS-232 standard by using the negative voltage on the receive line to provide a reference for the output level. Although the parts count is higher, the circuit uses inexpensive and non-critical components.



Alternate Serial Interface Circuit #2

Reset Circuit

You may also want to incorporate a reset button in your circuitry. If so, a normally open, momentary contact switch may be connected directly between pin 20 of the ZX and ground. If other circuitry in your application also needs the reset signal, you may want to isolate that circuitry from the ATN resets using a buffer gate or a diode. A suggested diode isolation circuit for the reset line is shown below.



Reset Signal Isolation

Appendix F - ZX-1280 Series Hardware Reference

The ZX-1280 series devices utilize the 100-pin TQFP package version of the Atmel AVR ATmega1280 microcontroller that has been programmed with the ZX control firmware. In order to use the ZX-1280 you must add several additional components as described below. In each of the diagrams presented below, only a portion of the ZX pins is shown. Those that are not germane to the circuit being discussed are omitted for clarity.

F.1 ZX-1280 Series Specifications

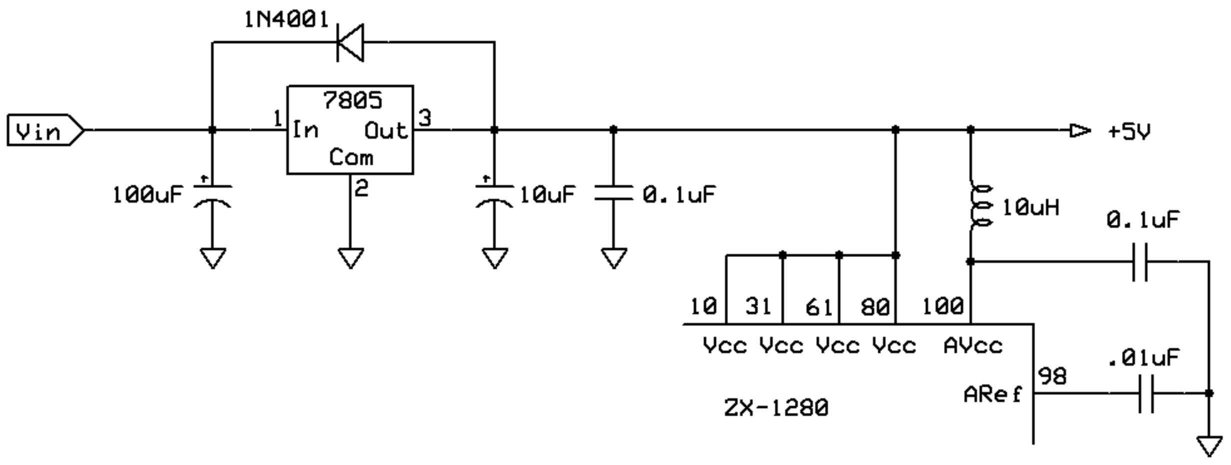
The electrical specifications of the ZX-1280 series devices are exactly those of the ATmega1280. Rather than reproducing them here the reader is directed to the datasheet published by Atmel. It can be obtained from the Atmel website http://www.atmel.com or from the ZBasic website http://www.zbasic.net.

F.2 ZX-1280 Series Devices Required External Components

The circuits described below represent the minimum external circuitry required to operate the ZX-1280. Depending on your application, you may need additional circuitry to take advantage of the capabilities of the ZX-1280 series device. In contrast to some other ZX devices, the ZX-1280 series does not require an external EEPROM for program storage. Rather, the compiled user program is stored in the ZX's internal Flash memory. The maximum user program size is 60K bytes.

Power Source

The ZX-1280 series devices need a regulated voltage source capable of providing at least 200mA of current. The voltage typically used is 5 volts but the ZX-1280 will operate between 4.5 volts and 5.5 volts. A recommended circuit is shown below. A suitable heatsink will probably be required in most cases to keep the regulator IC below its maximum operating temperature. Consult the regulator datasheet for more information.



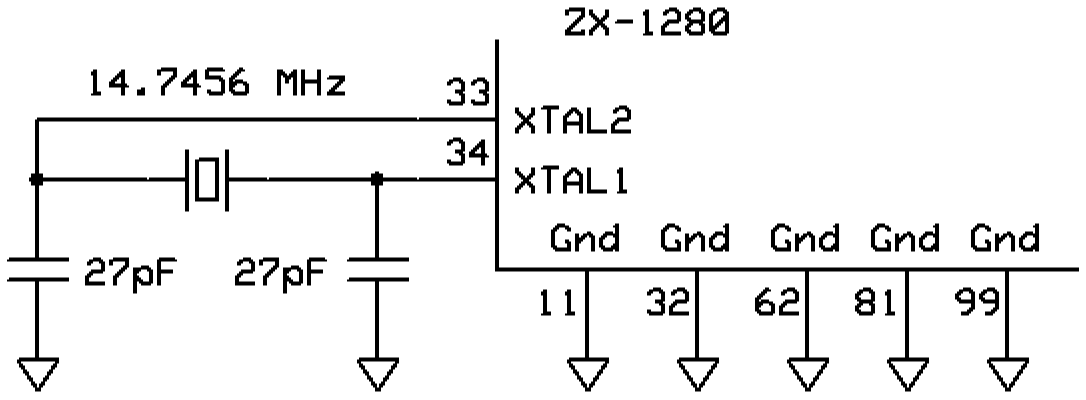
ZX-1280 Series Power Source

If you do not plan to use the Analog-to-Digital converter channels you can eliminate the inductor and the two capacitors on the right side of the diagram. In this case, pin 64 would be connected directly to the power source (same as pin 21, etc.) and pin 62 can be left open. Although not shown on this diagram, the ground pins of the ZX-1280 (pins 11, 32, 62, 81 and 99) must be connected to the common ground of the system.

Clock Source

The ZX-1280 series devices require a clock source running at 14.7456MHz. The easiest way to provide this clock source is to connect a crystal of that frequency to pins 33 and 34 along with the necessary capacitors.

The recommended crystal to use is ECS-147.4-20-4 (Digi-Key #X175-ND). An alternate crystal in a slightly larger package is ECS-147.4-20-1 (Digi-Key #X142-ND). The capacitor value to use depends somewhat on the layout (e.g. stray capacitance). Values in the range of 22pF to 27pF have been used successfully in various situations.



ZX-1280 Series Clock Source

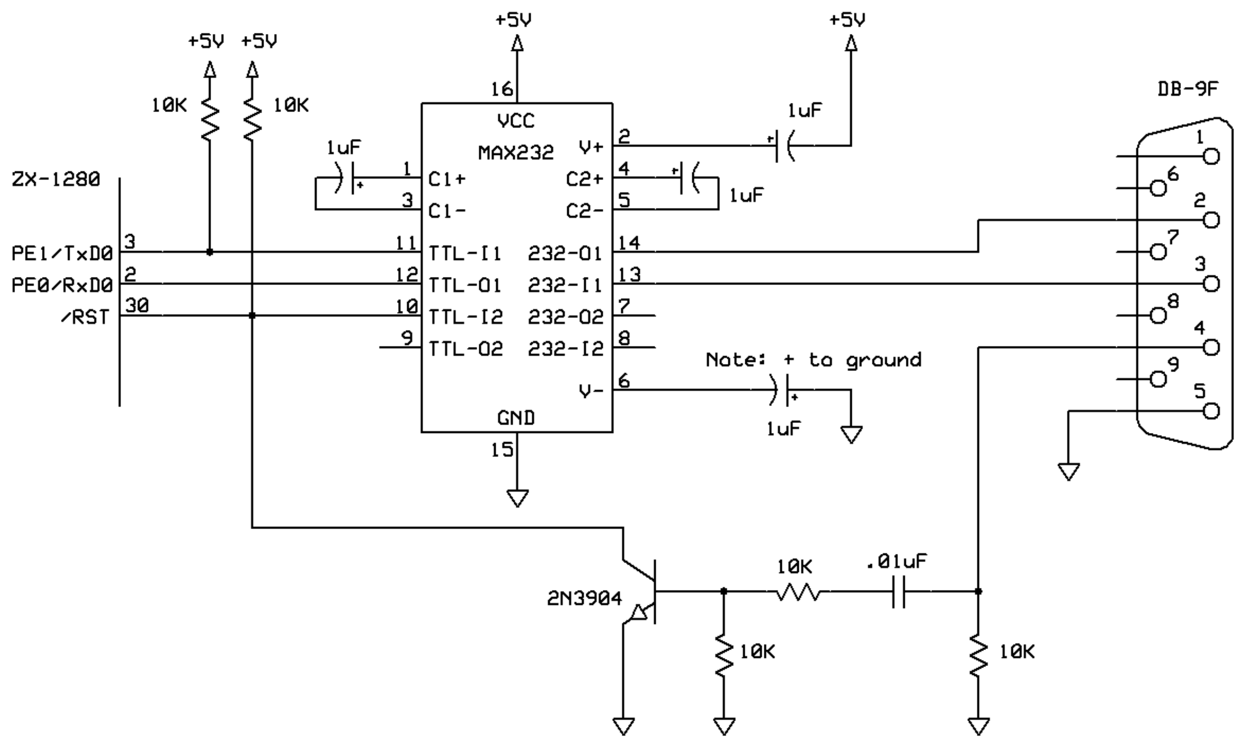
If you already have a clock source at the required frequency you can feed that signal directly to pin 34. In this case there would be no connection to pin 33.

Serial Interface

The ZX-1280 series devices require a serial interface for downloading code into Program Memory, performing field updates of the control program, and for your program's use via Com1. The recommended serial interface circuitry is shown below. This circuit has two sub components. The Max232 chip functions as an RS-232 level converter that translates the 0-5 volt signals of the ZX to the standard RS-232 voltage levels.

The second component is the ATN circuitry that is used to signal the ZX to go into download mode. The latter circuitry is that portion connected between pin 4 of the DB-9 serial connector and pin 20 of the ZX-1280. You may want to include a jumper in the ATN circuitry so that it can be disconnected when downloads are not required. With the ATN circuitry connected, the ZX will receive a reset pulse on every positive transition of pin 4 (DTR) of the serial connector. If alternate component values are chosen you must ensure that the ZX receives a reset pulse of at least 2uS on every positive transition of the DTR signal on pin 4 of the serial connector.

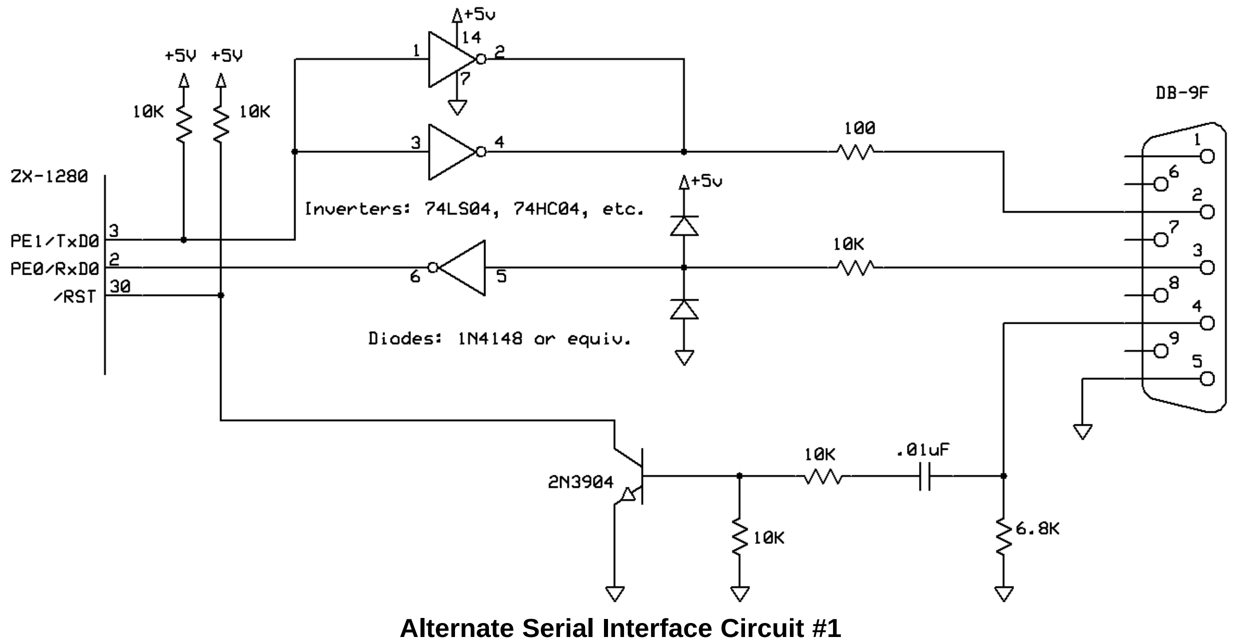
One advantage to using this recommended circuit is that the MAX232 chip includes a DC-DC converter that produces nominally +/- 12V from the 5 volt source. This is particularly useful if your application requires one or more of these voltages. For example, some LCD devices require a low-current negative supply for their backlight circuitry. The positive voltage is available on pin 2 of the Max232 while the negative voltage is available on pin 6. Consult the datasheet for the MAX232 device for information on the current capacity of these supplies.



Recommended ZX-1280 Series Serial Interface Circuit

The pullup resistor shown on the ZX's Com1 transmit output (pin 28) is needed to ensure that the serial output line stays in the idle state during reset cycles. Its value is non-critical – anything from 4.7K to 22K should work fine. The pullup resistor on the reset input isn't technically required since the chip itself has an internal pullup. The external pullup could be eliminated or could be made larger if desired.

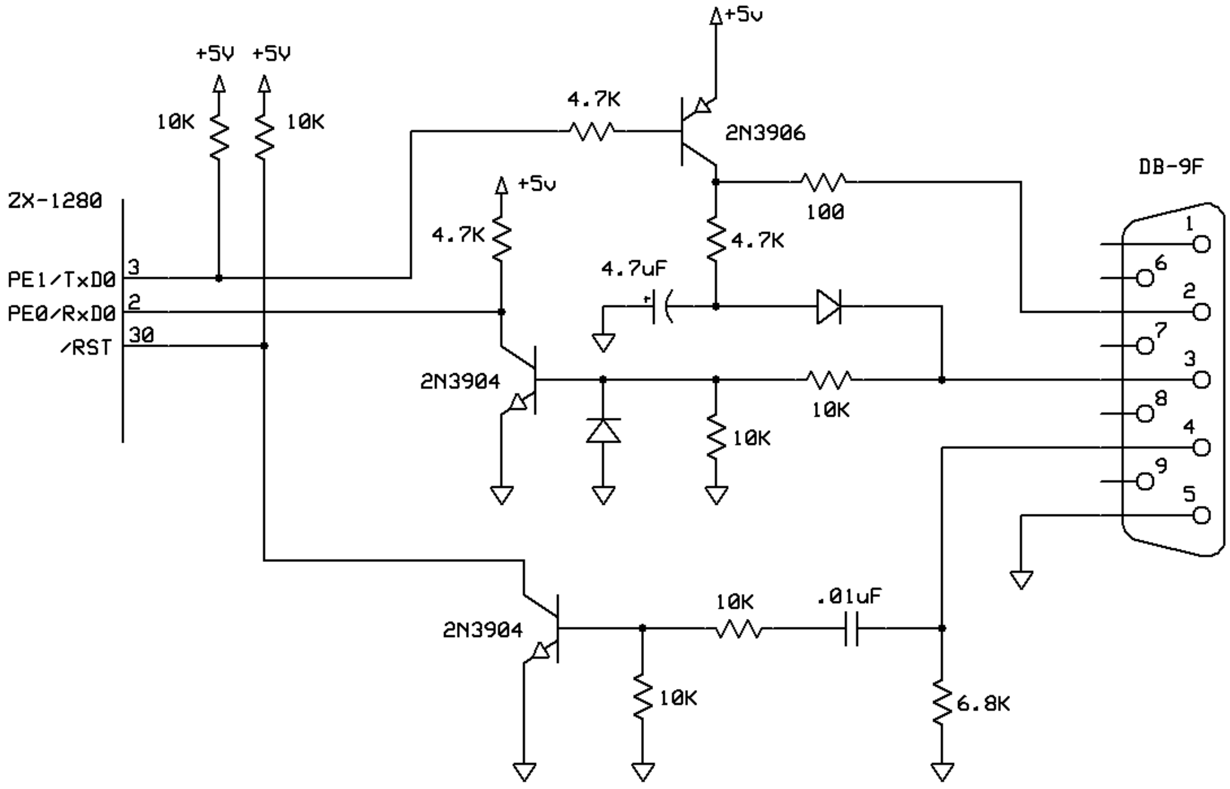
There are other serial interface circuits that would provide the required functionality, two of which are shown below. Both alternate circuits use the same ATN circuitry as the recommended circuit above.



Alternate Serial Interface Circuit #1

The first alternate above is similar to the circuitry used on the ZX-24 series devices. The advantage to this circuit is its simplicity. The disadvantage is that the voltage levels of the serial output signal do not meet the RS-232 standard. Even so, most serial receivers are able to properly interpret the output signal. Note that if the inverters used have protection diodes on their inputs the external diodes shown are not necessary.

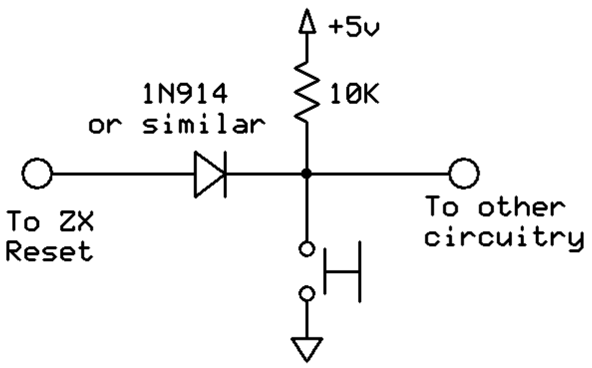
The second alternate serial interface is shown in the diagram below. This circuit produces a serial output signal that meets the RS-232 standard by using the negative voltage on the receive line to provide a reference for the output level. Although the parts count is higher, the circuit uses inexpensive and non-critical components.



Alternate Serial Interface Circuit #2

Reset Circuit

You may also want to incorporate a reset button in your circuitry. If so, a normally open, momentary contact switch may be connected directly between pin 20 of the ZX and ground. If other circuitry in your application also needs the reset signal, you may want to isolate that circuitry from the ATN resets using a buffer gate or a diode. A suggested diode isolation circuit for the reset line is shown below.



Reset Signal Isolation