

# ZBasic System Library Reference Manual

Version 2.2.2

## Publication History

|               |   |
|---------------|---|
| November 2005 | First publication   |
| December 2005 | Added new routine descriptions, minor corrections             |
| January 2006  | Added new routine descriptions, minor corrections             |
| February 2006 | Minor corrections   |
| May 2006      | Added new routine descriptions, minor corrections             |
| August 2006   | Added new routine descriptions, minor corrections             |
| October 2006  | Added new routine descriptions, minor corrections             |
| January 2007  | Added new routine descriptions                                |
| February 2007 | Added information on new ZX models                            |
| August 2007   | Updated for a new ZX model and added new routine descriptions |
| March 2008    | Updated for new ZX models and added new routine descriptions  |
| April 2008    | Added new routine descriptions                                |

## Disclaimer

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

## Trademarks

ZBasic, ZX-24, ZX-24a, ZX-24p, ZX-24n, ZX-40, ZX-40a, ZX-40p, ZX-40n, ZX-44, ZX-44a, ZX-44p, ZX-44n, ZX-1280, ZX-1280n, ZX-1281 and ZX-1281n are trademarks of Elba Corp.

ZX-24e, ZX-24ae, ZX-24pe, ZX-24ne, ZX-128e, ZX-128ne, ZX-1281e and ZX-1281ne are trademarks of Oak Micros used under license from Elba Corp.

AVR is a registered trademark of Atmel Corp.

BasicX, BX-24 and BX-35 are trademarks of NetMedia, Inc.

PBasic is a trademark and Basic Stamp is a registered trademark of Parallax, Inc.

Visual Basic is a registered trademark of Microsoft Corp.

Other brand and product names are trademarks or registered trademarks of their respective owners.

## Table of Contents

|                                     |    |
|-------------------------------------|----|
| Routines by Category .....          | 1  |
| Type Conversion Functions .....     | 1  |
| Mathematical Functions .....        | 1  |
| Memory-related Routines .....       | 2  |
| String-related Routines .....       | 2  |
| Input/Output Routines .....         | 3  |
| Serial Communication Routines ..... | 4  |
| Queue Management Routines .....     | 4  |
| Date/Time Routines .....            | 4  |
| Data Manipulation Routines .....    | 4  |
| Task-related Routines .....         | 5  |
| Miscellaneous Routines .....        | 5  |
| Resource Usage .....                | 3  |
| USART .....                         | 3  |
| SPI Interface .....                 | 3  |
| Analog-to-Digital Converters .....  | 4  |
| Interrupts .....                    | 4  |
| Interrupt Service Routines .....    | 5  |
| Timers .....                        | 6  |
| Detailed Descriptions .....         | 10 |
| Abs .....                           | 11 |
| Acos .....                          | 12 |
| ADCtoCom1 .....                     | 13 |
| Asc .....                           | 14 |
| Asin .....                          | 15 |
| Atn .....                           | 16 |
| Atn2 .....                          | 17 |
| BitCopy .....                       | 18 |
| BlockMove .....                     | 19 |
| BusRead .....                       | 20 |
| BusWrite .....                      | 21 |
| CallTask .....                      | 22 |
| CBit .....                          | 24 |
| CBool .....                         | 25 |
| CByte .....                         | 26 |
| CByteArray .....                    | 27 |
| Ceiling .....                       | 28 |
| Chr .....                           | 29 |
| CInt .....                          | 30 |
| ClearQueue .....                    | 31 |

|                           |    |
|---------------------------|----|
| CLng .....                | 32 |
| CloseCom .....            | 33 |
| Closel2C .....            | 34 |
| ClosePWM.....             | 35 |
| CloseSPI .....            | 36 |
| CloseWatchDog.....        | 37 |
| CloseX10 .....            | 38 |
| CNibble.....              | 39 |
| Com1toDAC .....           | 40 |
| ComChannels .....         | 41 |
| Console.Read .....        | 43 |
| Console.ReadLine.....     | 44 |
| Console.Write.....        | 45 |
| Console.WriteLine .....   | 46 |
| Cos.....                  | 47 |
| CountTransitions.....     | 48 |
| CPUSleep .....            | 49 |
| CRC16.....                | 50 |
| CRC32.....                | 52 |
| CSng.....                 | 53 |
| CStr.....                 | 54 |
| CStrHex.....              | 55 |
| CType .....               | 56 |
| CUInt.....                | 57 |
| CULng .....               | 58 |
| DACPin.....               | 59 |
| Debug.Print.....          | 60 |
| DefineBus .....           | 61 |
| DefineCom.....            | 62 |
| DefineCom3.....           | 64 |
| DefineX10 .....           | 65 |
| DegToRad .....            | 66 |
| Delay.....                | 67 |
| DelayUntilClockTick ..... | 68 |
| DisableInt .....          | 69 |
| EnableInt.....            | 70 |
| ExitTask .....            | 71 |
| Exp .....                 | 72 |
| Exp10 .....               | 73 |
| FirstTime .....           | 74 |
| Fix.....                  | 75 |
| FixB.....                 | 76 |

|                                |     |
|--------------------------------|-----|
| FixI .....                     | 77  |
| FixL .....                     | 78  |
| FixUI .....                    | 79  |
| FixUL .....                    | 80  |
| FlipBits.....                  | 81  |
| Floor.....                     | 82  |
| Fmt.....                       | 83  |
| Fraction .....                 | 84  |
| FreqOut .....                  | 85  |
| Get1Wire.....                  | 87  |
| Get1WireByte.....              | 88  |
| Get1WireData .....             | 89  |
| GetADC (subroutine form) ..... | 90  |
| GetADC (function form) .....   | 91  |
| GetBit.....                    | 92  |
| GetDate.....                   | 93  |
| GetDayNumber .....             | 94  |
| GetDayOfWeek.....              | 95  |
| GetDayOfYear.....              | 96  |
| GetEEPROM.....                 | 97  |
| GetNibble .....                | 98  |
| GetPersistent.....             | 99  |
| GetPin.....                    | 100 |
| GetProgMem .....               | 101 |
| GetQueue .....                 | 102 |
| GetQueueBufferSize.....        | 104 |
| GetQueueCount.....             | 105 |
| GetQueueSpace .....            | 106 |
| GetQueueStr .....              | 107 |
| GetTime .....                  | 108 |
| GetTimestamp.....              | 109 |
| HiByte .....                   | 110 |
| HiWord .....                   | 111 |
| I2CCmd .....                   | 112 |
| I2CGetByte .....               | 114 |
| I2CPutByte.....                | 115 |
| I2CStart .....                 | 116 |
| I2CStop .....                  | 117 |
| IIf .....                      | 118 |
| InputCapture.....              | 119 |
| InputCaptureEx.....            | 120 |
| LBound .....                   | 122 |

|                                |     |
|--------------------------------|-----|
| LCase .....                    | 123 |
| Left .....                     | 124 |
| Len .....                      | 125 |
| LoByte .....                   | 126 |
| LockTask.....                  | 127 |
| Log .....                      | 128 |
| Log10.....                     | 129 |
| LongJump .....                 | 130 |
| LoWord.....                    | 131 |
| MakeDword .....                | 132 |
| MakeString.....                | 133 |
| MakeWord .....                 | 134 |
| Max .....                      | 135 |
| MemAddress .....               | 136 |
| MemAddressU .....              | 137 |
| MemCmp .....                   | 138 |
| MemCopy.....                   | 139 |
| MemSet.....                    | 140 |
| Mid .....                      | 141 |
| MidWord.....                   | 142 |
| Min .....                      | 143 |
| NoOp .....                     | 144 |
| OpenCom.....                   | 145 |
| OpenI2C.....                   | 147 |
| OpenI2CSlave.....              | 149 |
| OpenPWM .....                  | 150 |
| OpenQueue .....                | 151 |
| OpenSPI.....                   | 152 |
| OpenSPISlave.....              | 154 |
| OpenWatchDog .....             | 155 |
| OpenX10.....                   | 156 |
| OutputCapture.....             | 158 |
| OutputCaptureEx.....           | 160 |
| ParityCheck .....              | 162 |
| Pause .....                    | 163 |
| PeekQueue.....                 | 164 |
| PersistentPeek.....            | 165 |
| PersistentPoke.....            | 166 |
| PlaySound .....                | 167 |
| PortBit.....                   | 169 |
| Pow .....                      | 170 |
| PulseIn (subroutine form)..... | 171 |

|                               |     |
|-------------------------------|-----|
| PulseIn (function form).....  | 172 |
| PulseOut .....                | 173 |
| Put1Wire .....                | 174 |
| Put1WireByte .....            | 175 |
| Put1WireData.....             | 176 |
| PutBit .....                  | 177 |
| PutDAC .....                  | 178 |
| PutDate .....                 | 180 |
| PutEEPROM .....               | 181 |
| PutNibble .....               | 182 |
| PutPersistent .....           | 183 |
| PutPin .....                  | 184 |
| PutProgMem .....              | 185 |
| PutQueue.....                 | 186 |
| PutQueueByte.....             | 187 |
| PutQueueStr.....              | 188 |
| PutTime.....                  | 189 |
| PutTimeStamp .....            | 190 |
| PWM.....                      | 191 |
| RadToDeg .....                | 193 |
| RamPeek .....                 | 194 |
| RamPeekDword.....             | 195 |
| RamPeekWord.....              | 196 |
| RamPoke .....                 | 197 |
| RamPokeDword.....             | 198 |
| RamPokeWord.....              | 199 |
| Randomize.....                | 200 |
| RCTime (subroutine form)..... | 201 |
| RCTime (function form) .....  | 202 |
| Reset1Wire.....               | 203 |
| ResetProcessor .....          | 204 |
| ResumeTask .....              | 205 |
| Right .....                   | 206 |
| Rnd .....                     | 207 |
| RunTask.....                  | 208 |
| Semaphore .....               | 209 |
| SerialNumber .....            | 210 |
| SetBits .....                 | 211 |
| SetInterval.....              | 212 |
| SetJmp .....                  | 213 |
| ShiftIn .....                 | 214 |
| ShiftInEx.....                | 215 |

|                           |     |
|---------------------------|-----|
| ShiftOut .....            | 217 |
| ShiftOutEx.....           | 218 |
| Shl .....                 | 220 |
| Shr.....                  | 221 |
| Signum .....              | 222 |
| Sin .....                 | 223 |
| SizeOf.....               | 224 |
| SizeOfU .....             | 225 |
| Sleep.....                | 226 |
| SngClass.....             | 227 |
| SPICmd.....               | 228 |
| Sqr.....                  | 230 |
| StackCheck .....          | 231 |
| StatusCom .....           | 232 |
| StatusQueue .....         | 233 |
| StatusTask.....           | 234 |
| StatusX10 .....           | 235 |
| StrAddress .....          | 236 |
| StrCompare .....          | 237 |
| StrFind.....              | 238 |
| StrReplace .....          | 239 |
| StrType.....              | 240 |
| System.Alloc.....         | 241 |
| System.DeviceID .....     | 242 |
| System.Free .....         | 243 |
| System.HeapHeadRoom.....  | 244 |
| System.HeapSize .....     | 245 |
| System.TaskHeadRoom ..... | 246 |
| Tan.....                  | 247 |
| TaskIsLocked .....        | 248 |
| TaskIsValid .....         | 249 |
| Timer.....                | 250 |
| To<enum> .....            | 251 |
| ToggleBits.....           | 252 |
| Trim.....                 | 253 |
| UBound .....              | 254 |
| UCase.....                | 255 |
| UnlockTask.....           | 256 |
| UpdateRTC.....            | 257 |
| ValueI .....              | 258 |
| ValueL .....              | 259 |
| ValueS.....               | 260 |



|                       |     |
|-----------------------|-----|
| VarPtr .....          | 261 |
| WaitForInterrupt..... | 262 |
| WaitForInterval.....  | 266 |
| WatchDog .....        | 268 |
| X10Cmd .....          | 269 |
| Yield.....            | 271 |
| ZXCmdMode .....       | 272 |

# System Library Reference

## Routines by Category

The ZBasic System Library provides a rich collection of subroutines and functions that you can use to add functionality to your application. The routines may be divided into several conceptual categories as shown below.

### Type Conversion Functions

|              |   |
|--------------|---|
| CBit()       | convert a value to type Bit                                   |
| CBool()      | convert a value to type Boolean                               |
| CByte()      | convert a value to type Byte                                  |
| CByteArray() | convert an integral value to a reference to a Byte array      |
| CInt()       | convert a value to type Integer                               |
| CLng()       | convert a value to type Long                                  |
| CNibble()    | convert a value to type Nibble                                |
| CSng()       | convert a value to type Single                                |
| CStr()       | convert a value to type String                                |
| CStrHex()    | convert a value to a String containing hexadecimal characters |
| CType()      | convert a value to an enumeration member                      |
| CUInt()      | convert a value to type UnsignedInteger                       |
| CULng()      | convert a value to type Long                                  |
| FixB()       | convert a Single value to type Byte                           |
| FixI()       | convert a Single value to type Integer                        |
| FixL()       | convert a Single value to type Long                           |
| FixUI()      | convert a Single value to type UnsignedInteger                |
| FixUL()      | convert a Single value to type UnsignedLong                   |
| To<enum>()   | convert a value to an enumeration member                      |

### Mathematical Functions

|            |  |
|------------|--|
| Abs()      | absolute value                                     |
| ACos()     | arc cosine   |
| ASin()     | arc sine   |
| Atn()      | arc tangent  |
| Atn2()     | arc tangent (quadrant-correct)                     |
| Ceiling()  | largest integer not greater than a Single value    |
| Cos()      | cosine   |
| DegToRad() | convert degrees to radians                         |
| Exp()      | $e^x$  |
| Exp10()    | $10^x$   |
| Fix()      | integer portion of a Single value                  |
| Floor()    | smallest integer not less than a Single value      |
| Fraction() | fractional portion of a Single value               |
| Log()      | natural logarithm                                  |
| Log10()    | common logarithm                                   |
| Max()      | determine the largest of two values                |
| Min()      | determine the smallest of two values               |
| Pow()      | raise a value to a power                           |
| RadToDeg() | convert radians to degrees                         |
| Signum()   | determine if a value is negative, zero or positive |
| Sin()      | sine   |
| SngClass() | return the class information for a Single value    |
| Sqr()      | square root  |
| Tan()      | tangent  |

## Memory-related Routines

|                                    |   |
|------------------------------------|---|
| <code>BitCopy()</code>             | copy a sequence of bits from one part of RAM to another |
| <code>BlockMove()</code>           | copy data from one part of RAM to another               |
| <code>GetBit()</code>              | extract a bit from a value in RAM                       |
| <code>GetEEPROM()</code>           | copy data from Program Memory to RAM                    |
| <code>GetPersistent()</code>       | copy data from Persistent Memory to RAM                 |
| <code>GetProgMem()</code>          | copy data from Program Memory to RAM                    |
| <code>MemAddress()</code>          | determine the RAM address of a variable                 |
| <code>MemAddressU()</code>         | determine the RAM address of a variable                 |
| <code>MemCmp()</code>              | compare two blocks of data in RAM                       |
| <code>MemCopy()</code>             | copy data from one part of RAM to another               |
| <code>MemSet()</code>              | initialize a block of memory with a byte value          |
| <code>PersistentPeek()</code>      | read a byte from Persistent Memory                      |
| <code>PersistentPoke()</code>      | write a byte to Persistent Memory                       |
| <code>PutBit()</code>              | set or clear a bit in a value in RAM                    |
| <code>PutEEPROM()</code>           | copy data from RAM to Program Memory                    |
| <code>PutPersistent()</code>       | copy data from RAM to Persistent Memory                 |
| <code>PutProgMem()</code>          | copy data from RAM to Program Memory                    |
| <code>RamPeek()</code>             | read a byte from RAM                                    |
| <code>RamPeekDword()</code>        | read a 32-bit value from RAM                            |
| <code>RamPeekWord()</code>         | read a 16-bit value from RAM                            |
| <code>RamPoke()</code>             | write a byte to RAM                                     |
| <code>RamPokeDword()</code>        | write a 32-bit value to RAM                             |
| <code>RamPokeWord()</code>         | write a 16-bit value to RAM                             |
| <code>System.Alloc()</code>        | allocate a block of memory                              |
| <code>System.Free()</code>         | deallocate a block of memory                            |
| <code>System.HeapHeadRoom()</code> | determine the amount of unused space in the heap        |
| <code>System.HeapSize()</code>     | determine the amount of space reserved for the heap     |
| <code>VarPtr()</code>              | determine the RAM address of a variable                 |

## String-related Routines

|                           |  |
|---------------------------|--|
| <code>Asc()</code>        | extract a character value from a string                                |
| <code>Chr()</code>        | convert a character value to a string                                  |
| <code>Fmt()</code>        | convert a <code>Single</code> value to a string                        |
| <code>LCase()</code>      | convert upper case letters to lower case in a string                   |
| <code>Left()</code>       | return the leftmost characters from a string                           |
| <code>Len()</code>        | determine the number of characters in a string                         |
| <code>Mid()</code>        | extract or set a substring in a string                                 |
| <code>Right()</code>      | return the rightmost characters from a string                          |
| <code>StrAddress()</code> | determine the address where string characters are stored               |
| <code>StrCompare()</code> | compare two strings, optionally ignoring alphabetic case               |
| <code>StrFind()</code>    | search for the first occurrence of a string within a string            |
| <code>StrReplace()</code> | replace character sequences in a string                                |
| <code>StrType()</code>    | determine the characteristics of a string                              |
| <code>Trim()</code>       | remove leading and trailing spaces from a string                       |
| <code>UCase()</code>      | convert lower case letters to upper case in a string                   |
| <code>ValueI()</code>     | convert string characters to the equivalent <code>Integer</code> value |
| <code>ValueL()</code>     | convert string characters to the equivalent <code>Long</code> value    |
| <code>ValueS()</code>     | convert string characters to the equivalent <code>Single</code> value  |

## Input/Output Routines

|                    |   |
|--------------------|---|
| ADCToCom1()        | stream analog conversion data to Com1                       |
| BusRead()          | read data from a bus-oriented device                        |
| BusWrite()         | write data to a bus-oriented device                         |
| CloseI2C()         | deinitialize an I2C communication channel                   |
| ClosePWM()         | deinitialize an PWM channel                                 |
| CloseSPI()         | deinitialize an SPI2 communication channel                  |
| CloseX10()         | deinitialize an X-10 communication channel                  |
| Com1toDAC()        | receive stream of analog conversion data                    |
| CountTransitions() | count transitions on an input pin                           |
| DACPin()           | produce an analog voltage on an output pin                  |
| DefineBus()        | specify the parameters for accessing a bus-oriented device  |
| DefineX10()        | specify the communication parameters for an X-10 channel    |
| FreqOut()          | produce a dual-frequency sine wave on an output pin         |
| Get1Wire()         | receive a bit using the 1-Wire protocol                     |
| Get1WireByte()     | receive a byte using the 1-Wire protocol                    |
| Get1WireData()     | receive one or more bytes using the 1-Wire protocol         |
| GetADC()           | perform an analog to digital conversion on an input         |
| GetPin()           | read the state of an input pin                              |
| I2CCmd()           | send/receive data over an I2C channel                       |
| I2CGetByte()       | receive a byte on an I2C channel                            |
| I2CPutByte()       | send a byte on an I2C channel                               |
| I2CStart()         | create a Start condition on an I2C channel                  |
| I2CStop()          | create a Stop condition on an I2C channel                   |
| InputCapture()     | record the high/low times of a pulse train on an input pin  |
| InputCaptureEx()   | record the high/low times of a pulse train on an input pin  |
| OpenI2C()          | prepare for I2C communication with an external device       |
| OpenI2CSlave()     | activate I2C slave mode                                     |
| OpenSPI()          | prepare for SPI communication with an external device       |
| OpenSPISlave()     | activate SPI slave mode                                     |
| OpenPWM()          | prepare for PWM generation                                  |
| OpenX10()          | prepare an X-10 communication channel for use               |
| OutputCapture()    | produce a pulse train                                       |
| OutputCaptureEx()  | produce a pulse train on any output pin                     |
| PlaySound()        | reproduce sampled audio on an output pin                    |
| PortBit()          | compose a designator for a specific bit in an I/O port      |
| PulseIn()          | measure a pulse width on an input pin                       |
| PulseOut()         | generate a pulse on an output pin                           |
| Put1Wire()         | send a bit using the 1-Wire protocol                        |
| Put1WireByte()     | send a byte using the 1-Wire protocol                       |
| Put1WireData()     | send one or more bytes using the 1-Wire protocol            |
| PutDAC()           | produce an analog voltage on an output pin                  |
| PutPin()           | configure an I/O pin  |
| PWM()              | initiate PWM generation or change the duty cycle            |
| RCTime()           | measure an RC charge/discharge time                         |
| Reset1Wire()       | send a reset signal using the 1-Wire protocol               |
| ShiftIn()          | perform synchronous serial input                            |
| ShiftInEx()        | perform synchronous serial input with more configurability  |
| ShiftOut()         | perform synchronous serial output                           |
| ShiftOutEx()       | perform synchronous serial output with more configurability |
| SPICmd()           | perform SPI communication with an external device           |
| StatusX10()        | determine the status of an X-10 communication channel       |
| X10Cmd()           | send commands using the X-10 protocol                       |

## Serial Communication Routines

|                                  |   |
|----------------------------------|---|
| <code>Debug.Print</code>         | send strings to Com1 via the system output queue          |
| <code>CloseCom()</code>          | terminate the use of a serial channel                     |
| <code>ComChannels()</code>       | prepare for using multiple serial channels                |
| <code>Console.Read()</code>      | retrieve a character from Com1 via the system input queue |
| <code>Console.ReadLine()</code>  | retrieve a line from Com1 via the system input queue      |
| <code>Console.Write()</code>     | send a string to Com1 via the system output queue         |
| <code>Console.WriteLine()</code> | send a string to Com1 via the system output queue         |
| <code>DefineCom()</code>         | set the characteristics of a serial channel               |
| <code>DefineCom3()</code>        | set the characteristics of serial channel 3               |
| <code>OpenCom()</code>           | prepare a serial channel for use                          |
| <code>StatusCom()</code>         | determine the status of a serial channel                  |

## Queue Management Routines

|                                   |  |
|-----------------------------------|--|
| <code>ClearQueue()</code>         | delete data from a queue                           |
| <code>GetQueue()</code>           | retrieve data from a queue                         |
| <code>GetQueueBufferSize()</code> | determine the size of the data area of a queue     |
| <code>GetQueueCount()</code>      | determine the number of bytes of data in a queue   |
| <code>GetQueueSpace()</code>      | determine the amount of space available in a queue |
| <code>GetQueueStr()</code>        | populate a string with characters from a queue     |
| <code>OpenQueue()</code>          | prepare a queue for use                            |
| <code>PeekQueue()</code>          | copy data from a queue without removing it         |
| <code>PutQueue()</code>           | put data in a queue                                |
| <code>PutQueueByte()</code>       | put a byte into a queue                            |
| <code>PutQueueStr()</code>        | put the characters of a string in a queue          |
| <code>StatusQueue()</code>        | determine if a queue has data available            |

## Date/Time Routines

|                             |   |
|-----------------------------|---|
| <code>GetDate()</code>      | get the month, day, year corresponding to a date value        |
| <code>GetDayNumber()</code> | compute the day number corresponding to a day of a year       |
| <code>GetDayOfWeek()</code> | get the day of the week corresponding to a date value         |
| <code>GetDayOfYear()</code> | get the ordinal day of the year corresponding to a date value |
| <code>GetTime()</code>      | get the current hour, minute and second                       |
| <code>GetTimestamp()</code> | get the current date and time information                     |
| <code>PutDate()</code>      | set the current month, day, year                              |
| <code>PutTime()</code>      | set the current hour, minute and second                       |
| <code>PutTimeStamp()</code> | set the current date and time information                     |
| <code>Timer()</code>        | get the current clock tick value                              |

## Data Manipulation Routines

|                           |   |
|---------------------------|---|
| <code>FlipBits()</code>   | reverse the order of bits in a byte             |
| <code>HiByte()</code>     | extract the high byte of a value                |
| <code>HiWord()</code>     | extract the high word of a value                |
| <code>LoByte()</code>     | extract the low byte of a value                 |
| <code>LoWord()</code>     | extract the low word of a value                 |
| <code>MakeDword()</code>  | construct a 32-bit value from two 16-bit values |
| <code>MakeWord()</code>   | construct a 16-bit value from two 8-bit values  |
| <code>MakeString()</code> | construct a string from a sequence of bytes     |
| <code>MidWord()</code>    | extract the middle two bytes of a 4-byte value  |
| <code>SetBits()</code>    | set the state of specified bits in a byte       |
| <code>Shl()</code>        | shift a value to the left                       |
| <code>Shr()</code>        | shift a value to the right                      |
| <code>ToggleBits()</code> | change the state of specified bits in a byte    |

## Task-related Routines

|                       |  |
|-----------------------|--|
| CallTask              | prepare a task to begin execution                |
| DisableInt()          | disable interrupts                               |
| Delay()               | pause a task                                     |
| DelayUntilClockTick() | pause a task                                     |
| EnableInt()           | conditionally re-enable interrupts               |
| ExitTask()            | cause a task to terminate                        |
| LockTask()            | suspend normal task switching                    |
| Pause()               | pause a task without relinquishing control       |
| ResumeTask()          | cause a waiting task to resume execution         |
| RunTask()             | cause a specific task to run                     |
| Semaphore()           | coordinate the use of a resource                 |
| SetInterval()         | set the interval timer period                    |
| Sleep()               | pause a task                                     |
| StackCheck()          | enable or disable stack checking                 |
| StatusTask()          | determine the status of a task                   |
| System.TaskHeadRoom() | determine the unused space in a task's stack     |
| TaskIsLocked()        | determine if a task is locked                    |
| TaskIsValid()         | determine if a task stack is in the task list    |
| UnlockTask()          | resume normal task switching                     |
| UpdateRTC()           | update RTC registers to account for missed ticks |
| WaitForInterrupt()    | pause a task until an external event occurs      |
| WaitForInterval()     | pause a task until an interval timer expires     |
| Yield()               | allow another task to run                        |

## Miscellaneous Routines

|                   |   |
|-------------------|---|
| CloseWatchDog()   | deactivate the watchdog timer   |
| CPUSleep()        | cause the CPU to go into sleep mode                                       |
| CRC16()           | compute a 16-bit CRC value  |
| CRC32()           | compute a 32-bit CRC value  |
| FirstTime()       | determine if this is the first the program has been run since downloading |
| IIf()             | select the value of one of two expressions                                |
| LBound()          | determine the lower bound of an array                                     |
| LongJump()        | perform a non-local goto (e.g. for exception handling)                    |
| NoOp()            | execute a "nop" instruction   |
| OpenWatchDog()    | activate the watchdog timer   |
| ParityCheck()     | check the parity of a data byte   |
| Randomize()       | initialize the random number generator                                    |
| ResetProcessor()  | reset the CPU   |
| Rnd()             | retrieve the next random number   |
| SerialNumber()    | retrieve the system software serial number                                |
| SetJump()         | prepare for a non-local Goto (e.g. exception handling)                    |
| SizeOf()          | determine the size of a data item   |
| SizeOfU()         | determine the size of a data item   |
| System.DeviceID() | retrieve the identification characters for the device                     |
| UBound()          | determine the upper bound of an array                                     |
| WatchDog()        | reset the watchdog timer  |
| ZXCmdMode()       | activate the "command mode" (for downloading)                             |







## Resource Usage

The AVR devices on which the ZX microcontrollers are based offer a variety of resources for use in your program, e.g. timers, interrupts, USART (hardware serial port), analog-to-digital converters, etc. Some of these resources are allocated to specific functions of the ZX microcontroller and/or are used by certain ZBasic System Library routines. The resources available on a particular ZX device vary depending on the particular CPU upon which the device is based. The table below indicates the underlying CPU for the various ZX devices. The remainder of this section provides an overview of resource allocation for ZX devices.

**Underlying CPU Type for ZX Devices**

| <b>ZX Model</b>  | <b>CPU Type</b> |
|--|-----------------|
| ZX-24, ZX-40, ZX-44, ZX-24e                                      | mega32          |
| ZX-24a, ZX-40a, ZX-44a, ZX-24ae                                  | mega644         |
| ZX-24p, ZX-40p, ZX-44p, ZX-24n, ZX-40n, ZX-44n, ZX-24pe, ZX-24ne | mega644P        |
| ZX-1280, ZX-1280n  | mega1280        |
| ZX-1281, ZX-1281n, ZX-1281e, ZX-1281ne                           | mega1281        |
| ZX-128e, ZX-128ne  | mega128         |

## USART

An on-board hardware serial port, or USART, is used for the Com1 serial channel. By default, the USART is configured to operate at 19,200 baud and is utilized by the System Library Routines Console.Read, Console.ReadLine, Console.Write, Console.WriteLine and Debug.Print. You may reconfigure the USART to a different speed by using the System Library routine OpenCom, specifying channel 1. The USART is also used for the ADCtoCom1 and Com1toDAC routines. In both of these cases, the Com1 speed is automatically configured.

Some ZX devices have more than one hardware USART. In these cases, one of the USARTs is assigned to the Com1 serial channel, a second USART is assigned to the Com2 serial channel, etc. as shown in the table below. The effect of these assignments is generally only important with respect to which I/O pins are available for other purposes if the additional hardware USARTs are not being used. It also will be important if your program manipulates the USART registers directly.

**Hardware USART Channel Assignment and I/O Pin Usage**

| <b>ZX Model</b>                        | <b>USART</b> | <b>Serial Channel</b> | <b>Tx Pin</b> | <b>Rx Pin</b> |
|--|--------------|-----------------------|---------------|---------------|
| ZX-24p, ZX-24n, ZX-24pe, ZX-24ne       | USART0       | Com1                  | 1, D.1        | 2, D.0        |
|  | USART1       | Com2                  | 11, D.3       | 6, D.2        |
| ZX-40p, ZX-40n                         | USART0       | Com1                  | 15, D.1       | 14, D.0       |
|  | USART1       | Com2                  | 17, D.3       | 16, D.2       |
| ZX-44p, ZX-44n                         | USART0       | Com1                  | 10, D.1       | 9, D.0        |
|  | USART1       | Com2                  | 12, D.3       | 11, D.2       |
| ZX-1281                                | USART1       | Com1                  | 28, D.3       | 27, D.2       |
|  | USART0       | Com2                  | 3, E.1        | 2, E.0        |
| ZX-1280                                | USART0       | Com1                  | 3, E.1        | 2, E.0        |
|  | USART1       | Com2                  | 46, D.3       | 45, D.2       |
|  | USART2       | Com7                  | 13, H.1       | 12, H.0       |
|  | USART3       | Com8                  | 64, J.1       | 63, J.0       |
| ZX-128e, ZX-128ne, ZX-1281e, ZX-1281ne | USART0       | Com1                  | 19, E.1       | 20, E.0       |
|  | USART1       | Com2                  | 9, D.3        | 10, D.2       |

## SPI Interface

On some ZX devices, your program is stored in an external EEPROM that is read and written using the SPI interface. A dedicated I/O pin is required for selecting the EEPROM device during SPI operations and this I/O pin cannot be used for other purposes. However, the SPI bus itself can be used to

communicate with other SPI devices. Although most SPI devices are tolerant of the ZX device using the SPI bus to fetch instructions from your program, a few are not. Generally speaking, if you can send and receive all of the data that an SPI device requires using a single call to `SPICmd()`, then that SPI device is usable with the ZX models that utilize an external EEPROM. The table below indicates which devices use an external EEPROM for user programs and, if so, the I/O pin used for the chip select.

| SPI Channel and EEPROM Usage              |                 |            |
|---|-----------------|------------|
| ZX Model                                  | Uses SPI EEPROM | SPI CS Pin |
| ZX-24, ZX-24a, ZX-24p                     | Yes             | B.4        |
| ZX-40, ZX-40a, ZX-40p                     | Yes             | 5, B.4     |
| ZX-44, ZX-44a, ZX-44p                     | Yes             | 44, B.4    |
| ZX-24e, ZX-24ae, ZX-24pe                  | Yes             | 24, B.4    |
| ZX-24n, ZX-40n, ZX-44n                    | No              | B.4        |
| ZX-24ne                                   | No              | 24, B.4    |
| ZX-1281, ZX-1281n                         | No              | 10, B.0    |
| ZX-1280, ZX-1280n                         | No              | 19, B.0    |
| ZX-128e, ZX-128ne,<br>ZX-1281e, ZX-1281ne | No              | 28, B.0    |

It is important to note that even for the devices that do not use the external SPI EEPROM, the SPI CS pin cannot be used as a general input if the SPI bus is used in your application. This restriction is an artifact of the design of the CPU's SPI controller. The SPI CS pin can, however, be used as a general purpose output.

## Analog-to-Digital Converters

Most ZX devices support up to 8 analog inputs. These inputs may be fed to the internal analog-to-digital converter (ADC) or they may be used to perform analog level comparisons. The I/O port containing the analog inputs varies by ZX device as indicated in the table below. The System Library routines `GetADC()` and `ADCtoCom1()` use the ADC. The analog comparator is used by `WaitForInterrupt()` when configured to await an analog comparator event.

| Analog Input Ports by CPU Type |               |               |
|--------------------------------|---------------|---------------|
| Underlying CPU Type            | Analog Port 1 | Analog Port 2 |
| mega32, mega644, mega644P      | Port A        | -             |
| mega128, mega1281              | Port F        | -             |
| mega1280                       | Port F        | Port K        |

## Interrupts

Some of the System Library routines disable interrupts in order to achieve the precise timing that is required. Having interrupts disabled for long periods of time can interfere with the operation of other parts of the system that use interrupts like task management, serial I/O and the real time clock. In most cases, the System Library routines have been implemented to keep track of real time clock interrupts that should have occurred during the time interrupts are disabled and then the RTC is updated at the end of the operation. This strategy avoids the problem of the RTC losing time.

Unfortunately, there is no way to similarly protect the serial I/O process. You can reduce the impact of having interrupts disabled with respect to serial output by ensuring that all serial output queues are empty before calling a System Library routine that disables interrupts. This is not as critical for a hardware-based serial channel (e.g. Com1) as it is for the software-based serial channels Com3 to Com6. There is no way, however, to work around the problem of serial input data arriving while interrupts are disabled. The hardware-based serial channels will store one received character and hold it while interrupts are disabled but if a second character arrives while interrupts are disabled it will be lost. Channels 3-6 rely on interrupts for every bit received so the situation is much more problematic. In this case, having interrupts disabled for longer than approximately one-third of the bit time will likely cause garbled input if a character's transmit time overlaps the period when interrupts are disabled. For characters being

transmitted by channels 3-6, having interrupts disabled for more than about 10% of the bit time may cause the receiver to lose synchronization.

For reference purposes, the table below indicates which I/O routines disable interrupts for the duration of their execution. See the individual descriptions for more detailed information.

### System Library Routines that Disable Interrupts

|                  |             |            |
|------------------|-------------|------------|
| CountTransitions | I2CPutByte  | RCTime     |
| DACPin           | PlaySound   | ResetWire  |
| FreqOut          | PulseIn     | ShiftIn    |
| GetWire          | PulseOut    | ShiftInEx  |
| GetWireByte      | PutWire     | ShiftOut   |
| GetWireData      | PutWireByte | ShiftOutEx |
| I2CCmd           | PutWireData |            |
| I2CGetByte       | PutDAC      |            |

## Interrupt Service Routines

For the native code devices (e.g. ZX-24n), a few interrupt service routines (ISRs) are always included in your program (e.g. for Com1 and the RTC) while others are included only if certain System Library routines are used in your program. In some cases, the additional ISRs that are included when a specific System Library routine is used depends on how the routine is invoked and what the compiler can deduce regarding which ISRs might be needed. For example, if OpenCom() is invoked one or more times but the compiler can determine that the Com1 is always the channel being used, no additional ISRs are included since the Com1 ISRs are always included. On the other hand, if the compiler cannot determine which channel is being opened in one or more cases, it includes the ISRs for all Com channels, both hardware-based and software-based channels.

In the description of each System Library routine, information is given about the set of ISRs might be included in your program if you use that routine. This information is only important, of course, if you are also providing one or more ISRs in your code because conflicts may arise. (See the section entitled “Defining Interrupt Service Routines” in the ZBasic Language Reference Manual for more information on how this is done.) The table below gives an overview of which System Library routines may cause ISRs to be included automatically in your program. The shaded entries represent ISRs that are always included.

| System Library Routines that May Load ISRs |                           |              |              |              |
|--|---------------------------|--------------|--------------|--------------|
| Routine                                    | mega644P                  | mega128      | mega1281     | mega1280     |
| ADCToCom1()                                | Timer1_CompA              | Timer1_CompA | Timer4_CompA | Timer4_CompA |
| Com1ToDAC()                                | Timer1_CompA              | Timer1_CompA | Timer4_CompA | Timer4_CompA |
| InputCapture()                             | Timer1_Capt<br>Timer1_Ovf | Timer1_Capt  | Timer1_Capt  | Timer1_Capt  |
|  |                           | Timer1_Ovf   | Timer1_Ovf   | Timer1_Ovf   |
|  |                           | Timer3_Capt  | Timer3_Capt  | Timer3_Capt  |
|  |                           | Timer3_Ovf   | Timer3_Ovf   | Timer3_Ovf   |
|  |                           |              |              | Timer4_Capt  |
| OpenCom()                                  |                           |              |              | Timer4_Ovf   |
|  |                           |              |              | Timer5_Capt  |
|  |                           |              |              | Timer5_Ovf   |
|  | USART0_RX                 | USART0_RX    | USART0_RX    | USART0_RX    |
|  | USART0_TX                 | USART0_TX    | USART0_TX    | USART0_TX    |
|  | USART0_UDRE               | USART0_UDRE  | USART0_UDRE  | USART0_UDRE  |
|  | USART1_RX                 | USART1_RX    | USART1_RX    | USART1_RX    |
|  | USART1_TX                 | USART1_TX    | USART1_TX    | USART1_TX    |
|  | USART1_UDRE               | USART1_UDRE  | USART1_UDRE  | USART1_UDRE  |
|  |                           |              |              | USART2_RX    |
|  |                           |              |              | USART2_TX    |
|  |                           |              |              | USART2_UDRE  |
|  |                           |              |              | USART3_RX    |
|  |                           |              |              | USART3_TX    |
|  |                           |              |              | USART3_UDRE  |

|                    |              |              |              |              |
|--------------------|--------------|--------------|--------------|--------------|
|                    | Timer2_CompA | Timer2_CompA | Timer0_CompA | Timer0_CompA |
| OpenX10()          | INT0         | INT0         | INT0         | INT0         |
|                    | Timer0_CompB | Timer0_CompB | Timer2_CompB | Timer2_CompB |
| OutputCapture()    | Timer1_CompB | Timer1_CompB | Timer1_CompB | Timer1_CompB |
|                    |              | Timer1_CompC | Timer1_CompC | Timer1_CompC |
|                    |              | Timer3_CompB | Timer3_CompB | Timer3_CompB |
|                    |              |              |              | Timer4_CompB |
|                    |              |              |              | Timer5_CompB |
| WaitForInterrupt() | INT0         | INT0         | INT0         | INT0         |
|                    | INT1         | INT1         | INT1         | INT1         |
|                    | INT2         | INT2         | INT2         | INT2         |
|                    |              | INT3         | INT3         | INT3         |
|                    |              | INT4         | INT4         | INT4         |
|                    |              | INT5         | INT5         | INT5         |
|                    |              | INT6         | INT6         | INT6         |
|                    |              | INT7         | INT7         | INT7         |
|                    | PCINT0       |              | PCINT0       | PCINT0       |
|                    | PCINT1       |              | PCINT1       | PCINT1       |
|                    | PCINT2       |              | PCINT2       | PCINT2       |
|                    | PCINT3       |              |              |              |
|                    | Analog_Comp  | Analog_Comp  | Analog_Comp  | Analog_Comp  |

## Timers

ZX devices have three or more timers, depending on the underlying CPU type, that are used for various purposes. One of the timers is used to implement the real time clock (RTC), another is used for the software-based serial ports and a third timer is used to provide the precise timing required for certain I/O routines. The specific timer that is used for a particular function varies depending on the underlying CPU type as shown in the table below.

| 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        |
| mega644P                | 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 programmed to generate an interrupt that is used to update the RTC and to trigger task switching. Because its role is so central, the RTC Timer cannot be used for any other purpose. The I/O Timer is used by several I/O related routines as explained in more detail below. The Serial Timer is used to generate interrupts to implement the timing required for serial channels Com3 to Com6. If none of the channels 3-6 is open, the Serial Port Timer can be used for other purposes in your program. Timers are also used for some specialized I/O functions as indicated in the table above.

For each timer, there exists a built-in variable that indicates when the timer is in use. For example, `Register.Timer0Busy` is a Boolean value that indicates when Timer0 is in use. Prior to using a timer, the system checks the value of this variable to see if it is already being used. If it is not in use, the system sets the flag to `True` and then proceeds to use the timer. When it is finished using the timer, the system sets the busy flag to `False`. Your program may do the same by passing the `Register` variable as a parameter to the `Semaphore()` function.

## I/O Timer Pre-scaler Values

Some of the System Library routines that use a timer allow you to modify the frequency used to clock the timer while others use a fixed frequency determined by the requirements of the routine. The routines that do allow frequency modification are divided into two groups, one controlled by the value of `Register.TimerSpeed1` and the other controlled by the value of `Register.TimerSpeed2`. The table below shows the System Library routines that use a timer and, where applicable, the timer speed variable that controls the timer frequency.

| System Library Routines Using TimerSpeed Values |                          |
|---|--------------------------|
| Routine   | TimerSpeed Value         |
| ADCToCom1()                                     |                          |
| Com1toDAC()                                     |                          |
| CountTransitions()                              | TimerSpeed1 <sup>1</sup> |
| FreqOut()                                       |                          |
| Get1Wire()                                      |                          |
| Get1WireByte()                                  |                          |
| Get1WireData()                                  |                          |
| I2CCmd() <sup>2</sup>                           | TimerSpeed1              |
| I2CGetByte() <sup>2</sup>                       | TimerSpeed1              |
| I2CPutByte() <sup>2</sup>                       | TimerSpeed1              |
| InputCapture()                                  | TimerSpeed1              |
| InputCaptureEx()                                | TimerSpeed1              |
| OutputCapture()                                 | TimerSpeed1              |
| OutputCaptureEx()                               | TimerSpeed1              |
| OpenPWM()                                       |                          |
| RCTime()  | TimerSpeed2 <sup>1</sup> |
| PlaySound()                                     |                          |
| PulseIn()                                       | TimerSpeed2 <sup>1</sup> |
| PulseOut()                                      | TimerSpeed2 <sup>1</sup> |
| Put1Wire()                                      |                          |
| Put1WireByte()                                  |                          |
| Put1WireData()                                  |                          |
| PWM()   |                          |
| Reset1Wire()                                    |                          |
| ShiftIn()                                       | TimerSpeed1              |
| ShiftInEx()                                     | TimerSpeed1              |
| ShiftOut()                                      | TimerSpeed1              |
| ShiftOutEx()                                    | TimerSpeed1              |
| X10Cmd()  |                          |

**Notes:**

- 1) The timer frequency is scaled. See below.
- 2) The timer is used only for channels 1-4.

The default value of `Register.TimerSpeed1` is 1 and the default value for `Register.TimerSpeed2` is 2. The table below shows the correspondence between the allowable values for the `TimerSpeed` registers and the resulting clock frequency applied to the I/O Timer. The divisor specified is applied to the CPU clock frequency to yield the I/O Timer clock frequency. For compatibility with BasicX (but only for ZX processors running at 14.7456MHz), some of the routines effectively divide the timer frequency by 2 so that the time units associated with parameters or return values is preserved. If you change the timer speed setting, the scale factor is still applied.

| TimerSpeed Pre-Scaler Values |         |                            |                    |
|------------------------------|---------|----------------------------|--------------------|
| TimerSpeed Value             | Divisor | Timer Clock Freq.          | Timer Clock Period |
| 0                            | n/a     | 0 Hz                       | -                  |
| 1                            | 1       | 14.7456 MHz                | 67.8nS             |
| 2                            | 8       | 1.8432 MHz                 | 542nS              |
| 3                            | 64      | 230.4 KHz                  | 4.34µS             |
| 4                            | 256     | 57.6 KHz                   | 17.4µS             |
| 5                            | 1024    | 14.4 KHz                   | 69.4µS             |
| 6                            | n/a     | External – T1 falling edge | -                  |
| 7                            | n/a     | External – T1 rising edge  | -                  |

Note that setting the value of either of the timer speed registers other than by direct assignment using an assignment statement will produce undefined results. Also note that on the ZX-24 series devices, the T1 input signal is common with Port C, bit 7. If you wish to use an external clock source you'll have to configure pin 5 to be an input so as not to interfere with that signal. Of course, transitions on Port C bit 7 can be used to clock the timer when the T1 input signal is selected.

There are several important facts to keep in mind if you modify either of the timer speed values. Firstly, the timer speed values are initialized by the system when it begins running and they are never modified by the system thereafter. If you change a timer speed value, that value will be used by all of the related System Library routines until you change it again. Secondly, values returned by some of the System Library routines are scaled based on the default timer speed values. If you change the timer speed, you'll have to apply an additional scale factor in order to get the correct results. For example, if you set `Register.TimerSpeed2` to 3 and then call the subroutine `PulseIn()`, a pulse having a width of 100µS will return the value of approximately 12.5µS since the clock speed that you specified is 1/8 that of the default. In order to get the correct pulse width, in seconds, you will have to multiply the value returned by 8. Those return values that are not scaled to seconds represent a number of periods of the timer frequency. So, for example, if you change `Register.TimerSpeed1` to 2, the values returned by `InputCapture()` represent units of 542nS instead of the default 67.8nS.

The other Register value related to the I/O Timer is the "timer busy" flag, e.g. `Register.Timer1Busy`. Whenever a System Library routine that requires the I/O Timer prepares to execute, it first checks the value of this Boolean flag to see if the timer is already in use. If the flag is `True`, the routine will not execute; usually returning without doing anything (but see the descriptions of the various routines for specific details). If the flag is `False`, the routine sets it to `True` and then goes about using the timer. When it has finished its function, it sets the flag back to `False`.

Your code can use the timer busy flag as the parameter to the `Semaphore()` function in order to get exclusive access to the timer. Of course, you must set timer busy flag to `False` when your code is finished with the timer to indicate that the timer is no longer in use. Likewise, you may want to acquire a semaphore on a timer busy flag for the I/O Timer before calling a System Library routine that uses I/O Timer. If you succeed in setting the semaphore you'll know that the timer is not already in use. An example of code for this purpose (for ZX devices that use Timer1 for the I/O Timer) is shown below.

```
' wait until the timer is available
Do While (Not Semaphore(Register.Timer1Busy))
    Call Sleep(0.5)
Loop

' use the timer
Call LockTask()
Register.Timer1Busy = False
Call ShiftOut(12, 13, 8, &H55)
Call UnlockTask()
```

Note, particularly, the line immediately before the call to `ShiftOut()`. After the semaphore is acquired `Register.Timer1Busy` will be `True`. Unless it is set to `False`, the call to `ShiftOut()` will fail because that subroutine will think that the timer is in use.

**Caution:** setting the busy flag for a timer to `True` and never setting it back to `False` will prevent System Library routines that require that timer from functioning.

## Detailed Descriptions

In the descriptions that follow, the parameter types that are accepted by each routine are described. Some parameters accept a specific fundamental data type while others may accept a few similar types. Others accept virtually any parameter type. In order to more succinctly describe the types of parameters accepted, some descriptive type categories are used. For example, the category *integral* is used to connote those types that have the integral characteristic, such as `Byte`, `Integer`, `UnsignedInteger`, `Long` and `UnsignedLong`. The table below indicates which types belong to which categories.

**Type Category Membership**

| Type/Category   | any type | integral | int8/16 | int16 | int32 | any 32-bit | signed | numeric |
|-----------------|----------|----------|---------|-------|-------|------------|--------|---------|
| Boolean         | x        |          |         |       |       |            |        |         |
| Bit             | x        | x        | x       |       |       |            |        | x       |
| Nibble          | x        | x        | x       |       |       |            |        | x       |
| Byte            | x        | x        | x       |       |       |            |        | x       |
| Integer         | x        | x        | x       | x     |       |            | x      | x       |
| UnsignedInteger | x        | x        | x       | x     |       |            |        | x       |
| Enum            | x        |          |         |       |       |            |        |         |
| Long            | x        | x        |         |       | x     | x          | x      | x       |
| UnsignedLong    | x        | x        |         |       | x     | x          |        | x       |
| Single          | x        |          |         |       |       | x          | x      | x       |
| String          | x        |          |         |       |       |            |        |         |

The remainder of this document presents complete descriptions of each of the System Library routines, arranged in alphabetical order. Unless specifically noted otherwise, the descriptions apply to all ZX models. In some cases, a routine exhibits different behavior in BasicX compatibility mode or operates in a manner that is slightly different from that implemented in the BasicX environment. In these cases, the heading **Compatibility** will appear in the description detailing the differences. The advanced System Library routines that are not present in the BasicX environment are also similarly noted. If you are not using BasicX compatibility mode or are not upgrading BasicX code these notations may be safely ignored.



# Abs

---

**Type**            Function returning the same type as the parameter

**Invocation**    Abs(arg)

| Parameter | Method | Type    | Description   |
|-----------|--------|---------|---|
| arg       | ByVal  | numeric | The value from which the absolute value will be computed. |

## Discussion

The absolute value function returns the magnitude of the passed value. It is primarily useful for signed numeric types such as `Single`, `Integer` and `Long`. Unsigned parameter values will be returned unchanged.

The type of the return value will be the same as the type of the parameter provided.

## Example

```
Dim i as Integer, j as Integer

i = -45
j = Abs(i) ' result is 45
```

# Acos

---

**Type**            Function returning Single

**Invocation**    Acos(arg)

| Parameter | Method | Type   | Description   |
|-----------|--------|--------|---|
| arg       | ByVal  | Single | The value from which the arc cosine will be computed. |

## Discussion

The arc cosine function is the inverse of the cosine function. The return value will be the angle, expressed in radians, whose cosine corresponds to the passed value. The type of the return value will be `Single` and the value will range from 0.0 to  $\pi$ . If the argument is greater than 1.0 or less than  $-1.0$ , the result will be undefined.

## Example

```
Dim val as Single, theta as Single

val = 0.5
theta = Acos(val) 'the result will be approximately 1.0472.
```

**See Also**        Cos, DegToRad, RadToDeg

# ADCtoCom1

**Type** Subroutine

**Invocation** ADCtoCom1(pin, rate)

| Parameter | Method | Type  | Description   |
|-----------|--------|-------|---|
| pin       | ByVal  | Byte  | The pin number from which analog readings will be taken. Valid pins are those corresponding to PortA, pins 13 to 20.                                  |
| rate      | ByVal  | int16 | The rate at which conversions will be performed. The value is the number of conversions per second and may range from 28 to 11000 samples per second. |

## Discussion

Calling this subroutine causes a continuous series of analog-to-digital conversions to be performed on the signal appearing at the specified pin. Each 8-bit digital result is automatically sent out the Com1 serial port. Before starting the conversions, the baud rate of Com1 is set to 115,200. The specified pin is automatically set to the proper state for A/D conversion so no additional setup is required prior to use. The conversion stream will continue until `ADCToCom1()` is called again with the `pin` parameter set to zero (the `rate` parameter being meaningless in this case).

The analog input range is approximately 0.25 to 0.75 times  $V_{cc}$  (1.25 volts to 3.75 volts when running on 5 volts) and the resulting digital range is 0 to 255. Analog input levels below the low end of the range and above the high end of the range will produce the low and high digital values, respectively.

Note that the subroutine `Com1ToDAC()` is designed to receive the data stream generated by this Subroutine.

For best accuracy, state changes on other pins of PortA should be avoided during the conversion process.

## Resource Usage

This subroutine uses the processor's A/D converter, Com1 and the I/O Timer. No other use of these resources should be attempted while the conversion is active. For native code devices, the following ISRs are required.

| ISRs Required     |              |
|-------------------|--------------|
| Underlying CPU    | ISR Name     |
| mega644P, mega128 | Timer1_CompA |
| mega1281          | Timer4_CompA |
| mega1280          | Timer4_CompA |

## Example

```
' Begin the conversion stream on pin 12 at 500 samples per second
Call ADCtoCom1(12, 500)

' Stop the conversion process after two minutes
Call Delay(120.0)
Call ADCtoCom1(0, 0)
```

**See Also** Com1toDAC

# Asc

---

**Type**                Function returning Byte

**Invocation**        Asc(str)  
                         Asc(str, index)

| Parameter | Method | Type    | Description   |
|-----------|--------|---------|---|
| str       | ByVal  | String  | The string from which a character will be returned.                           |
| index     | ByVal  | int8/16 | The 1-based position in the string from which the character will be returned. |

## Discussion

This function returns the ASCII character code of the character at the position of the string that is specified. If the second parameter is missing, position 1 is assumed. Note that if the index is less than 1 or larger than the number of characters in the string the return value will be zero.

## Example

```
Dim s as String
Dim b as Byte

s = "Howdy"
b = Asc(s)
```

After execution, the variable `b` will have the value of 72 (48 hex), the character code for H.

## Compatibility

BasicX does not support the presence of the second parameter.

**See Also**            Chr

# Asin

---

**Type**            Function returning Single

**Invocation**    Asin(arg)

| Parameter | Method | Type   | Description   |
|-----------|--------|--------|---|
| arg       | ByVal  | Single | The value from which the arc sine will be computed. |

## Discussion

The arc sine function is the inverse of the sine function. The return value will be the angle, expressed in radians, whose sine corresponds to the passed value. The type of the return value will be `Single` and the value will range from  $-\pi/2$  to  $\pi/2$ . If the argument is greater than 1.0 or less than  $-1.0$ , the result will be undefined.

## Example

```
Dim val as Single, theta as Single

val = 0.5
theta = Asin(val)                    ' result is approximately 0.5236
```

**See Also**        Sin, DegToRad, RadToDeg

# Atn

---

**Type**            Function returning Single

**Invocation**    Atn(arg)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| arg       | ByVal  | Single | The value from which the arc tangent will be computed. |

## Discussion

The arc tangent function is the inverse of the tangent function. The return value will be the angle, expressed in radians, whose tangent corresponds to the passed value. The return value will be of type `Single` and the value will range from  $-\pi/2$  to  $\pi/2$ .

## Example

```
Dim val as Single, theta as Single

val = 0.5
theta = Atn(val) ' result is approximately 0.4636
```

**See Also**            Atn2, DegToRad, RadToDeg

# Atn2

**Type**            Function returning Single

**Invocation**    Atn2(y, x)

| Parameter | Method | Type   | Description   |
|-----------|--------|--------|---------------|
| y         | ByVal  | Single | y coordinate. |
| x         | ByVal  | Single | x coordinate. |

## Discussion

This function computes the principal value of the arc tangent of  $y/x$ , using the signs of both arguments to determine the quadrant of the return value. The return value will be the angle, expressed in radians, from the positive x-axis to the line connecting the origin and the given point. The type of the return value will be `Single` and the value will range from - to . If x is zero, the result is undefined unless y is also zero in which case 0.0 will be returned.

## Example

```
Dim x as Single, y as Single, theta as Single

x = 1.0
y = -1.0
theta = Atn2(y, x)            ' result is -0.7854
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        Atn, DegToRad, RadToDeg

# BitCopy

**Type** Subroutine

**Invocation** BitCopy(destAddr, destBitOfst, srcAddr, srcBitOfst, bitCount)

| Parameter  | Method | Type     | Description                                 |
|------------|--------|----------|---|
| dstAddr    | ByVal  | integral | The address to which to begin copying.      |
| dstBitOfst | ByVal  | integral | The bit offset to which to begin copying.   |
| srcAddr    | ByVal  | integral | The address from which to begin copying.    |
| srcBitOfst | ByVal  | integral | The bit offset from which to begin copying. |
| bitCount   | ByVal  | integral | The number of bits to copy.                 |

## Discussion

This subroutine can be used to copy an arbitrary number of bits from one location in RAM to another. The copy operation may begin and/or end in the middle of a byte if desired. An overlapping copy (when the destination is in the midst of the data being copied) is handled correctly so that the data to be copied is not overwritten.

For the purposes of this subroutine, RAM considered a sequence of bits with the least significant bits of each byte preceding the more significant bits. This is the same model of RAM that is utilized by `GetBit()` and `PutBit()`. The least significant bit of a byte is at offset zero and the most significant bit is at offset 7.

Note that the bit offsets specified for the second and fourth parameters may have values greater than 7. If a bit offset greater than 7 is given, the corresponding address component is adjusted internally to give the same effect. For example, if an address of 200 and a bit offset of 19 are specified, these are converted internally to 202 and 3, respectively.

All six parameters are converted internally to `UnsignedInteger`.

## Caution

This subroutine should be used with care because it is possible to overwrite important data on the stack or other areas of memory which may cause your program to malfunction.

## Compatibility

This subroutine is not available on ZX models that are based on the ATmega32 processor (e.g. the ZX-24). Moreover, it is not available in BasicX compatibility mode.

**See Also** MemCopy, MemSet



# BlockMove

---

**Type** Subroutine

**Invocation** BlockMove(count, source, destination)

| Parameter   | Method | Type     | Description                              |
|-------------|--------|----------|--|
| count       | ByVal  | integral | The number of bytes to copy.             |
| source      | ByVal  | integral | The address from which to begin copying. |
| destination | ByVal  | integral | The address to which to begin copying.   |

## Discussion

This subroutine is provided for compatibility with BasicX. The more aptly named `MemCopy()` should be used by new applications. An overlapping copy (when the destination is in the midst of the data being copied) is handled correctly so that the data to be copied is not overwritten.

## Compatibility

With firmware versions prior to v1.1.0 an overlapping copy is not handled correctly nor is it handled correctly in BasicX. A BasicX application that relies on the incorrect handling will, therefore, not work as expected when run on ZX processors.

**See Also** BitCopy, MemCopy

# BusRead

**Type** Subroutine

**Invocation** BusRead(addr, data, count)  
BusRead(addr, data, count, delta)

| Parameter | Method | Type     | Description  |
|-----------|--------|----------|--|
| addr      | ByVal  | integral | The bus address at which to begin reading.                                 |
| data      | ByRef  | anyType  | A buffer to receive the data read.   |
| count     | ByVal  | integral | The number of bytes to read.   |
| delta     | ByVal  | integral | The amount by which the address should be changed after each byte is read. |

## Discussion

For ZX models that support external RAM (e.g. ZX-1281), if the external RAM interface is enabled and a bus has not been defined using DefineBus(), then the external RAM interface is used for the read operation. In this case, the full 16 bits of the specified address are used and the delta parameter is interpreted as a signed 8-bit value that is sign-extended before adding it to the address with each iteration.

For ZX models that do not support external RAM or if the external RAM interface is not enabled, this routine performs a series of read operations on the bus previously defined with the DefineBus() call. This is called the “bit bang” mode. For each read cycle, the low 8-bits of the address is output on the previously specified port and then the ALE pin is strobed (high, then back low). Next, the port is made an input and the RD pin is set low, data is read via the PIN register corresponding to the port, and the RD pin is set back high again. The data value read is stored in the buffer, the specified delta is added to the 8-bit bus address and the cycle is repeated until the specified number of bytes has been read.

It is important to remember that in the bit bang mode only 8 bits of the address are used. Depending on the values of the addr, count and delta parameters, the effective address may wrap around to zero. For example, with delta=1 specifying a count parameter larger than (256 - LoByte(addr)) will result in the effective address wrapping around to zero.

In either mode, if the optional delta parameter is not specified, the value of 1 is assumed. Specifying the delta as zero will result in multiple reads from the same address. A delta of -1 or &Hff will result in the address being decremented after each read.

## Example

```
Dim data(1 to 20) as Byte
Call DefineBus(Port.A, C.0, C.1, C.2)
Call BusRead(0, data, SizeOf(data))
```

## Compatibility

This subroutine is not available on ZX models that are based on the ATmega32 processor (e.g. the ZX-24). Moreover, it is not available in BasicX compatibility mode.

**See Also** BusWrite, DefineBus

# BusWrite

**Type** Subroutine

**Invocation** BusWrite(addr, data, count)  
BusWrite(addr, data, count, delta)

| Parameter | Method | Type     | Description   |
|-----------|--------|----------|---|
| addr      | ByVal  | integral | The bus address at which to begin writing.                                    |
| data      | ByRef  | anyType  | The data to be written.   |
| count     | ByVal  | integral | The number of bytes to write.   |
| delta     | ByVal  | integral | The amount by which the address should be changed after each byte is written. |

## Discussion

For ZX models that support external RAM (e.g. ZX-1281), if the external RAM interface is enabled and bus has not been defined using DefineBus(), then the external RAM interface is used for the write operation. In this case, the full 16 bits of the specified address are used and the delta parameter is interpreted as a signed 8-bit value that is sign-extended before adding it to the address with each iteration.

For ZX models that do not support external RAM or if the external RAM interface is not enabled, this routine performs a series of write operations on the bus previously defined with the DefineBus() call. This is called the “bit bang” mode. For each write cycle, the low 8-bits of the address is output on the previously specified port and then the ALE pin is strobed (high, then back low). Then, the next data value to be written is output on the port and the WR pin is strobed (low then back high). Finally, the specified delta is added to the bus address and the cycle is repeated until the specified number of bytes has been written.

It is important to remember that in the bit bang mode only 8 bits of the address are used. Depending on the values of the addr, count and delta parameters, the effective address may wrap around to zero. For example, with delta=1 specifying a count parameter larger than (256 - LoByte(addr)) will result in the effective address wrapping around to zero.

In either mode, if the optional delta parameter is not specified, the value of 1 is assumed. Specifying the delta as zero will result in multiple writes to the same address. A delta of -1 or &Hfff will result in the address being decremented after each write.

## Example

```
Dim data(1 to 20) as Byte
Call DefineBus(Port.A, C.0, C.1, C.2)
Call BusWrite(0, data, SizeOf(data))
```

## Compatibility

This subroutine is not available on ZX models that are based on the ATmega32 processor (e.g. the ZX-24). Moreover, it is not available in BasicX compatibility mode.

**See Also** BusRead, DefineBus

# CallTask

**Type** Special Purpose

**Invocation** CallTask taskName, taskStack  
CallTask taskName, taskStack, taskStackSize  
CallTask taskName( parameterList ), taskStack  
CallTask taskName( parameterList ), taskStack, taskStackSize

| Parameter     | Method | Type          | Description  |
|---------------|--------|---------------|--|
| taskName      | ByVal  | identifier    | The name of the task to invoke.  |
| parameterList | varies | varies        | Zero or more parameters to be passed to the task, separated by commas. |
| taskStack     | ByRef  | array of Byte | The stack for the task (see discussion)                                |
| taskStackSize | ByVal  | integral      | The size of the stack.   |

## Discussion

This construct is used to prepare a task for running; the task doesn't actually execute until its turn comes up in the normal task rotation. In the first and second cases, the `taskName` given must be the name of a user-defined subroutine that takes no parameters. In the third and fourth cases, the `taskName` given must be a user-defined subroutine that takes a number of parameters whose type and number match that of the supplied parameter list. The subroutine may be public or private but if it is private it must exist in the same module as the `CallTask` invocation that refers to it.

The `taskStack` may be a `Byte` array, typically defined at the module level, that contains a sufficient amount of space for the task's stack needs. The array can be public or private but if it is private it must exist in the same module as the `CallTask` invocation that refers to it. Alternately, the stack for a task may be specified by giving its address as an integral expression. In this case, it is usually also advisable to specify the size of the stack since the compiler cannot deduce the size. A task must have exclusive use of the memory dedicated to its task stack. A particular task stack may be used by more than one task but one task must terminate before the next task can re-use the task stack.

If a task is passed parameters when it is invoked, it is advisable that those parameters be passed `ByVal` because the lifetime of the task may exceed the lifetime of the routine from which the task was invoked. If parameters are passed `ByRef` (explicitly or implicitly), the compiler will issue a warning. Also, certain types of expressions (notably, those involving user-defined functions that return `String` types) may not be used as parameter values for task invocation because they require the creation of temporary variable space on the stack during evaluation. The compiler will issue an error message when it detects such situations. This problem can be rectified by manually creating a variable (preferably at the module level) to hold the parameter value.

For native mode devices (e.g. ZX-24n), the task stack size must either be explicitly specified or it must be determinable by the compiler from the size of the task stack array. The compiler will issue an error message if it cannot determine the size of the task stack.

Please read the section on multi-tasking in the ZBasic Reference Manual for more details, including information about how to determine the proper task stack size.

## Example 1

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

```

## Example 2

```

Dim taskStack(1 to 50) as Byte

Sub Main()
    CallTask MyTask(2.0), taskStack
    Do
        Debug.Print "Hello from Main"
        Call Delay(1.0)
    Loop
End Sub

Sub MyTask(ByVal taskDelay as Single)
    Do
        Debug.Print "Hello from MyTask"
        Call Delay(taskDelay)
    Loop
End Sub

```

## Example 3

```

Dim taskStack(1 to 50) as Byte

Sub Main()
    Dim stkAddr as UnsignedInteger
    Dim stkSize as Integer

    stkAddr = taskStack.DataAddress
    stkSize = SizeOf(taskStack)
    CallTask MyTask(2.0), stkAddr, stkSize
    Do
        Debug.Print "Hello from Main"
        Call Delay(1.0)
    Loop
End Sub

Sub MyTask(ByVal taskDelay as Single)
    Do
        Debug.Print "Hello from MyTask"
        Call Delay(taskDelay)
    Loop
End Sub

```

## Compatibility

In BasicX compatibility mode, the task name must be enclosed in quotes (i.e. so that it appears to be a string). Also, task parameters, specifying the task stack by address, and specifying the task stack size are not supported in BasicX compatibility mode.

# CBit

---

**Type**            Function returning Bit

**Invocation**    CBit(arg)

| Parameter | Method | Type                | Description                          |
|-----------|--------|---------------------|--------------------------------------|
| arg       | ByVal  | integral or Boolean | The value to convert to a Bit value. |

## Discussion

This function converts a numeric or Boolean value to a Bit value. In all cases, the result will be the least significant bit of the passed value without regard to its type.

## Example

```
Dim pinVal as Bit  
  
pinVal = CBit(GetPin(12))
```

## Compatibility

This function is not available in BasicX compatibility mode.

# CBool

---

**Type**            Function returning Boolean

**Invocation**    CBool(arg)

| Parameter | Method | Type | Description                              |
|-----------|--------|------|--|
| arg       | ByVal  | Byte | The value to convert to a Boolean value. |

## Discussion

This function converts a Byte value to a Boolean value. If the byte has the value 0 the result will be False, otherwise it will be True.

## Example

```
Dim pinHi as Boolean  
  
pinHi = CBool(GetPin(12))
```

# CByte

**Type**            Function returning Byte

**Invocation**    CByte(arg)

| Parameter | Method | Type            | Description                   |
|-----------|--------|-----------------|-------------------------------|
| arg       | ByVal  | numeric or Enum | The value to convert to Byte. |

## Discussion

This function converts any numeric or enumeration value to a Byte value. See the table below for details of the conversion.

| Input Type      | Result  |
|-----------------|---|
| Boolean         | Returns the byte value of the Boolean data item: 0 or 255.  |
| Byte            | No effect, the value is as supplied.  |
| Integer         | Returns the low byte of the value provided. However, if the supplied value is negative or greater than 255, the returned value will be 255.   |
| UnsignedInteger | Returns the low byte of the value provided. However, if the supplied value is greater than 255, the returned value will be 255.   |
| Enum            | Returns the low byte of the value provided. However, if the supplied value is greater than 255, the returned value will be 255.   |
| Long            | Returns the low byte of the value provided. However, if the supplied value is negative or greater than 255, the returned value will be 255.   |
| UnsignedLong    | Returns the low byte of the value provided. However, if the supplied value is greater than 255, the returned value will be 255.   |
| Single          | The supplied value is converted to a Long value (signed 32-bit integer), rounded to the nearest integer. If the fractional part is exactly 0.5, the resulting integer will be even. This is known as "statistical rounding". If the resulting integer value is negative or larger than 255, the result will be 255. Otherwise, the result will be the integral value.   |
| String          | The result is the numeric value of the characters in the string, ignoring leading space and tab characters. The value string may begin with a plus or minus sign and an optional radix indicator (&H for hexadecimal, &O for octal, &B or &X for binary, all case insensitive). The conversion is terminated upon reaching the end of the string or encountering the first character that is not valid for the indicated radix. |

## Compatibility

In BasicX, calling CByte() with an UnsignedInteger argument returns the low byte of the value. This behavior is inconsistent with the other type conversions. This implementation attempts to make them consistent.



# CByteArray

---

**Type**                Function returning a reference to a Byte array

**Invocation**        CByteArray(addr)

| Parameter | Method | Type  | Description   |
|-----------|--------|-------|---|
| addr      | ByVal  | int16 | The address to be converted to a reference to a Byte array. |

## Discussion

This special function is useful when you have an integral value that you know to be the address of a Byte array and you want to pass it to a subroutine or function that requires a Byte array parameter. The example below shows it being used to determine the number of bytes of data available in the system input queue.

## Example

```
Dim cnt as Integer
cnt = GetQueueCount(CByteArray(Register.RxQueue))
```

**See Also**            StatusTask

# Ceiling

---

**Type**                Function returning Single

**Invocation**        Ceiling(arg)

| Parameter | Method | Type   | Description                                |
|-----------|--------|--------|--|
| arg       | ByVal  | Single | The value of which to compute the ceiling. |

## Discussion

This function returns a Single value that is the smallest integer that is greater than or equal to the supplied value, effectively rounding up to the nearest integer.

## Example

```
Dim ceil as Single

ceil = Ceiling(1.5)        ' result is 2.0
ceil = Ceiling(-1.5)      ' result is -1.0
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            Floor, Fraction

# Chr

---

**Type**            Function returning String

**Invocation**    Chr(arg)

| Parameter | Method | Type     | Description                                |
|-----------|--------|----------|--|
| arg       | ByVal  | integral | The character code to place in the string. |

## Discussion

This function returns a string containing a single character having the value of the supplied parameter. If the parameter is a multi-byte type such as Integer or Long the least significant byte of the value is used and the remaining bytes are ignored.

Tables of ASCII character values may be found in many places on the Internet. A search for “ASCII table” or “ASCII chart” will produce many results.

## Example

```
Dim s as String  
s = Chr(33)
```

After execution, *s* will be " !" because 33 is the decimal code for the exclamation mark.

**See Also**        Asc

# CInt

**Type**            Function returning Integer

**Invocation**    CInt(arg)

| Parameter | Method | Type            | Description                      |
|-----------|--------|-----------------|----------------------------------|
| arg       | ByVal  | numeric or Enum | The value to convert to Integer. |

## Discussion

This function converts any numeric or enumeration value to an Integer value. See the table below for details of the conversion.

| Input Type      | Result  |
|-----------------|---|
| Byte, Boolean   | High byte zero, low byte as supplied.   |
| Integer         | No effect, the value is as supplied.  |
| UnsignedInteger | Value bits are the same as supplied, although interpreted as a signed value.  |
| Enum            | The resulting value is the Enum member value.   |
| Long            | The resulting value will be the low word of the supplied value.   |
| UnsignedLong    | The resulting value will be the low word of the supplied value.   |
| Single          | The supplied value is converted to signed 32-bit integer, rounded to the nearest integer. If the fractional part is exactly 0.5, the resulting integer will be even. This is known as "statistical rounding". If the resulting integer is larger than will fit in 16-bits, the result is undefined.   |
| String          | The result is the numeric value of the characters in the string, ignoring leading space and tab characters. The value string may begin with a plus or minus sign and an optional radix indicator (&H for hexadecimal, &O for octal, &B or &X for binary, all case insensitive). The conversion is terminated upon reaching the end of the string or encountering the first character that is not valid for the indicated radix. |

## Example

```
Dim i as Integer
```

```
i = CInt(2.5)            ' result is 2  
i = CInt(1.5)            ' result is 2
```

# ClearQueue

**Type** Subroutine

**Invocation** ClearQueue(queue)

| Parameter | Method | Type          | Description              |
|-----------|--------|---------------|--------------------------|
| queue     | ByRef  | array of Byte | The queue to be cleared. |

## Discussion

This routine modifies the tracking information contained in the queue data structure to indicate that the queue is empty. If the queue is already empty, this has no effect. If there are characters in the queue, they will be discarded.

Note that before any queue operations are performed, the queue data structure must be initialized. See the discussion of `OpenQueue( )` for more details.

## Example

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

Call OpenQueue(inQueue, SizeOf(inQueue))
Call PutQueueStr(inQueue, "Hello")
Call ClearQueue(inQueue)
```

After the call to `ClearQueue()` the queue will no longer contain the characters that were added.

## Compatibility

BasicX allows any type for the first parameter. This implementation requires that it be an array of `Byte`.

# CLng

**Type**                      Function returning Long

**Invocation**            CLng(arg)

| Parameter | Method | Type            | Description                   |
|-----------|--------|-----------------|-------------------------------|
| arg       | ByVal  | numeric or Enum | The value to convert to Long. |

## Discussion

This function converts any numeric or enumeration value to a Long value. See the table below for details of the conversion.

| Input Type      | Result  |
|-----------------|---|
| Byte, Boolean   | High 3 bytes zero, low byte as supplied.  |
| Integer         | High word will be all ones if the supplied value is negative, zero otherwise. Low word as supplied.   |
| UnsignedInteger | High word zero, low word as supplied.   |
| Enum            | The resulting value is the Enum member value.   |
| Long            | No effect, the value is as supplied.  |
| UnsignedLong    | Value bits are the same as supplied, although interpreted as a signed value.  |
| Single          | The supplied value is converted to a signed 32-bit integer, rounded to the nearest integer. If the fractional part is exactly 0.5, the resulting integer will be even. This is known as "statistical rounding". If the magnitude of the supplied value is too large to be represented in 32 bits, the result is undefined.  |
| String          | The result is the numeric value of the characters in the string, ignoring leading space and tab characters. The value string may begin with a plus or minus sign and an optional radix indicator (&H for hexadecimal, &O for octal, &B or &X for binary, all case insensitive). The conversion is terminated upon reaching the end of the string or encountering the first character that is not valid for the indicated radix. |

## Example

```
Dim l as Long
```

```
l = CLng(2.5)            ' result is 2  
l = CLng(1.5)            ' result is 2
```

# CloseCom

---

**Type** Subroutine

**Invocation** CloseCom(channel, inQueue, outQueue)

| Parameter | Method | Type          | Description                                   |
|-----------|--------|---------------|---|
| channel   | ByVal  | Byte          | The serial channel to close.                  |
| inQueue   | ByRef  | array of Byte | The input queue associated with the channel.  |
| outQueue  | ByRef  | array of Byte | The output queue associated with the channel. |

## Discussion

This routine shuts down the specified serial channel. All communication is terminated even if there are still characters in the output queue that have not yet been sent. This call does not clear the queues. If that is a requirement, calls to `ClearQueue()` will need to be made. Alternately, you may want to use the value returned by `StatusCom()` to wait for all queued characters to be transmitted before invoking `CloseCom()`.

Invoking this subroutine for Com1 (`channel = 1`) does not actually close the Com1 channel. Rather, doing so causes Com1 to revert to the default speed (19.2K baud) and to using the default I/O queues.

If the specified serial channel is not open or if an invalid channel number is given the call has no effect. If the channel being closed is the only one of the software-based channels (Com3-Com6) that is open, the Serial Timer will be turned off and the corresponding timer busy flag will be set to False indicating that the Serial Timer is available for other uses.

**See Also** DefineCom, OpenCom, StatusCom

# Closel2C

---

**Type**            Subroutine

**Invocation**    Closel2C(channel)

| Parameter | Method | Type | Description                   |
|-----------|--------|------|-------------------------------|
| channel   | ByVal  | Byte | The I2C channel number (0-4). |

## Discussion

This subroutine closes an I2C channel. For the hardware I2C channel, it disables the on-board I2C controller allowing the hardware I2C pins to be used for other purposes. For software I2C channels it has no effect.

## Compatibility

This subroutine is not available in BasicX compatibility mode.

**See Also**        OpenI2C



# ClosePWM

---

**Type** Subroutine

**Invocation** ClosePWM(channel)

| Parameter | Method | Type | Description               |
|-----------|--------|------|---------------------------|
| channel   | ByVal  | Byte | The PWM channel to close. |

## Discussion

This subroutine terminates the PWM signal generation on the specified channel and all other PWM channels associated with the same timer. The resulting state of the output pins for the affected channels is indeterminate. If your application requires a specific output state, it is recommended that you call `PutPin()` to set the desired state prior to calling `ClosePWM()`.

A side effect of a successful `ClosePWM()` call is that the timer busy flag for the associated timer (e.g. `Register.Timer1Busy`) will be set to `False` indicating that the timer may be used for other purposes.

## Example

```
Call ClosePWM(1) ' terminate PWM on channel 1 and 2
```

## Compatibility

This subroutine is not available in BasicX compatibility mode.

**See Also** OpenPWM, PWM

# CloseSPI

---

**Type**            Subroutine

**Invocation**    CloseSPI(channel)

| Parameter | Method | Type | Description                   |
|-----------|--------|------|-------------------------------|
| channel   | ByVal  | Byte | The SPI channel number (1-4). |

## Discussion

This subroutine closes an SPI channel. The primary purpose for this subroutine is to cancel SPI Slave mode. It has no effect for channels that are not open or that are open in Master mode.

## Compatibility

This subroutine is not available in BasicX compatibility mode.

**See Also**        OpenSPI, SPICmd

# CloseWatchDog

---

**Type**            Subroutine

**Invocation**    CloseWatchDog()

## Discussion

This subroutine disables the watchdog timer.

## Compatibility

This subroutine is not available in BasicX compatibility mode.

**See Also**        OpenWatchDog, WatchDog

# CloseX10

**Type** Subroutine

**Invocation** CloseX10(channel, inQueue, outQueue)

| Parameter | Method | Type          | Description                                   |
|-----------|--------|---------------|---|
| channel   | ByVal  | Byte          | The X-10 channel to close.                    |
| inQueue   | ByRef  | array of Byte | The input queue associated with the channel.  |
| outQueue  | ByRef  | array of Byte | The output queue associated with the channel. |

## Discussion

This routine shuts down the specified X-10 communication channel. All communication is terminated even if there are still data in the output queue that have not yet been sent. This call does not clear the queues. If that is a requirement, calls to `ClearQueue()` will need to be made.

If the specified X-10 channel is not open or if an invalid channel number is given the call has no effect. The `inQueue` and `outQueue` parameters are currently not used but are present for congruency with `CloseCom()`. Zero values may be used for either or both parameters.

## Resource Usage

The X-10 communication requires the use of INT0. While any X-10 channel is open, a task awaiting an interrupt on the Int0 pin will be suspended indefinitely. Once all X-10 channels are closed, Int0 will function normally again.

## Compatibility

This subroutine is not available on ZX models that are based on the ATmega32 processor (e.g. the ZX-24). Moreover, it is not available in BasicX compatibility mode.

**See Also** DefineX10, OpenX10, StatusX10

# CNibble

**Type**            Function returning Nibble

**Invocation**    CNibble(arg)

| Parameter | Method | Type                | Description                             |
|-----------|--------|---------------------|---|
| arg       | ByVal  | integral or Boolean | The value to convert to a Nibble value. |

## Discussion

This function converts a numeric or Boolean value to a Nibble value. If the passed value is a Bit or Boolean value the three most significant bits of the return value will be zero. In all other cases, the result will be the four least significant bits of the passed value without regard to its type.

## Example

```
Dim nVal as Nibble  
  
nVal = CNibble(Register.PortC)
```

## Compatibility

This function is not available in BasicX compatibility mode.

# Com1toDAC

**Type** Subroutine

**Invocation** Com1toDAC(pin)

| Parameter | Method | Type | Description  |
|-----------|--------|------|--|
| pin       | ByVal  | Byte | The pin number on which the analog voltage will be re-created. |

## Discussion

Calling this subroutine prepares Com1 to receive a continuous stream of 8-bit values from an external source. The baud rate is automatically set 115,200. When each value is received, the value is output as an analog voltage on the specified pin. The resulting analog voltage will range from near 0 volts corresponding to the received value of 0 to near the processor voltage (usually +5 volts) corresponding to the received value of 255. The method used to create the analog voltage is similar to that used for `PutDAC()` and the signal will require some filtering. See the description of `PutDAC()` for more details. The output pin is updated at a fixed rate of 11,000 times per second.

This routine returns immediately after setting up the conversion process. The conversion process will be terminated if `Com1toDAC()` is called again with a parameter of zero. Also, if data is not received for approximately 200 cycles, the conversion process will be automatically terminated.

Note that the subroutine `ADCtoCom1()` is designed to produce the data stream to be received by this subroutine.

## Resource Usage

This subroutine uses Com1 and the I/O Timer. No other use of these resources should be attempted while the reception is active. For native code devices, the following ISRs are automatically loaded.

| ISRs Required     |              |
|-------------------|--------------|
| Underlying CPU    | ISR Name     |
| mega644P, mega128 | Timer1_CompA |
| mega1281          | Timer4_CompA |
| mega1280          | Timer4_CompA |

**See Also** ADCtoCom1

# ComChannels

**Type** Subroutine

**Invocation** ComChannels(count, maxSpeed)

| Parameter | Method | Type    | Description   |
|-----------|--------|---------|---|
| count     | ByVal  | Byte    | The total desired number of software-derived serial channels. |
| maxSpeed  | ByVal  | int8/16 | The desired maximum baud rate to be supported.                |

## Discussion

In addition to the serial channel implemented in hardware on the processor (Com1), the system can support up to four additional serial communication channels that are implemented in the system software. The software-based serial channels are numbered Com3 through Com6. However, by default, only one additional channel, Com3, is supported. If you want to use serial channels 4 through 6 you must call this subroutine first to specify the maximum number (up to 4) that you want to have available. This subroutine must be called only when there are no open software-based serial channels (COM3 through COM6). If it is called when one or more channels are already open, it will have no effect.

After ComChannels() has been invoked, the serial channels that will be available depends on the value specified by the `count` parameter. If the value 2 is specified, for example, channels Com3 and Com4 will be available. If the maximum value of 4 is specified, then serial channels 3, 4, 5 and 6 will be available. Once the number of software-based serial channels has been established you may then use DefineCom(), OpenCom(), and CloseCom() to manage the available channels by specifying the appropriate channel number in those calls.

In addition to specifying the total number of software-based serial channels that you want, you must also specify the maximum baud rate that you wish to utilize. The supported rates are 300, 600, 1200, 2400, 4800, 9600 and 19,200 baud but see below for additional discussion about the maximum baud.

Because the COM3 to COM6 serial channels are implemented in software, when one or more of the channels is open there will be a certain amount of processing overhead that will reduce the speed at which program instructions will be executed. Moreover, the processing overhead is higher when supporting higher baud rates as compared to lower baud rates and the overhead is higher when supporting a larger number of channels. It is prudent, therefore, to choose the lowest baud rate and lowest number of channels that is practical for your circumstances.

Also note that when supporting two or more channels, there is a small possibility that incoming characters might not be properly recognized at the highest rate. The probability of not being able to properly synchronize on the incoming character's start bit increases with each additional channel that is supported. For this reason, it is recommended that the maximum baud rate be limited to 9600 when configured for 2 or more channels.

## Resource Usage

The software-implemented serial channels utilize the Serial Timer for the bit rate timing. No other use of the Serial Timer should be attempted when serial channels 3-6 are open.

## Example

```
Dim iq4(1 to 20) as Byte
Dim oq4(1 to 20) as Byte

Call ComChannels(4, 4800)
```

```
Call DefineCom(6, 12, 13, &H80)  
Call OpenCom(6, 4800, iq4, oq4)
```

**Compatibility**

This routine is not available in BasicX compatibility mode.

**See Also**      DefineCom, CloseCom, OpenCom, StatusCom



# Console.Read

---

**Type**            Function returning Byte

**Invocation**    Console.Read()

## Discussion

This function can be invoked to retrieve a character from the input queue associated with Com1. If the value of `Register.Console.Echo` is `True`, the character will automatically be sent back out via the system output queue. When this function is called it will not return until a character is available. However, other tasks will continue to execute. You may wish to use the value returned by `Register.RxQueue` to find out if there are characters available before calling it. See the example below.

## Example

```
Dim b as Byte
b = Console.Read() ' this will wait until a character is available

If (GetQueueCount(CByteArray(Register.RxQueue)) > 0) Then
    b = Console.Read() ' read the next available character
End If
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        Console.ReadLine, Console.Write, Console.WriteLine

# Console.ReadLine

---

**Type**                Function returning String

**Invocation**        Console.ReadLine()

## Discussion

This function can be invoked to retrieve a sequence of characters from the system input queue terminated by an end-of-line character. If the value of `Register.Console.Echo` is `True`, each character received will automatically be sent back out via the system output queue. When this function is called it will not return until an end-of-line character is received. However, other tasks will continue to execute. The end-of-line character is line feed (&H0a) by default but you may change it using `Register.Console.EOL`.

While the characters of the line are being read, if a backspace character is received (&H08) the most recently received character will be deleted from the internal buffer. Additional backspace characters will each remove another character from the buffer until it is empty. If a carriage return is received (&H0d) it will be ignored unless `Register.Console.EOL` is a carriage return.

The end-of-line character is not included in the returned string and the maximum length of the string is 255 characters. Additional characters received after the 255<sup>th</sup> character will be discarded while awaiting the end-of-line character.

## Example

```
Dim s as String
s = Console.ReadLine()
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            Console.Read, Console.Write, Console.WriteLine

# Console.Write

---

**Type**            Special Purpose

**Invocation**    Console.Write(arg)

| Parameter | Method | Type   | Description               |
|-----------|--------|--------|---------------------------|
| arg       | ByVal  | String | A string to send to Com1. |

## Discussion

Console.Write is neither a subroutine nor a function. It has more in common with ZBasic statements but it is described here for ease of reference. This special purpose method is useful for outputting debugging information and other data to Com1. Note that no carriage return/new line is output after the string.

When this method is invoked, execution of the current task will not continue and no other task will be allowed to run until the string's characters have been transferred to the system output queue. The Debug.Print page contains some example code that illustrates a way to mitigate the latency that results from this implementation detail.

In contrast to other System Library routines that copy data to a queue, the string length is not limited to the system output queue length.

## Example

```
Console.Write("Hello, world! ")
```

```
Console.Write("The value is " & CStr(val))
```

This example uses the concatenation operator to produce a single string that is passed to the method.

**See Also**            Debug.Print, Console.Read, Console.ReadLine, Console.WriteLine

# Console.WriteLine

---

**Type** Special Purpose

**Invocation** Console.WriteLine(arg)

| Parameter | Method | Type   | Description               |
|-----------|--------|--------|---------------------------|
| arg       | ByVal  | String | A string to send to Com1. |

## Discussion

Console.WriteLine is neither a subroutine nor a function. It has more in common with ZBasic statements but it is described here for ease of reference. This special purpose method is useful for outputting debugging information and other data to Com1. Note that a carriage return/new line is always output following the string.

When this method is invoked, execution of the current task will not continue and no other task will be allowed to run until the string's characters have been transferred to the system output queue. This caveat applies separately to the string specified by the parameter and to the end-of-line sequence that is also output. The Debug.Print page contains some example code that illustrates a way to mitigate the latency that results from this implementation detail.

In contrast to other System Library routines that copy data to a queue, the string length is not limited to the system output queue length.

## Examples

```
Console.WriteLine("Hello, world! ")
```

```
Console.WriteLine("The value is " & CStr(val))
```

The second example uses the concatenation operator to produce a single string that is passed to the method.

**See Also** Debug.Print, Console.Read, Console.ReadLine, Console.Write

# Cos

---

**Type**            Function returning Single

**Invocation**    Cos(arg)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| arg       | ByVal  | Single | The angle, in radians, of which the cosine will be computed. |

## Discussion

The return value will be the cosine of the supplied value, ranging from -1.0 to 1.0.

## Example

```
Const pi as Single = 3.14159
Dim val as Single

val = Cos(pi)            ' result is -1.0
```

**See Also**        Acos, DegToRad, RadToDeg

# CountTransitions

**Type**                Function returning Long

**Invocation**        CountTransitions(pin, interval)

| Parameter | Method | Type           | Description   |
|-----------|--------|----------------|---|
| pin       | ByVal  | Byte           | The pin on which logic transitions will be counted.   |
| interval  | ByVal  | Single or Long | The time interval specified in seconds or I/O Timer ticks respectively, during which transitions will be counted. See the discussion below for information on range and resolution. |

## Discussion

When called, this routine will begin counting logic transitions on the specified pin and will continue until the specified interval has elapsed. During the counting process processor interrupts are disabled. This strategy allows high precision in measuring the interval but has the drawback that other processes that utilize interrupts will not function correctly. Among such affected processes are all serial communication and multi-tasking. For this reason, the counting interval should be kept as short as possible. RTC ticks that occur during the counting process are accumulated and the RTC is updated when the counting is finished.

The specified pin, which you must configure to be an input before calling, is sampled at a fixed rate of approximately 500KHz. The default resolution of the measurement interval is approximately 2.441 $\mu$ S with a maximum interval length of 5.2 seconds. If the `interval` parameter is specified using a `Single` value the units are seconds, otherwise the units are I/O Timer ticks where each tick is approximately 2.441  $\mu$ S (1/409.6KHz). You may modify the range and resolution of the measurement interval by modifying the built-in variable `Register.TimerSpeed1`. See the special section on Timers for more details.

## Resource Usage

This function uses the I/O Timer and disables interrupts during the counting process. However, RTC ticks are accumulated during the process and the RTC is updated upon completion.

## Compatibility

In BasicX missed RTC ticks are not accounted for.

# CPUSleep

---

**Type**                Subroutine

**Invocation**        CPUSleep()

## Discussion

This routine puts the processor into a special sleep mode in which activity and power consumption are reduced. The nature of the sleep mode is controlled by certain bits in one of the CPU registers (see table below). For more information about the sleep mode, consult the Atmel documentation for the ATmega processor on your ZX device is based.

**Register Containing the Sleep Mode Bits**

| <b>ZX Model</b>   | <b>Register</b> |
|---|-----------------|
| ZX-24, ZX-40, ZX-44, ZX-24e                               | Register.MCUCR  |
| ZX-24a, ZX-40a, ZX-44a, ZX-24ae                           | Register.SMCR   |
| ZX-24p, ZX-40p, ZX-44p, ZX-24pe                           | Register.SMCR   |
| ZX-24n, ZX-40n, ZX-44n, ZX-24ne                           | Register.SMCR   |
| ZX-1281, ZX-1281n, ZX-1280, ZX-1280n, ZX-1281e, ZX-1281ne | Register.SMCR   |
| ZX-128e, ZX-128ne   | Register.MCUCR  |

# CRC16

**Type** Function returning UnsignedInteger

**Invocation** CRC16(data, count, crcPoly, crclnit, crcFlags)

| Parameter | Method | Type            | Description                                 |
|-----------|--------|-----------------|---|
| data      | ByRef  | anyType         | The data bytes to add to the CRC value.     |
| count     | ByVal  | integral        | The number of bytes to process.             |
| crcPoly   | ByVal  | UnsignedInteger | The CRC polynomial to use.                  |
| crclnit   | ByVal  | UnsignedInteger | The initial value of the CRC.               |
| crcFlags  | ByVal  | integral        | Flag bits that control the CRC computation. |

## Discussion

This function computes the CRC value over a number of data bytes using a specified polynomial and initial value. The values to use for the polynomial and the initial value depend on the style of CRC that you need to generate. See the discussion below for further details. The `flags` parameter contains bits that control aspects of the CRC computation as described in the table below.

| Flag Values for the CRC Computation |      |           |  |
|-------------------------------------|------|-----------|--|
| Constant                            | Hex  | Binary    | Description                                |
| zxCRCRefIn                          | &H01 | xxxx xxx1 | Each input data bytes will be “reflected”. |
| zxCRCRefOut                         | &H02 | xxxx xx1x | The final CRC value will be “reflected”.   |

The remaining bits are reserved for future use and should always be zero.

In this context, the term “reflection” refers to reversing the order of the bits in a data item so that the most significant becomes the least significant and vice versa. For a multi-byte data item, the bits in each byte are reversed and the order of the bytes is reversed as well.

Although this function will typically be used to compute the CRC value for an entire block of data at once, it may also be used in a byte-by-byte or data burst mode. To do so, you would pass the computed CRC value from the previous iteration as the initial value. Note, however, that you shouldn’t use the `zxRefOut` flag bit in this case. Rather, if you need reflected output you would perform the bit reversal on the final CRC value when you reach the end of the data stream. You can reverse the bit order of a 16-bit value by using the following code fragment.

```
crc = MakeWord(FlipBits(HiByte(crc)), FlipBits(LoByte(crc)))
```

CRC algorithms can be described by a parametric model known as the RockSoft model (see [http://www.repairfaq.org/filipg/LINK/F\\_crc\\_v34.html#CRCV\\_005](http://www.repairfaq.org/filipg/LINK/F_crc_v34.html#CRCV_005)). This CRC implementation supports the POLY, INIT, REFIN and REFOUT parameters of the model with WIDTH=16 and XOROUT=0. If necessary, you can easily implement a non-zero XOROUT parameter by using the following code fragment.

```
crc = crc Xor XorOutValue
```

The Rocksoft model parameters for commonly used CRC computations are given in the table below.



**Rocksoft Model Parameters for Common CRC Algorithms**

| <b>Parameter/Type</b> | <b>CRC-16</b> | <b>CRC-CCITT</b> | <b>ModBus</b> | <b>CRC-32</b> |
|-----------------------|---------------|------------------|---------------|---------------|
| WIDTH                 | 16            | 16               | 16            | 32            |
| POLY                  | &H8005        | &H1021           | &H8005        | &H04c11db7    |
| INIT                  | &H0000        | &Hffff           | &Hffff        | &Hffffffff    |
| REFIN                 | True          | False            | True          | True          |
| REFOUT                | True          | False            | True          | True          |
| XOROUT                | &H0000        | &H0000           | &H0000        | &Hffffffff    |
| CHECK                 | &Hbb3d        | &H29b1           | &H4b37        | &Hcbf43926    |

The parameters are included in the table above for the CRC-32 algorithm but, of course, they must be used with the `CRC32()` function. The CHECK value is the CRC result for the string of characters "123456789".

Additional information on CRC calculations may be found in many places on the Internet. One useful site that implements a CRC calculator is <http://www.zorc.breitbandkatze.de/crc.html>. If you don't know the parameters required for a particular CRC, you may be able to deduce the correct parameters by using the calculator if you have a sample message and its CRC value. One of the variables available in the CRC calculator on the web page mentioned is "direct" vs. "nondirect". This implementation uses the "direct" method.

### Example

```
Dim data(1 to 20) as Byte
Dim crc as UnsignedInteger
' compute the CRC using the CRC-16 algorithm
crc = CRC16(data, 10, &H8005, &H0000, zxCRCRefIn Or zxCRCRefOut)
```

### Compatibility

This function is not available in BasicX compatibility mode. Also, on ZX models that are based on the ATmega32 processor (e.g. the ZX-24) this function is implemented in "user code" (as opposed to being part of the VM) and is consequently slower than on other ZX models.

**See Also**      CRC32

# CRC32

**Type** Function returning UnsignedLong

**Invocation** CRC32(data, count, crcPoly, crclnit, crcFlags)

| Parameter | Method | Type         | Description                                 |
|-----------|--------|--------------|---|
| data      | ByRef  | anyType      | The data bytes to add to the CRC value.     |
| count     | ByVal  | integral     | The number of bytes to process.             |
| crcPoly   | ByVal  | UnsignedLong | The CRC polynomial to use.                  |
| crclnit   | ByVal  | UnsignedLong | The initial value of the CRC.               |
| crcFlags  | ByVal  | integral     | Flag bits that control the CRC computation. |

## Discussion

This function computes the CRC value over a number of data bytes using a specified polynomial and initial value. The values to use for the polynomial and the initial value depend on the style of CRC that you need to generate. The `flags` parameter contains bits that control aspects of the CRC computation as described in the table below.

| Flag Values for the CRC Computation |      |           |   |
|-------------------------------------|------|-----------|---|
| Constant                            | Hex  | Binary    | Description                               |
| zxCRCRefIn                          | &H01 | xxxx xxx1 | The input data bytes will be “reflected”. |
| zxCRCRefOut                         | &H02 | xxxx xx1x | The final CRC will be “reflected”.        |

The remaining bits are reserved for future use and should always be zero.

Although this function will typically be used to compute the CRC value for an entire block of data at once, it may also be used in a byte-by-byte or data burst mode. To do so, you would pass the computed CRC value from the previous iteration as the initial value. Note, however, that you shouldn't use the `zxRefOut` flag bit in this case. Rather, if you need reflected output you would perform the bit reversal on the final CRC value when you reach the end of the data stream.

See the discussion of the `CRC16()` function for additional information.

## Example

```
Dim data(1 to 20) as Byte
Dim crc as UnsignedLong
crc = Not CRC32(data, 10, &H04c11db7, &Hfffffff, zxCRCRefIn Or zxCRCRefOut)
```

## Compatibility

This function is not available in BasicX compatibility mode. Also, on ZX models that are based on the ATmega32 processor (e.g. the ZX-24) this function is implemented in “user code” (as opposed to being part of the VM) and is consequently slower than on other ZX models.

**See Also** CRC16

# CSng

**Type**            Function returning Single

**Invocation**    CSng(arg)

| Parameter | Method | Type            | Description                     |
|-----------|--------|-----------------|---------------------------------|
| arg       | ByVal  | numeric or Enum | The value to convert to Single. |

## Discussion

This function converts any numeric or enumeration value to a `Single` value. For integral and `Enum` types, the result will be the floating point approximation of the integral value. If a `Single` type parameter is supplied, the result is identical to the parameter value. If a `String` type parameter is supplied, the result will be the numeric value of the character string. The form of the character representation supported is identical to that supported by `ValueS()`.

## Example

```
Dim b as Byte
Dim f as Single

b = 21
f = CSng(b)
```

## Compatibility

In BasicX, passing an `UnsignedLong` value larger than 2,147,483,647 erroneously generates a negative `Single` result. This implementation handles `UnsignedLong` values correctly.

# CStr

**Type**            Function returning String

**Invocation**    CStr(arg)

| Parameter | Method | Type     | Description                     |
|-----------|--------|----------|---------------------------------|
| arg       | ByVal  | any type | The value to convert to String. |

## Discussion

This function converts any Boolean, numeric or enumeration value to a String value. See the table below for details of the conversion.

| Input Type        | Result  |
|-------------------|---|
| Boolean           | The string "True" or "False".   |
| Byte, Bit, Nibble | A string containing decimal digits representing the value.  |
| Integer           | A string containing decimal digits representing the value. If the value is negative, the string will begin with a minus sign.   |
| UnsignedInteger   | A string containing decimal digits representing the value.  |
| Enum              | A string containing decimal digits representing the Enum member value.  |
| Long              | A string containing decimal digits representing the value. If the value is negative, the string will begin with a minus sign.   |
| UnsignedLong      | A string containing decimal digits representing the value.  |
| Single            | A string representing the value. Depending on the value, the form may be standard decimal form with a decimal point separating the whole and fractional parts or it may be in "scientific notation" form. In some cases, there will be no decimal point at all, e.g. with values having no fractional part. |

When converting `Single` values, some special cases are detected resulting in the strings shown in the table below. See the function `SngClass()` for more information about the special cases.

| Special Value      | Result    |
|--------------------|-----------|
| NaN                | " * . * " |
| ±Infinity          | " & . & " |
| Denormalized value | " # . # " |

## Compatibility

In BasicX the special `Single` values are not handled properly.

**See Also**        CStrHex, Fmt

# CStrHex

**Type**            Function returning String

**Invocation**    CStrHex(arg)

| Parameter | Method | Type    | Description                                   |
|-----------|--------|---------|---|
| arg       | ByVal  | numeric | The value to convert to a hexadecimal String. |

## Discussion

This function converts any Boolean, numeric or enumeration value to a String value. The content of the string will be hexadecimal characters that represent the value of the bytes comprising the passed value. The number of characters in the string varies depending on the type of the value passed. See the table below.

| Input Type                     | Number of Characters |
|--------------------------------|----------------------|
| Boolean, Bit, Nibble, Byte     | 2                    |
| Integer, UnsignedInteger, Enum | 4                    |
| Long, UnsignedLong, Single     | 8                    |

## Compatibility

This function is not available in BasicX compatibility mode.

# CType

**Type**            Function returning an enumeration member

**Invocation**    CType(value, enumType)

| Parameter | Method | Type            | Description                             |
|-----------|--------|-----------------|---|
| value     | ByVal  | numeric or Enum | The value to convert to an Enum member. |
| enumType  | ByVal  | Enum            | The name of the Enum type.              |

## Discussion

This function converts any numeric or enumeration member value to an enumeration member. No checking is done to confirm that the given value actually corresponds to one of the enumeration members.

See the section on enumerations in the ZBasic Reference Manual for more information.

## Example

```
Enum Color
    Red
    Green
    Blue
End Enum

Dim c as Color

c = CType(1, Color)       ' c will have the value Green
```

# CUInt

**Type**            Function returning UnsignedInteger

**Invocation**    CUInt(arg)

| Parameter | Method | Type            | Description                              |
|-----------|--------|-----------------|--|
| arg       | ByVal  | numeric or Enum | The value to convert to UnsignedInteger. |

## Discussion

This function converts any numeric or enumeration value to an UnsignedInteger value. See the table below for details of the conversion.

| Input Type      | Result  |
|-----------------|---|
| Byte, Boolean   | High byte zero, low byte as supplied.   |
| Integer         | Value bits are the same as supplied, although interpreted as an unsigned value.   |
| UnsignedInteger | No effect, the value is as supplied.  |
| Enum            | Resulting value is the Enum member value.   |
| Long            | Resulting value is the low word of the supplied value.  |
| UnsignedLong    | Resulting value is the low word of the supplied value.  |
| Single          | The supplied value is converted to a signed 32-bit integer, rounded to the nearest integer. If the fractional part is exactly 0.5, the resulting integer will be even. This is known as "statistical rounding". If the resulting signed integer is negative or larger than 65535, the result is undefined. Otherwise, the result is the value of the integer.   |
| String          | The result is the numeric value of the characters in the string, ignoring leading space and tab characters. The value string may begin with a plus or minus sign and an optional radix indicator (&H for hexadecimal, &O for octal, &B or &X for binary, all case insensitive). The conversion is terminated upon reaching the end of the string or encountering the first character that is not valid for the indicated radix. |

## Example

```
Dim u as UnsignedInteger

u = CUInt(2.5)        ' result is 2
u = CUInt(1.5)        ' result is 2
```

## Compatibility

The ability to convert from `Single` is not supported in BasicX compatibility mode.

# CULng

**Type**            Function returning UnsignedLong

**Invocation**    CULng(arg)

| Parameter | Method | Type            | Description                           |
|-----------|--------|-----------------|---------------------------------------|
| arg       | ByVal  | numeric or Enum | The value to convert to UnsignedLong. |

## Discussion

This function converts any numeric or enumeration value to an UnsignedLong value. See the table below for details of the conversion.

| Input Type      | Result  |
|-----------------|---|
| Byte, Boolean   | High 3 bytes zero, low byte as supplied.  |
| Integer         | High word will be zero, low word as supplied.   |
| UnsignedInteger | High word will be zero, low word as supplied.   |
| Enum            | High word zero, low word contains Enum member value.  |
| Long            | Value bits are the same as supplied, although interpreted as an unsigned value.   |
| UnsignedLong    | No effect, the value is as supplied.  |
| Single          | Supplied value converted to signed 32-bit integer, rounded to the nearest integer. If the fractional part is exactly 0.5, the resulting integer will be even. This is known as “statistical rounding”. If the supplied value is negative or if it is too large to be represented in 32 bits, the result is undefined.   |
| String          | The result is the numeric value of the characters in the string, ignoring leading space and tab characters. The value string may begin with a plus or minus sign and an optional radix indicator (&H for hexadecimal, &O for octal, &B or &X for binary, all case insensitive). The conversion is terminated upon reaching the end of the string or encountering the first character that is not valid for the indicated radix. |

## Example

```
Dim ul as UnsignedLong
```

```
ul = CULng(2.5)     ' result is 2  
ul = CULng(1.5)     ' result is 2
```



# DACPin

**Type** Subroutine

**Invocation** DACPin(pin, dacValue, dacAccumulator)

| Parameter      | Method | Type | Description   |
|----------------|--------|------|---|
| pin            | ByVal  | Byte | The pin to which the DAC signal will be output.                             |
| dacValue       | ByVal  | Byte | The value representing the desired analog output. See the discussion below. |
| dacAccumulator | ByRef  | Byte | A value used in the DAC process. See discussion below.                      |

## Discussion

This routine creates a digital approximation of an analog signal on the specified pin using a pseudo-PWM technique. ZBasic supports this routine for backward compatibility. New applications should use `PutDAC()` as it is more flexible. See the description of `PutDAC()` for more information.

## Resource Usage

This routine disables interrupts for approximately 200µS during the generation process.

**See Also** PutDAC

# Debug.Print

**Type** Special Purpose

**Invocation** Debug.Print stringList

| Parameter  | Method | Type   | Description                                     |
|------------|--------|--------|---|
| stringList | ByVal  | String | One or more strings or values to send out Com1. |

## Discussion

Debug.Print is neither a subroutine nor a function. It has more in common with ZBasic statements but it is described here for ease of reference. This special purpose method is useful for outputting debugging information and other data to Com1. The arguments provided to the command consist of zero or more strings or values each separated by a semicolon. If non-string values are supplied, they are converted to strings automatically using the CStr() function. Unless the list ends with a semicolon, a carriage return/new line will also be output after all of the strings have been output.

When this statement is invoked, execution of the current task will not continue and no other task will be allowed to run until the string's characters have been transferred to the system output queue. This caveat applies independently to each string in the semicolon-separated list as well as to the end-of-line string, if applicable. The latency-inducing effect described above can be mitigated by preparing a new output queue that is sufficiently large such that there is always enough free space in the queue when this method is invoked. See the example below.

In contrast to other System Library routines that copy data to a queue, the string length is not limited to the system output queue length.

## Examples

```
Debug.Print "Hello, world! "
```

This prints the given string followed by a carriage return/new line.

```
Debug.Print "The value is ";CStr(val);
```

This prints the string followed immediately by the string equivalent of the value. Note that since the command ends with a semicolon, no carriage return/new line will be generated.

```
Dim iq(1 to 20) as Byte
Dim oq(1 to 100) as Byte
Call OpenQueue(iq, SizeOf(iq))
Call OpenQueue(oq, SizeOf(oq))
Call OpenCom(1, 19200, iq, oq)
```

This example code shows how to increase the size of the output queue in order to reduce latency. The default input could be retained by replacing the last line above with the following line and deleting the other lines that refer to the variable iq.

```
Call OpenCom(1, 19200, CByteArray(Register.RxQueue), oq)
```

**See Also** Console.Write, Console.WriteLine

# DefineBus

**Type** Subroutine

**Invocation** DefineBus(port, alePin, rdPin, wrPin)

| Parameter | Method | Type     | Description  |
|-----------|--------|----------|--|
| port      | ByVal  | integral | The port to use for address and data. PortA=0, PortB=1, etc. |
| alePin    | ByVal  | integral | The pin to use for the address latch strobe.                 |
| rdPin     | ByVal  | integral | The pin to use for the read data strobe.                     |
| wrPin     | ByVal  | integral | The pin to use for the write data strobe.                    |

## Discussion

This subroutine is used to define the parameters to use for subsequent BusRead() and BusWrite() operations. The port specified by the `port` parameter is used both for outputting the address from which to read/write and for reading/writing the data. The port is specified by giving a port index – PortA = 0, PortB = 1, etc. You may use the built-in constants `Port.A`, `Port.B`, etc. to specify the port index. If all the parameters are valid, the pin specified by the `alePin` parameter is set to output low while the pins specified by the `rdPin` and `wrPin` parameters are set to output high. If any of the provided parameters is invalid, the bus will not be properly configured and subsequent calls to `BusRead()` or `BusWrite()` will return immediately with no effect.

The pin numbers specified for the `alePin`, `rdPin` and `wrPin` parameters must all be different and none of them should be in the port specified by the `port` parameter. If these conditions are violated, the result is undefined.

## Example

```
Call DefineBus(Port.A, C.0, C.1, C.2)
```

## Compatibility

This subroutine is not available on ZX models that are based on the ATmega32 processor (e.g. the ZX-24). Moreover, it is not available in BasicX compatibility mode.

**See Also** BusRead, BusWrite

# DefineCom

**Type** Subroutine

**Invocation** DefineCom(channel, rxPin, txPin, flags)  
DefineCom(channel, rxPin, txPin, flags, stopBits)

| Parameter | Method | Type | Description                                    |
|-----------|--------|------|--|
| channel   | ByVal  | Byte | The serial channel being defined.              |
| rxPin     | ByVal  | Byte | The pin which will serve as the receive line.  |
| txPin     | ByVal  | Byte | The pin which will serve as the transmit line. |
| flags     | ByVal  | Byte | Configuration flags. See the discussion below. |
| stopBits  | ByVal  | Byte | The desired number of stop bits.               |

## Discussion

This routine configures a serial channel, preparing it to be opened using `OpenCom()`. If the specified channel is already open, this routine does nothing. Likewise, there is no effect if the specified channel is invalid (see the `ComChannels()` routine) or if either of the `rxPin` and `txPin` parameters are invalid or both are zero. Note that either `rxPin` or `txPin` may be zero, allowing you to define a transmit-only or receive-only serial channel.

If the specified channel is a hardware UART (e.g. `Com1`), the `rxPin` and `txPin` parameters must both be zero. Otherwise, if they are valid, the pins specified by `rxPin` and `txPin` are automatically configured as input and output, respectively.

The `flags` parameter contains several bit fields used to specify some of the details of the operation of the serial channel.

**Serial Channel Configuration Flag Values**

| Function                    | Hex Value | Bit Mask    |
|-----------------------------|-----------|-------------|
| Inverted Logic <sup>1</sup> | &H80      | 1x xx xx xx |
| Non-inverted Logic          | &H00      | 0x xx xx xx |
| Ignore Parity Bit           | &H40      | x1 xx xx xx |
| Even Parity                 | &H30      | xx 11 xx xx |
| Odd Parity                  | &H20      | xx 10 xx xx |
| No Parity                   | &H00      | xx 00 xx xx |
| 7-bit Data                  | &H07      | xx xx 01 11 |
| 8-bit Data                  | &H08      | xx xx 10 00 |

<sup>1</sup> Applicable only to software-based channels (3-6).

The remaining bits are currently undefined but may be employed in the future.

When Non-inverted Logic is selected, the idle state of the transmit line will be logic 1. When a character transmission is begun, a “start bit” of logic zero will be sent for one bit time (the inverse of the baud rate). Next the data bits are sent, each for one bit time, beginning with the least significant bit and continuing through the eighth data bit or parity bit as the case may be. Finally, one or more “stop bits” of logic one are sent, each for one bit time. With Inverted Logic, each of these elements is complemented – the idle state of the transmit line is logic 0.

Whether you should choose the Inverted or Non-inverted mode depends on the device that you intend to communicate with and how many, if any, level converters exist between the two devices. Typically, if the other device is capable of sending and receiving TTL-level serial data, you’ll likely choose Non-inverted Logic.

If the “Ignore Parity” flag is asserted, in 7-bit mode the most significant bit of each character received will be zero and in 8-bit mode only one byte will be stored in the queue for each character received. If the “Ignore Parity” bit is not asserted, in 7-bit mode the MSB will contain the received parity bit and in 8-bit mode a second byte containing the parity bit will be stored in the queue for each character received. The `ParityCheck()` function is useful for checking the parity of a received character.

If the optional `stopBits` parameter is not specified, one stop bit is transmitted for each character sent. Otherwise, the specified number of stop bits is transmitted. The allowable range for `stopBits` is 1 to 240. If a value outside this range is specified, the default of 1 stop bit will be used. The ability to specify two or more stop bits is useful for slowing down the transmission of data in cases where the receiver needs additional time to process received data.

Note that a pullup resistor (Non-inverted mode) or a pulldown resistor (Inverted mode) is recommended on the transmit line to force the transmit line to the idle state prior to the time your program initializes the COM port. If you don't do this, the receiving device may see false transmissions prior to the first character actually transmitted. Depending on what other circuitry is connected to the receive line, you may need to do the same to prevent the ZX from receiving false transmissions.

This subroutine may be used to specify the data width, parity mode and stop bits for a hardware UART channel (e.g. Com1) provided that it is called when the channel is closed. When used this way, the `txPin` and `rxPin` parameters are ignored and values of 2 or more for the `stopBits` parameter will select 2 stop bits. Also, the flag for inverted data mode is likewise ignored.

### Example

```
Call ComChannels(2, 9600)
Call DefineCom(4, 0, 12, &H08)
```

This call prepares channel 4 for transmit-only using pin 12, eight data bits, no parity and Non-inverted Logic.

### Compatibility

This function is not available in BasicX compatibility mode; you must use `DefineCom3()`. Additionally, BasicX does not support 8-bit plus parity modes nor does it support the “Strip Parity” mode. Furthermore, in BasicX characters received in 7-bit/no parity mode are aligned toward the MSB while in this implementation they are properly aligned toward the LSB.

The ability to define the characteristics of Com1 is not available on mega32-based devices such as the ZX-24.

**See Also**      `ComChannels`, `OpenCom`, `StatusCom`

# DefineCom3

---

**Type**            Subroutine

**Invocation**    DefineCom3(rxPin, txPin, flags)

| Parameter | Method | Type | Description                                    |
|-----------|--------|------|--|
| rxPin     | ByVal  | Byte | The pin which will serve as the receive line.  |
| txPin     | ByVal  | Byte | The pin which will serve as the transmit line. |
| flags     | ByVal  | Byte | Configuration flags. See the discussion below. |

## Discussion

This routine is provided solely for BasicX compatibility. It is equivalent to using `Call DefineCom(3, rxPin, txPin, flags)`. See the `DefineCom()` routine for more information.

# DefineX10

**Type** Subroutine

**Invocation** DefineX10(channel, rxPin, txPin, flags)

| Parameter | Method | Type | Description   |
|-----------|--------|------|---|
| channel   | ByVal  | Byte | The X-10 channel being defined. The valid range is 1-2. |
| rxPin     | ByVal  | Byte | The pin which will serve as the receive line.           |
| txPin     | ByVal  | Byte | The pin which will serve as the transmit line.          |
| flags     | ByVal  | Byte | Configuration flags. See the discussion below.          |

## Discussion

This routine configures an X-10 communication channel, preparing it to be opened using `OpenX10()`. If the specified channel is already open, this routine does nothing. Likewise if the specified channel is invalid or if both the `rxPin` and `txPin` parameters are zero or invalid. Note that either `rxPin` or `txPin` may be zero, allowing you to define a transmit-only or a receive-only X-10 channel. If valid, the pins specified by `rxPin` and `txPin` are automatically configured as input and output, respectively.

The `flags` parameter contains several bit fields used to specify some of the details of the operation of the X-10 channel.

### Configuration Flags Bit Values

| Function                     | Hex Value | Bit Mask    |
|------------------------------|-----------|-------------|
| LSB-first Transmit Bit Order | &H08      | xx xx 1x xx |
| MSB-first Transmit Bit Order | &H00      | xx xx 0x xx |
| Inverted Transmit Logic      | &H04      | xx xx x1 xx |
| Non-inverted Transmit Logic  | &H00      | xx xx x0 xx |
| LSB-first Receive Bit Order  | &H02      | xx xx xx 1x |
| MSB-first Receive Bit Order  | &H00      | xx xx xx 0x |
| Inverted Receive Logic       | &H01      | xx xx xx x1 |
| Non-inverted Receive Logic   | &H00      | xx xx xx x0 |

The remaining bits are currently undefined but may be employed in the future.

When non-inverted modes are selected, the idle state of the transmit line or receive line will be logic 0. Whether you should choose the inverted or non-inverted mode depends on the interface circuitry that you use to connect to your X-10 transmitter/receiver.

When LSB-first modes are selected, the first bit to be sent/received will be the least significant bit of each byte. This is useful when a Bit array is used to assemble/decompose the data that is sent/received since the lower-indexed bits in a byte are of lower significance.

## Example

```
Call DefineX10(1, 0, 12, &H00)
```

This call prepares channel 1 for transmit-only using pin 12, non-inverted logic, MSB-first operation.

## Compatibility

This subroutine is not available on ZX models that are based on the ATmega32 processor (e.g. the ZX-24). Moreover, it is not available in BasicX compatibility mode.

**See Also** CloseX10, OpenX10, StatusX10

# DegToRad

---

**Type**            Function returning Single

**Invocation**    DegToRad(angle)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| angle     | ByVal  | Single | The angle, in degrees, to convert to radian measure. |

## Discussion

The trigonometric functions in the System Library all use radian angle measure. Depending on the programming task, it is sometimes more convenient to think of angles in terms of degrees. This function and its companion `RadToDeg( )` facilitate the conversion between the two systems.

Depending on optimization settings, if the parameter supplied to this function is known to be constant at compile time, the compiler converts the value at compile time. Otherwise, code is generated to perform the conversion (multiplication by a conversion factor) at run time.

## Example

```
Dim f as Single
Dim theta as Single     ' the angle in degrees

f = Sin(DegToRad(theta))
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        RadToDeg



# Delay

**Type** Subroutine

**Invocation** Delay(time)

| Parameter | Method | Type   | Description                              |
|-----------|--------|--------|--|
| time      | ByVal  | Single | The amount of time to delay, in seconds. |

## Discussion

This routine suspends the current task for a period of time at least as long as specified. The actual delay depends on what other tasks actually do that may run in the interim. It is possible that the task will be suspended indefinitely depending on what another task might do.

Note that if the current task is locked, this call will unlock it.

There is a subtle difference between `Delay()` and `Sleep()` when the arguments are non-zero. For `Delay()` the specified time is the minimum amount of delay that the task will experience assuming that no other task is ready to run. The actual delay could be up to 1.95ms longer than the specified delay. For `Sleep()`, the specified time is the maximum amount of delay that the task will experience assuming that no other task is ready to run. The actual delay could be up to 1.95ms less than the specified delay.

## Example

```
Do
    Call PutPin(25, 0)
    Call Delay(0.5)
    Call PutPin(25, 1)
    Call Delay(0.5)
Loop
```

This loop causes the red LED to turn on and off alternately for a half second each.

## Compatibility

The BasicX documentation specifically indicates that `Delay()` will unlock a locked task. However, tests indicate that this only happens if the parameter to `Delay()` is non-zero. This implementation unlocks a task on any `Delay()` call.

**See Also** DelayUntilClockTick, Pause, Sleep, Register.RTCStopWatch

# DelayUntilClockTick

---

**Type**            Subroutine

**Invocation**    DelayUntilClockTick()

## Discussion

This routine suspends the current task until at least the next tick of the RTC. The actual delay depends on what other tasks actually do that may run in the interim. It is possible that the task will be suspended indefinitely.

If no other tasks are ready to run, the actual delay could be between 0 and 1.95ms.

This routine is exactly equivalent to `Sleep(1)`.

**See Also**        Delay, Pause, Sleep

# DisableInt

---

**Type**                      Function returning Byte

**Invocation**            DisableInt()

## Discussion

This routine disables interrupts, preventing any interrupt source from interrupting the current task. Most commonly, this function is used to temporarily disable interrupts thereby allowing a sequence of instructions to execute without interruption. Of course, interrupts should be disabled for the shortest possible time in order to avoid missing important interrupts (e.g. real time clock interrupts). If interrupts are disabled for longer than one period of the RTC fast tick (typically 976 uS) you run the risk of missing an RTC tick which will result in the RTC losing time.

The most common use for DisableInt() is to implement “atomic access” to variables. This should be done for any variable that occupies multiple bytes of memory (e.g. Integer, Long, etc.) or for a read-modify-write operation on any variable when there is a possibility that another task or interrupt handler might attempt to access the same variable.

The value returned by DisableInt() should be passed to EnableInt(). Doing so will allow proper nesting of DisableInt() and EnableInt() calls.

## Note

The Atomic block construct (described in the ZBasic Language Reference Manual) is the preferred method for implementing atomic access.

## Example

```
Dim iflag as Byte

iflag = DisableInt()
' place code here that must not be interrupted
Call EnableInt(iflag)
```

**See Also**                EnableInt, UpdateRTC, Yield

# EnableInt

**Type** Subroutine

**Invocation** EnableInt(flag)

| Parameter | Method | Type | Description                                      |
|-----------|--------|------|--|
| flag      | ByVal  | Byte | The value controlling re-enabling of interrupts. |

## Discussion

This routine conditionally re-enables interrupts depending on the value of the `flag` parameter. If the most significant bit of the `flag` parameter is a 1, interrupts will be re-enabled. Otherwise, the state of the interrupt enabling will not change. Passing the value returned from `DisableInt()` implements proper nesting of `DisableInt()` and `EnableInt()` calls so they are most often used in pairs as shown in the example below.

## Note

The Atomic block construct (described in the ZBasic Language Reference Manual) is the preferred method for implementing atomic access.

## Example

```
Dim iflag as Byte

iflag = DisableInt()
' place code here that must not be interrupted
Call EnableInt(iflag)
```

**See Also** DisableInt, UpdateRTC, Yield

# ExitTask

---

**Type** Subroutine

**Invocation** ExitTask(taskStack)  
ExitTask()

| Parameter | Method | Type          | Description                       |
|-----------|--------|---------------|-----------------------------------|
| taskStack | ByRef  | array of Byte | The stack for a task of interest. |

## Discussion

This routine attempts to terminate an active task. If no task stack is explicitly given, the task stack for the `Main()` routine is assumed.

If this routine is invoked using an array other than one that is or was being used for a task stack the result is undefined.

See the section on Task Management in the ZBasic Reference Manual for additional information regarding task management.

When a task exits, whether normally or via `ExitTask()`, that task's status is first set to 254 indicating that it is in the process of exiting but that it is still in the task list. The exiting task will remain in the task list until the task manager runs again. The task manager runs whenever a task switch is called for but you can force it to run by invoking `Sleep()` or `Yield()`. Once the task manager removes an exiting task from the task list, its status will change to 255 indicating that it is fully terminated.

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** ResumeTask, RunTask, StatusTask

# Exp

---

**Type**            Function returning Single

**Invocation**    Exp(arg)

| Parameter | Method | Type   | Description                    |
|-----------|--------|--------|--------------------------------|
| arg       | ByVal  | Single | The power of e to be computed. |

## Discussion

This function returns the `Single` value corresponding to the value `e` raised to the specified power. The transcendental value `e`, upon which the natural logarithm is based, is approximately 2.718. This function is the inverse of the `Log( )` function.

**See Also**        Exp10, Log, Log10, Pow

# Exp10

---

**Type**              Function returning Single

**Invocation**      Exp10(arg)

| Parameter | Method | Type   | Description                     |
|-----------|--------|--------|---------------------------------|
| arg       | ByVal  | Single | The power of 10 to be computed. |

## Discussion

This function returns the `Single` value corresponding to the value 10 raised to the specified power. This function is the inverse of the `Log10()` function.

**See Also**              Exp, Log, Log10, Pow

# FirstTime

---

**Type**            Function returning Boolean

**Invocation**    FirstTime()

## Discussion

When called the first time after downloading a program, this function will return True. Thereafter, it will always return False even if the processor is powered down or reset. Subsequently downloading again will again cause the function to return True on the first call, etc.



# Fix

---

**Type**            Function returning Single

**Invocation**    Fix(arg)

| Parameter | Method | Type   | Description              |
|-----------|--------|--------|--------------------------|
| arg       | ByVal  | Single | The value to be “fixed”. |

## Discussion

This function returns the `Single` representation of the integer that is nearest the supplied value, rounding toward zero.

## Example

```
Dim f as Single
```

```
f = Fix(1.5)            ' result is 1.0  
f = Fix(-1.5)          ' result is -1.0
```

**See Also**        Ceiling, Floor, Fraction

# FixB

**Type**            Function returning Byte

**Invocation**    FixB(arg)

| Parameter | Method | Type   | Description                               |
|-----------|--------|--------|---|
| arg       | ByVal  | Single | The value to be changed to integral form. |

## Discussion

The supplied `Single` value is first converted to a signed 32-bit integer, rounding toward zero, and then the low 8 bits of that value is returned. The result isn't particularly useful if the provided `Single` value is negative or larger than 255.

## Example

```
Dim b as Byte
```

```
b = FixB(100.5)                    ' result is 100
```

# FixI

**Type**            Function returning Integer

**Invocation**    FixI(arg)

| Parameter | Method | Type   | Description                               |
|-----------|--------|--------|---|
| arg       | ByVal  | Single | The value to be changed to integral form. |

## Discussion

The supplied `Single` value is first converted to a signed 32-bit integer, rounding toward zero, and then the low 16 bits of that value is returned. The result isn't particularly useful if the provided `Single` value is outside the range  $-32768$  to  $32767$ , inclusive.

## Example

```
Dim i as Integer  
  
i = FixI(-100.5)                    ' result is -100
```

## Compatibility

For compatibility with BasicX, if the provided `Single` value is larger than  $32767$  this function returns  $32767$ . Similarly, if the value is less than  $-32767$  (not  $-32768$  as one would expect) this function returns  $-32767$ .

# FixL

**Type**            Function returning Long

**Invocation**    FixL(arg)

| Parameter | Method | Type   | Description                               |
|-----------|--------|--------|---|
| arg       | ByVal  | Single | The value to be changed to integral form. |

## Discussion

The supplied `Single` value is converted to a signed 32-bit integer, rounding toward zero, and that value is returned. The result isn't particularly useful if the provided `Single` value is outside the range – 2,147,485,648 to 2,147,485,647, inclusive.

## Example

```
Dim l as Long
```

```
l = FixL(-100.5)                    ' result is -100
```

# FixUI

**Type**            Function returning `UnsignedInteger`

**Invocation**    `FixUI(arg)`

| Parameter        | Method             | Type                | Description                               |
|------------------|--------------------|---------------------|---|
| <code>arg</code> | <code>ByVal</code> | <code>Single</code> | The value to be changed to integral form. |

## Discussion

The supplied `Single` value is first converted to a signed 32-bit integer, rounding toward zero, and then the low 16 bits of that value is returned. The result isn't particularly useful if the provided `Single` value is outside the range 0 to 65535, inclusive.

## Example

```
Dim ui as UnsignedInteger

ui = FixUI(100.5)            ' result is 100
```

# FixUL

**Type**            Function returning UnsignedLong

**Invocation**    FixUL(arg)

| Parameter | Method | Type   | Description                               |
|-----------|--------|--------|---|
| arg       | ByVal  | Single | The value to be changed to integral form. |

## Discussion

The supplied `Single` value is converted to a signed 32-bit integer, rounding toward zero, and that value is returned. The result isn't particularly useful if the provided `Single` value is outside the range 0 to 4,294,967,295, inclusive.

## Example

```
Dim ul as UnsignedLong

ul = FixUL(100.5)            ' result is 100
```

# FlipBits

---

**Type**            Function returning Byte

**Invocation**    FlipBits(arg)

| Parameter | Method | Type | Description                        |
|-----------|--------|------|------------------------------------|
| arg       | ByVal  | Byte | The value to be bit-wise reversed. |

## Discussion

This function reverses the order of the bits in the supplied value and returns the result. This is useful, for example, if you want to send data using `ShiftOut()` but you want the least significant bit to be sent first.

## Example

```
Dim b as Byte

b = &B1011_0110
b = FlipBits(b)      ' result is &B0110_1101
```

# Floor

---

**Type**            Function returning Single

**Invocation**    Floor(arg)

| Parameter | Method | Type   | Description                              |
|-----------|--------|--------|--|
| arg       | ByVal  | Single | The value of which to compute the floor. |

## Discussion

This function returns a `Single` value that is equal to the largest integer that is less than or equal to the supplied value, effectively rounding down to the nearest integer.

## Example

```
Dim flr as Single

flr = Floor(1.5)    ' result is 1.0
flr = Floor(-1.5)  ' result is -2.0
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        Ceiling, Fix



# Fmt

**Type**            Function returning String

**Invocation**     Fmt(val, fracDigits)

| Parameter  | Method | Type   | Description  |
|------------|--------|--------|--|
| val        | ByVal  | Single | The value to convert to a string.                            |
| fracDigits | ByVal  | Byte   | The number of digits to produce following the decimal point. |

## Discussion

This function returns a `String` that represents the value of the `val` parameter. The string will have a number of digits following the decimal point as specified by the `fracDigits` parameter. The maximum number of digits to the right of the decimal point is 6. If the `fracDigits` parameter specifies a larger number, it will be ignored and 6 will be used.

For very large and very small values, the returned string may be in scientific notation form. Also, some special cases are detected resulting in the strings shown in the table below. See the System Library function `SngClass()` for more information about the special values.

| Special Value      | Result <sup>1</sup> |
|--------------------|---------------------|
| NaN                | " * . ** "          |
| ±Infinity          | " & . && "          |
| Denormalized value | " # . ## "          |

<sup>1</sup>The number of special characters following the decimal point will be the same as the number of fraction digits that would have been generated had the value been normal.

## Compatibility

In BasicX, the maximum number of fraction digits is 3 and the valid range of the value parameter is –999.0 to +999.0. If either of those ranges is exceeded, BasicX produces a string containing a single asterisk. Moreover, no provision is made for detecting special values such as NaN.

# Fraction

---

**Type**            Function returning Single

**Invocation**    Fraction(val)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| val       | ByVal  | Single | The value from which the fractional part will be returned. |

## Discussion

This function returns the fractional portion of the supplied value. The sign of the returned value will be the same as that of the value provided.

## Example

```
Dim frac as Single

frac = Fraction(1.5)      ' result is 0.5
frac = Fraction(-1.5)    ' result is -0.5
```

## Compatibility

This function is not available in BasicX compatibility mode.

# FreqOut

**Type** Subroutine

**Invocation** FreqOut(pin, freqA, freqB, duration)

| Parameter | Method | Type              | Description   |
|-----------|--------|-------------------|---|
| pin       | ByVal  | Byte              | The pin on which the signal will be created.  |
| freqA     | ByVal  | Integer           | The primary frequency, in Hertz.  |
| freqB     | ByVal  | Integer           | The secondary frequency, in Hertz.  |
| duration  | ByVal  | Single or Integer | The duration of the signal, in seconds or units. See the discussion below for more details. |

## Discussion

This routine generates a signal on the specified pin that is a digital approximation of two superimposed sine waves having the specified frequencies. The method used to produce the signal is a pseudo-PWM technique similar to that used for `DACPin()`. The output signal is actually purely digital, consisting of a series of precisely timed pulses that have an average value approximating that of two superimposed sine waves. This signal must be filtered to get an analog approximation. Depending on what you want to do with the signal, it may need to be amplified as well.

The duration of the signal may be specified in seconds by providing a `Single` value. Alternately, the time may be specified in units of approximately 1 millisecond by giving duration as an `Integer` or `UnsignedInteger` value. In either case, the valid range is approximately 1ms to 32 seconds.

Before beginning the frequency generation, the specified pin will be made an output. When the routine returns, the pin will still be an output.

If the pin is invalid, or both frequencies are zero, or the duration is zero, this routine does nothing. The maximum frequency that can be produced is approximately 14.4KHz. Requesting higher frequencies will produce undefined results.

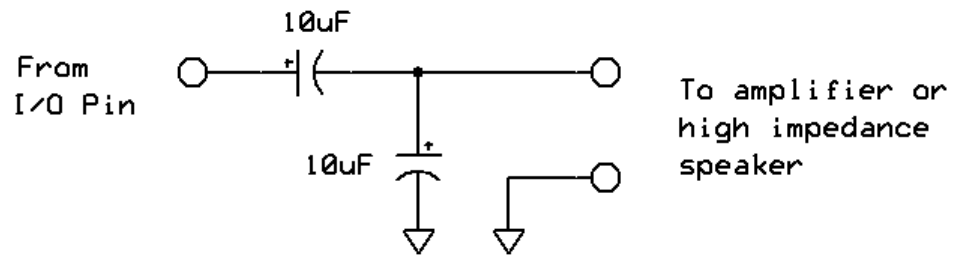
## Resource Usage

This routine uses the I/O Timer and disables interrupts until the signal generation is completed. RTC ticks are accumulated during the process so long signal durations should not cause a loss in RTC accuracy.

## Example

```
Call FreqOut(pin, 440, 880, 5.0) ' play middle C/high C for 5 seconds
```

Because of the high frequency nature of the pulse train used to synthesize the waveform some filtering is required. The example circuit below may be used to couple the output to a high impedance speaker (> 40  $\Omega$ ) or an amplifier. Note, however, that the signal is too large to be fed to the microphone input of an amplifier. Instead, the Auxiliary or Line input should be used.



### Compatibility

In BasicX, the RTC will lose time if the duration is longer than 1 millisecond. Also, the duration is documented as being limited to about 2.5 seconds

# Get1Wire

---

**Type**              Function returning Byte

**Invocation**      Get1Wire(pin)

| Parameter | Method | Type | Description                        |
|-----------|--------|------|------------------------------------|
| pin       | ByVal  | Byte | The pin to be used for 1-Wire I/O. |

## Discussion

This function retrieves a single bit using the 1-Wire protocol. To perform a 1-Wire operation, this function along with related 1-Wire routines must be used in the proper sequence. See the specifications of your 1-Wire device for more information.

The value returned will be either 0 or 1.

## Resource Usage

This routine uses the I/O Timer and disables interrupts for approximately 100µS.

## Example

```
Dim b as Byte  
  
b = Get1Wire(12)
```

**See Also**              Get1WireByte, Get1WireData, Put1Wire,  
Put1WireByte, Put1WireData, Reset1Wire

# Get1WireByte

---

**Type**                Function returning Byte

**Invocation**        Get1WireByte(pin)

| Parameter | Method | Type | Description                        |
|-----------|--------|------|------------------------------------|
| pin       | ByVal  | Byte | The pin to be used for 1-Wire I/O. |

## Discussion

This function reads a byte value (LSB first) using the 1-Wire protocol. It may be used instead of a series of calls to `Get1Wire()` in order to read a byte at a time. To perform a 1-Wire operation, this function along with related 1-Wire routines must be used in the proper sequence. See the specifications of your 1-Wire device for more information.

## Resource Usage

This routine uses the I/O Timer and disables interrupts for about 100µS for each bit received.

## Example

```
Dim b as Byte  
  
b = Get1WireByte(12)
```

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also**            Get1Wire, Get1WireData, Put1Wire,  
Put1WireByte, Put1WireData, Reset1Wire

# Get1WireData

---

**Type** Subroutine

**Invocation** Get1WireData(pin, data, count)

| Parameter | Method | Type     | Description                             |
|-----------|--------|----------|---|
| pin       | ByVal  | Byte     | The pin to be used for 1-Wire I/O.      |
| data      | ByRef  | any type | The variable to receive the bytes read. |
| count     | ByVal  | Byte     | The number of bytes to read.            |

## Discussion

This function retrieves 1 or more bytes (each LSB first) using the 1-Wire protocol and writes them to the given variable. To perform a 1-Wire operation, this function along with related 1-Wire routines must be used in the proper sequence. See the specifications of your 1-Wire device for more information.

## Caution

If the variable provided has fewer bytes than the given count, subsequent memory locations will be altered, usually with undesirable consequences.

## Resource Usage

This routine uses the I/O Timer and disables interrupts for about 100µS for each bit received.

## Example

```
Dim ba(1 to 10) as Byte  
  
Call Get1WireData(12, ba, SizeOf(ba))
```

**See Also** Get1Wire, Get1WireByte, Put1Wire, Put1WireByte, Put1WireData, Reset1Wire

# GetADC (subroutine form)

**Type** Subroutine

**Invocation** GetADC(pin, val)

| Parameter | Method | Type   | Description                                   |
|-----------|--------|--------|---|
| pin       | ByVal  | Byte   | The pin from which to read an analog voltage. |
| val       | ByRef  | Single | The variable in which to return the result.   |

## Discussion

This function performs an analog-to-digital conversion on the signal present on the specified pin which must be one of the analog port pins (see the table below). The return value will be a 10-bit digital approximation of the input voltage with a range from zero to the AVcc reference voltage (usually +5 volts) scaled to the range 0.0 to 1.0.

You must make the pin an input before calling this routine.

The conversion is performed using the AVcc reference voltage (connected internally to Vcc on the ZX-24, ZX-24a, ZX-24p, ZX-24n, ZX, ZX-24ae, ZX-24pe, ZX-24ne, ZX-128e, ZX-128ne, ZX-1281e and ZX-1281ne).

## Resource Usage

The ZX processors contain a single analog-to-digital converter thus allowing only one conversion to be performed at a time. The conversion process takes approximately 220uS during which time the calling task will be awaiting conversion completion.

Only analog port pins may be used to perform an analog-to-digital conversion. The analog port pins vary depending on the ZX model and some ZX models have more analog input pins available.

**Analog Ports and Pins**

| ZX Models                              | Port  | Pins  | Port  | Pins  |
|--|-------|-------|-------|-------|
| ZX-24, ZX-24a, ZX-24p, ZX-24n          | PortA | 13-20 | -     | -     |
| ZX-40, ZX-40a, ZX-40p, ZX-40n          | PortA | 33-40 | -     | -     |
| ZX-44, ZX-44a, ZX-44p, ZX-44n          | PortA | 30-37 | -     | -     |
| ZX-1281, ZX-1281n                      | PortF | 54-61 | -     | -     |
| ZX-1280, ZX-1280n                      | PortF | 90-97 | PortK | 82-89 |
| ZX-24e, ZX-24ae, ZX-24pe, ZX-24ne      | PortA | 29-36 | -     | -     |
| ZX-128e, ZX-128ne, ZX-1281e, ZX-1281ne | PortF | 29-36 | -     | -     |

## Compatibility

Although the BasicX manual indicates that that it is not necessary to configure the pin to be an input before calling, tests indicate that it is, in fact, necessary to do so. Consequently, the behavior of this implementation matches the actual behavior of the BasicX platform.



## GetADC (function form)

**Type**                Function returning Integer

**Invocation**        GetADC(pin)

| Parameter | Method | Type | Description                                   |
|-----------|--------|------|---|
| pin       | ByVal  | Byte | The pin from which to read an analog voltage. |

### Discussion

This function performs an analog-to-digital conversion of the voltage present on the specified pin which must be one of the analog port pins (see the table below). The return value will be a 10-bit digital approximation of the input voltage with a range from zero to the AVcc reference voltage (usually +5 volts). The return value represents the measured voltage according to the formula  $V_{ref} * adcVal / 1024$  where  $V_{ref}$  is the AVcc reference voltage and  $adcVal$  is the value returned by GetADC().

You must make the specified pin an input before calling this routine.

The conversion is performed using the AVcc reference voltage (connected internally to Vcc on the ZX-24, ZX-24a, ZX-24p, ZX-24n, ZX-24e, ZX-24ae, ZX-24pe, ZX-24ne, ZX-128e, ZX-128ne, ZX-1281e and ZX-1281ne).

### Resource Usage

The ZX processors contain a single analog-to-digital converter thus allowing only one conversion to be performed at a time. The conversion process takes approximately 220uS during which time the calling task will be wait for conversion completion.

Only analog port pins may be used to perform an analog-to-digital conversion. The analog port pins vary depending on the ZX model and some ZX models have more analog input pins available.

**Analog Ports and Pins**

| ZX Models                              | Port  | Pins  | Port  | Pins  |
|--|-------|-------|-------|-------|
| ZX-24, ZX-24a, ZX-24p, ZX-24n          | PortA | 13-20 | -     | -     |
| ZX-40, ZX-40a, ZX-40p, ZX-40n          | PortA | 33-40 | -     | -     |
| ZX-44, ZX-44a, ZX-44p, ZX-44n          | PortA | 30-37 | -     | -     |
| ZX-1281, ZX-1281n                      | PortF | 54-61 | -     | -     |
| ZX-1280, ZX-1280n                      | PortF | 90-97 | PortK | 82-89 |
| ZX-24e, ZX-24ae, ZX-24pe, ZX-24ne      | PortA | 29-36 | -     | -     |
| ZX-128e, ZX-128ne, ZX-1281e, ZX-1281ne | PortF | 29-36 | -     | -     |

### Compatibility

Although the BasicX manual indicates that that it is not necessary to configure the pin to be an input before calling, tests indicate that it is, in fact, necessary to do so. Consequently, the behavior of this implementation matches the actual behavior of the BasicX platform.

# GetBit

---

**Type**                Function returning `Byte`

**Invocation**        `GetBit(var, bitNumber)`

| Parameter              | Method             | Type                 | Description                                   |
|------------------------|--------------------|----------------------|---|
| <code>var</code>       | <code>ByRef</code> | any type             | The variable from which the bit will be read. |
| <code>bitNumber</code> | <code>ByVal</code> | <code>int8/16</code> | The bit number to read.                       |

## Discussion

This function extracts a single bit from memory beginning at the location of the specified variable. Bit numbers 0-7 are taken from the byte at the specified location, bit numbers 8-15 are taken from the subsequent byte, etc. In each case, the lower bit number corresponds to the least significant bit of the byte while the higher bit number corresponds to the most significant bit.

The return value will always be 0 or 1.

## Compatibility

In BasicX compatibility mode the second parameter must be a `Byte` type.

**See Also**            `PutBit`

# GetDate

---

**Type** Subroutine

**Invocation** GetDate(year, month, day)  
GetDate(year, month, day, dayNum)

| Parameter | Method | Type     | Description  |
|-----------|--------|----------|--|
| year      | ByRef  | int16    | The variable in which to place the year value (1999-2177). |
| month     | ByRef  | Byte     | The variable in which to place the month value (1-12).     |
| day       | ByRef  | Byte     | The variable in which to place the day value (1-31).       |
| dayNum    | ByVal  | integral | The day number to convert to year, month, day.             |

## Discussion

This routine decomposes a day number into the corresponding year, month and day components. The month value of 1 corresponds to January while 12 corresponds to December. If the day number is omitted, the value of `Register.RTCDay` is used.

Note that `Register.RTCDay` is initialized to zero on power-up or reset. This day number corresponds to January 1, 1999.

**See Also** GetDayNumber, GetDayOfWeek, GetDayOfYear, PutDate

# GetDayNumber

---

**Type**               Function returning UnsignedInteger

**Invocation**       GetDayNumber(dayOfYear, year)

| Parameter | Method | Type     | Description                                     |
|-----------|--------|----------|---|
| dayOfYear | ByVal  | integral | The ordinal day number of the year (Jan 1 = 1). |
| year      | ByVal  | integral | The year (1999 to 2178).                        |

## Discussion

This routine computes the day number corresponding to the day of the year specified by the parameters. Day number 0 is January 1, 1999. The days in a year are numbered beginning with 1.

## Example

```
Dim dayNum as UnsignedInteger  
  
dayNum = GetDayNumber(59, 2005)
```

**See Also**           GetDate, GetDayOfWeek, GetDayOfYear, PutDate

# GetDayOfWeek

---

**Type**            Function returning Byte

**Invocation**    GetDayOfWeek()  
                  GetDayOfWeek(dayNum)

| Parameter | Method | Type     | Description                                    |
|-----------|--------|----------|--|
| dayNum    | ByVal  | integral | The day number to convert to year, month, day. |

## Discussion

This routine computes the day of the week corresponding to a day number. If the day number is omitted, the value of `Register.RTCDay` is used.. A return value of 1 corresponds to Sunday and a value of 7 corresponds to Saturday with the remaining days falling in order in between. There are built-in constants that represent the day numbers as shown in the table below.

| Day of Week Constants    |       |
|--------------------------|-------|
| Constant                 | Value |
| <code>zxSunday</code>    | 1     |
| <code>zxMonday</code>    | 2     |
| <code>zxTuesday</code>   | 3     |
| <code>zxWednesday</code> | 4     |
| <code>zxThursday</code>  | 5     |
| <code>zxFriday</code>    | 6     |
| <code>zxSaturday</code>  | 7     |

Note that `Register.RTCDay` is initialized to zero on power-up or reset. This day number corresponds to Friday, January 1, 1999.

**See Also**        `GetDate`, `GetDayNumber`, `GetDayOfYear`

# GetDayOfYear

---

**Type**                      Function returning UnsignedInteger

**Invocation**            GetDayOfYear(dayNum)  
                              GetDayOfYear(dayNum, year)

| Parameter | Method | Type     | Description  |
|-----------|--------|----------|--|
| dayNum    | ByVal  | integral | The day number to convert to day of year and year. |
| year      | ByRef  | int16    | The variable in which the year will be stored.     |

## Discussion

This routine computes the day of the year and the year corresponding to a day number (such as represented by `Register.RTCDay`). The first day of the year is numbered 1. If the second parameter is present, the variable to which it refers will receive the year value.

## Example

```
Dim dayOfYear as UnsignedInteger
Dim year as UnsignedInteger

dayOfYear = GetDayOfYear(Register.RTCDay, year)
```

**See Also**                `GetDate`, `GetDayNumber`, `GetDayOfWeek`

# GetEEPROM

---

**Type** Subroutine

**Invocation** GetEEPROM(addr, var, count)

| Parameter | Method | Type     | Description   |
|-----------|--------|----------|---|
| addr      | ByVal  | Long     | The Program Memory address from which to begin reading. |
| var       | ByRef  | any type | The variable in which to place the data read.           |
| count     | ByVal  | int16    | The number of bytes to read.                            |

## Discussion

This routine is provided for compatibility with BasicX. The more aptly named GetProgMem() should be used by new applications.

**See Also** GetProgMem, PutProgMem

# GetNibble

---

**Type**                Function returning Nibble

**Invocation**        GetNibble(var, nibbleNumber)

| Parameter    | Method | Type     | Description                                      |
|--------------|--------|----------|--|
| var          | ByRef  | any type | The variable from which the nibble will be read. |
| nibbleNumber | ByVal  | int8/16  | The nibble number to read.                       |

## Discussion

This function extracts a nibble value from memory beginning at the location of the specified variable. Nibble numbers 0-1 are taken from the byte at the specified location, nibble numbers 2-3 are taken from the subsequent byte, etc. In each case, the lower nibble number corresponds to the least significant four bits of the byte while the higher nibble number corresponds to the most significant four bits of the byte.

The return value will always be in the range 0 to 15.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            PutNibble



# GetPersistent

---

**Type** Subroutine

**Invocation** GetPersistent(addr, var, count)

| Parameter | Method | Type     | Description  |
|-----------|--------|----------|--|
| addr      | ByVal  | int16    | The address in Persistent Memory from which to read. |
| var       | ByRef  | any type | The variable in which to place the data read.        |
| count     | ByVal  | int8/16  | The number of bytes to read.                         |

## Discussion

This routine reads one or more bytes from Persistent Memory and places them in RAM beginning at the location of the specified variable. Note that if a number of bytes is specified that is larger than the given variable, adjacent memory will be overwritten, possibly with detrimental results.

The DataAddress property is useful to get the address of a Persistent Memory data item.

## Example

```
Dim pvar(1 to 10) as PersistentByte
Dim var(1 to 10) as Byte

Call GetPersistent(pvar.DataAddress, var, SizeOf(pvar))
```

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** PutPersistent

# GetPin

---

**Type**            Function returning Byte

**Invocation**    GetPin(pin)

| Parameter | Method | Type | Description      |
|-----------|--------|------|------------------|
| pin       | ByVal  | Byte | The pin to read. |

## Discussion

If the specified pin is configured to be an input, this function reads the state of the pin and returns the value 0 or 1 corresponding to logic zero and logic one. If the pin number is invalid the result is undefined. If the pin is configured to be an output, it is reconfigured to be an input in tri-state mode before reading the input value.

## Compatibility

The BasicX documentation says that the result is undefined if `GetPin()` is called for a pin that is configured as an output. Tests show that the pin is actually reconfigured to be an input in tri-state mode. The ZBasic implementation of `GetPin()` does the same.

**See Also**        PutPin

# GetProgMem

---

**Type** Subroutine

**Invocation** GetProgMem(addr, var, count)

| Parameter | Method | Type     | Description   |
|-----------|--------|----------|---|
| addr      | ByVal  | Long     | The Program Memory address from which to begin reading. |
| var       | ByRef  | any type | The variable in which to place the data read.           |
| count     | ByVal  | int16    | The number of bytes to read.                            |

## Discussion

This routine reads one or more bytes from Program Memory (where the user program is stored) and places them in RAM beginning at the location of the specified variable. Note that if a number of bytes is specified that is larger than the given variable, adjacent memory will be overwritten, possibly with detrimental results.

**See Also** PutProgMem

# GetQueue

**Type** Subroutine

**Invocation** GetQueue(queue, var, count)  
GetQueue(queue, var, count, timeLimit, timeoutFlag)

| Parameter   | Method | Type          | Description   |
|-------------|--------|---------------|---|
| queue       | ByRef  | array of Byte | The queue from which to read data.                            |
| var         | ByRef  | any type      | The variable to which to write the data from the queue.       |
| count       | ByVal  | int16         | The number of bytes to read from the queue.                   |
| timeLimit   | ByVal  | Single        | The amount of time to wait for data availability, in seconds. |
| timeoutFlag | ByRef  | Boolean       | A variable to indicate if the call timed out.                 |

## Discussion

This routine has two forms. The first form simply attempts to read the given number of bytes from the specified queue and place them in RAM beginning at the location of the given variable. In this case, the routine will not return until requested number of bytes is available. If not enough data is placed in the queue, the routine will never return. Note that if the calling task is locked and the queue contains insufficient space for the data to be written data when this routine is called, the task will be unlocked to allow other tasks to run.

The second form specifies, additionally, a `timeLimit` and a `flag` variable. In this case, if the requested number of bytes does not become available within the specified time, the routine will return, having transferred zero bytes, and the `flag` variable will be set to `True` indicating that the routine timed out. If the requested number of bytes does become available before the specified time expires, that number of bytes will be removed from the queue and transferred to the specified memory location and the `flag` variable will be set to `False` indicating that the transfer did not time out. The resolution of the timeout value is the same as the RTC tick, approximately 1.95mS.

In either case, if data is removed from the queue it is written to RAM beginning at the location of the specified variable. Note that if the count specifies a number of bytes larger than the variable, the additional bytes will be written to subsequent RAM locations. This may have exactly the effect that you intended but depending on the function of those subsequent bytes it may have a deleterious effect on your program.

Note that before any queue operations are performed, the queue data structure must be initialized. See the discussion of `OpenQueue()` for more details.

Although this subroutine will accept a String variable as the second parameter it is generally not useful to do so because the control bytes at the beginning of the string will be overwritten. If you want to populate a string using data from a queue the alternatives are:

- 1) Build up the string by retrieving individual characters one by one and appending them to a string.
- 2) Retrieve a group of bytes to a Byte array and use the `MakeString()` function to create a string from the constituent bytes.
- 3) Use the `GetQueueStr()` function to obtain a string containing characters from the queue.

## Example

```
Dim inQueue(1 to 40) as Byte
Dim lval as Long

Call OpenQueue(inQueue, SizeOf(inQueue))
Call GetQueue(inQueue, lval, SizeOf(lval))
```

Alternately,

```
Dim inQueue(1 to 40) as Byte
Dim lval as Long
Dim timeOut as Boolean

Call OpenQueue(inQueue, SizeOf(inQueue))
Call GetQueue(inQueue, lval, SizeOf(lval), 1.0, timeOut)
```

### **Compatibility**

BasicX allows any type for the first parameter. This implementation requires that it be an array of `Byte`.

The BasicX manual indicates that the range of values for the `timeLimit` parameter is 0.0 to 65.536 seconds implying a 1ms resolution. This implementation has a 1.95ms resolution and a range of 0.0 to about 127.0 seconds.

**See Also**      `GetQueueStr`

# GetQueueBufferSize

---

**Type**                Function returning Integer

**Invocation**        GetQueueBufferSize(queue)

| Parameter | Method | Type          | Description            |
|-----------|--------|---------------|------------------------|
| queue     | ByRef  | array of Byte | The queue of interest. |

## Discussion

This function returns the number of bytes of data space in a queue that has been properly initialized using `OpenQueue()`. Note that the data space in a queue is somewhat less than the number of bytes in the byte array comprising the queue due to space required for queue management information. See `OpenQueue()` for more details.

## Compatibility

BasicX allows any type for the first parameter. This implementation requires that it be an array of `Byte`.

**See Also**            GetQueueCount, GetQueueSpace

# GetQueueCount

---

**Type**                Function returning Integer

**Invocation**        GetQueueCount(queue)

| Parameter | Method | Type          | Description            |
|-----------|--------|---------------|------------------------|
| queue     | ByRef  | array of Byte | The queue of interest. |

## Discussion

This function returns the number of bytes of data currently in the specified queue. It is useful to note that this value subtracted from that returned by `GetQueueBufferSize()` indicates the remaining available data space in the queue.

Note that before any queue operations are performed, the queue data structure must be initialized. See the discussion of `OpenQueue()` for more details.

## Compatibility

BasicX allows any type for the first parameter. This implementation requires that it be an array of Byte.

**See Also**            GetQueueBufferSize, GetQueueSpace

# GetQueueSpace

---

**Type**                Function returning Integer

**Invocation**        GetQueueSpace(queue)

| Parameter | Method | Type          | Description            |
|-----------|--------|---------------|------------------------|
| queue     | ByRef  | array of Byte | The queue of interest. |

## Discussion

This function returns the number of bytes of space remaining in the specified queue, effectively the same result as the expression `GetQueueBufferSize() - GetQueueCount()`.

Note that before any queue operations are performed, the queue data structure must be initialized. See the discussion of `OpenQueue()` for more details.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            GetQueueBufferSize, GetQueueCount



# GetQueueStr

---

**Type**                Function returning String

**Invocation**        GetQueueStr(queue) or  
                         GetQueueStr(queue, maxChars)

| Parameter | Method | Type          | Description                                   |
|-----------|--------|---------------|---|
| queue     | ByRef  | array of Byte | The queue of interest.                        |
| maxChars  | ByVal  | integral      | The maximum number of characters to retrieve. |

## Discussion

This function extracts a number of characters from the specified queue and returns a string populated with those characters. The number of characters is limited to the lesser of 1) the number of characters in the queue at the time of the call, 2) the value of `maxChars` (if specified), and 3) the maximum number of characters allowed in a string.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            GetQueue

# GetTime

---

**Type** Subroutine

**Invocation** GetTime(hour, minute, seconds)  
GetTime(hour, minute, seconds, tick)

| Parameter | Method | Type     | Description  |
|-----------|--------|----------|--|
| hour      | ByRef  | Byte     | The variable in which to place the hour value (0-23).    |
| minute    | ByRef  | Byte     | The variable in which to place the minutes value (0-59). |
| seconds   | ByRef  | Single   | The variable in which to place the seconds value.        |
| tick      | ByVal  | integral | The tick count to decompose.                             |

## Discussion

This routine decomposes a tick count into the equivalent hour, minute and second components. If the tick count is omitted, the value of `Register.RTCTick` is used. The resolution of the `seconds` value is approximately 1.95ms.

Note that `Register.RTCTick` is initialized to zero on power-up or reset. This corresponds to 0:00:00.

## Compatibility

Explicitly specifying the tick count to use is not supported in BasicX compatibility mode.

# GetTimestamp

---

**Type** Subroutine

**Invocation** GetTimestamp(year, month, day, hour, minute, seconds)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| year      | ByRef  | int16  | The variable in which to place the year value (1999-2177). |
| month     | ByRef  | Byte   | The variable in which to place the month value (1-12).     |
| day       | ByRef  | Byte   | The variable in which to place the day value (1-31).       |
| hour      | ByRef  | Byte   | The variable in which to place the hour value (0-23).      |
| minute    | ByRef  | Byte   | The variable in which to place the minutes value (0-59).   |
| seconds   | ByRef  | Single | The variable in which to place the seconds value.          |

## Discussion

This routine decomposes the value of `Register.RTCDay` and `Register.RTCTick` into year, month, day, hour, minute and second components. See `GetDate()` and `GetTime()` for more details.

# HiByte

---

**Type**            Function returning Byte

**Invocation**    HiByte(val)

| Parameter | Method | Type    | Description                                  |
|-----------|--------|---------|--|
| val       | ByVal  | numeric | The value of which the high byte is desired. |

## Discussion

This function returns the most significant byte of the specified value except that if the specified value is a Byte value, the result will be zero.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        HiWord, LoByte, LoWord, MidWord

# HiWord

---

**Type**            Function returning UnsignedInteger

**Invocation**    HiWord(val)

| Parameter | Method | Type    | Description                                  |
|-----------|--------|---------|--|
| val       | ByVal  | numeric | The value of which the high word is desired. |

## Discussion

This function returns the most significant word of the specified value except that if the specified value is a Byte, Integer or UnsignedInteger value, the result will be zero.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        HiByte, LoByte, LoWord, MidWord

# I2CCmd

**Type** Function returning Integer

**Invocation** I2CCmd(channel, slaveID, writeCnt, writeData, readCnt, readData)

| Parameter | Method | Type     | Description  |
|-----------|--------|----------|--|
| channel   | ByVal  | Byte     | The I2C channel number (0-4).                                      |
| slaveID   | ByVal  | Byte     | The identifier of the I2C slave device (in the 7 high order bits). |
| writeCnt  | ByVal  | integral | The number of bytes to write (0 – 65535).                          |
| writeData | ByRef  | any type | The variable containing the data to write.                         |
| readCnt   | ByVal  | integral | The number of bytes to read (0 – 65535).                           |
| readData  | ByRef  | any type | The variable in which to place the data read.                      |

## Discussion

The routine allows you to send and/or receive data from an I2C device. The specified channel must have been previously opened with a call to `OpenI2C()`. If the channel has not been opened, the results are undefined. If an invalid channel is specified or if both `writeCnt` and `readCnt` are zero, the function returns immediately without doing anything and the return value is zero. You may specify the value 0 for `writeData` or `readData` if no data is being provided for writing or reading, respectively. If you do this, the corresponding data count parameter must also be zero or the compiler will issue an error message.

The execution of the I2C command sequence begins by issuing an I2C start condition on the SDA and SCL lines. Next, if `writeCnt` is non-zero the given `slaveID` value is transmitted (with the least significant bit being zero) followed by the specified number of bytes taken from `writeData`. Then, if `readCnt` is non-zero the `slaveID` value is transmitted again but with the least significant bit being one and the specified number of bytes is read from the slave and placed in `readData`. Finally, an I2C stop condition is issued followed by both the SDA and SCL lines returning to the idle state.

The return value may be negative, zero or positive. If the return value is negative it signifies that the slave failed to positively acknowledge one of the transmitted bytes. The value is the negative of the number of bytes that were not successfully transmitted. If the slave fails to positively acknowledge either the slave ID or the first data byte, the return value will be the negative of the `writeCnt` parameter value. If the return value is non-negative it represents the number of data bytes read from the slave and placed in `readData`.

## Example

```
Dim odata(1 to 2) as Byte, idata(1 to 10) as Byte
Dim ival as Integer

Call OpenI2C (1, 12, 13)
odata(1) = &H06
odata(2) = &H00
ival = I2CCmd(1, &H7e, 2, odata(1), 10, idata(1))
```

## Resource Usage

This function uses the I/O Timer for channels 1 to 4. If the timer is already in use, the result and the return value are both undefined. Interrupts are disabled for periods of about 9 times the selected I2C bit time plus additional amounts due to slave clock stretching for each byte sent and received (interrupts are reenabled between bytes). However, RTC ticks are accumulated during the process so the RTC should not lose time.

**Compatibility**

This function is not available in BasicX compatibility mode.

**See Also**      `OpenI2C`, `I2CGetByte`, `I2CPutByte`, `I2CStart`, `I2CStop`, `CloseI2C`

# I2CGetByte

---

**Type**                Function returning Byte

**Invocation**        I2CGetByte(channel, ackValue)

| Parameter | Method | Type    | Description   |
|-----------|--------|---------|---|
| channel   | ByVal  | Byte    | The I2C channel number (0-4).                                       |
| ackValue  | ByVal  | Boolean | The value to send to the slave in acknowledgement of the data byte. |

## Discussion

This function retrieves a data value from an I2C slave and responds to the receipt of that data by sending back the specified acknowledgement value. The value returned by this function is the data byte received from the slave.

This function can be used in conjunction with `I2CStart()`, `I2CPutByte()` and `I2CStop()` to perform a lower level interaction with an I2C slave device. Knowledge of the I2C protocol and the specifications of the particular I2C device are required in order to use this function.

If the specified I2C channel has not been properly prepared using `OpenI2C()`, the results are undefined. If an invalid channel number is specified, the function returns immediately without doing anything.

## Resource Usage

This function uses the I/O Timer for channels 1 to 4. If the timer is already in use, the function will do nothing and the return value is undefined. Interrupts are disabled for about 9 times the selected I2C bit time plus additional amounts due to slave clock stretching. However, RTC ticks are accumulated during the process so the RTC should not lose time.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            `OpenI2C`, `Closel2C`, `I2CPutByte`, `I2CStart`, `I2CStop`, `I2CCmd`



# I2CPutByte

**Type**                      Function return Boolean

**Invocation**            I2CPutByte(channel, dataVal)

| Parameter | Method | Type | Description                         |
|-----------|--------|------|-------------------------------------|
| channel   | ByVal  | Byte | The I2C channel number (0-4).       |
| dataVal   | ByVal  | Byte | The data byte to send to the slave. |

## Discussion

This function transmits a data value to an I2C slave and reads the acknowledgement bit returned by the slave. The value returned by this function is the value of the acknowledge bit received from the slave device – a positive acknowledgement results in a True value being returned.

This function can be used in conjunction with `I2CStart()`, `I2CGetByte()` and `I2CStop()` to perform a lower level interaction with an I2C slave device. Knowledge of the I2C protocol and the specifications of the particular I2C device are required in order to use this function.

If the specified I2C channel has not been properly prepared using `OpenI2C()`, the results are undefined. If an invalid channel number is specified, the function returns immediately without doing anything.

## Resource Usage

This function uses the for channels 1 to 4. If the timer is already in use, the function will do nothing and the return value is undefined. Interrupts are disabled for about 9 times the selected I2C bit time plus additional amounts due to slave clock stretching. However, RTC ticks are accumulated during the process so the RTC should not lose time.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**                      `OpenI2C`, `Closel2C`, `I2CGetByte`, `I2CStart`, `I2CStop`, `I2CCmd`

# I2CStart

---

**Type** Subroutine

**Invocation** I2CStart(channel)

| Parameter | Method | Type | Description                   |
|-----------|--------|------|-------------------------------|
| channel   | ByVal  | Byte | The I2C channel number (0-4). |

## Discussion

This subroutine initiates an I2C bus cycle by implementing the proper sequence of transitions on the SDA and SCL lines.

This subroutine can be used in conjunction with `I2CGetByte()`, `I2CPutByte()` and `I2CStop()` to perform a lower level interaction with an I2C slave device. Knowledge of the I2C protocol and the specifications of the particular I2C device are required in order to use this function.

If the specified I2C channel has not been properly prepared using `OpenI2C()`, the results are undefined. If an invalid channel number is specified, the function returns immediately without doing anything.

## Compatibility

This subroutine is not available in BasicX compatibility mode.

**See Also** OpenI2C, CloseI2C, I2CGetByte, I2CPutByte, I2CStop, I2CCmd

# I2CStop

---

**Type** Subroutine

**Invocation** I2CStop(channel)

| Parameter | Method | Type | Description                   |
|-----------|--------|------|-------------------------------|
| channel   | ByVal  | Byte | The I2C channel number (0-4). |

## Discussion

This subroutine terminates an I2C bus cycle by implementing the proper sequence of transitions on the SDA and SCL lines.

This subroutine can be used in conjunction with `I2CStart()`, `I2CGetByte()` and `I2CPutByte()` to perform a lower level interaction with an I2C slave device. Knowledge of the I2C protocol and the specifications of the particular I2C device are required in order to use this function.

If the specified I2C channel has not been properly prepared using `OpenI2C()`, the results are undefined. If an invalid channel number is specified, the function returns immediately without doing anything.

## Compatibility

This subroutine is not available in BasicX compatibility mode.

**See Also** OpenI2C, CloseI2C, I2CGetByte, I2CPutByte, I2CStart, I2CCmd

# IIf

**Type** Function returning the same type as the second parameter

**Invocation** IIf(testExpr, trueExpr, falseExpr)

| Parameter | Method | Type     | Description  |
|-----------|--------|----------|--|
| testExpr  | ByVal  | Boolean  | The expression to evaluate, the result of which determine which expression value will be returned. |
| trueExpr  | ByVal  | any type | The value to return if <i>testExpr</i> evaluates to True.  |
| falseExpr | ByVal  | any type | The value to return if <i>testExpr</i> evaluates to False.   |

## Discussion

This function is adapted from VB6 where it is sometimes called “Immediate If”. It is used to select one of two values based on the result of a test. Employing this function will generally result in less code than an equivalent If-Then-Else structure. On the other hand, the execution of this function does use more stack space than an equivalent If-Then-Else structure. Also, it is important to note that using this function is not exactly the same as an If-Then-Else because both the *trueExpr* and the *falseExpr* are always evaluated. This difference is only significant if the evaluation of one or both of these expressions has side effects.

Note that *trueExpr* and *falseExpr* must have the same type or be of compatible types.

## Examples

```
Dim a as Byte
Dim b as UnsignedInteger
Dim u as UnsignedInteger

u = IIf(a > 3, 5, b)

Debug.Print IIf(a = 5, "Hello", "Goodbye")
```

## Compatibility

This function is not available in BasicX compatibility mode. Also, it is only supported by ZX firmware v1.1.0 or later.

# InputCapture

**Type** Subroutine

**Invocation** InputCapture(data, count, flags)  
InputCapture(data, count, flags, timeout)

| Parameter | Method | Type                        | Description   |
|-----------|--------|-----------------------------|---|
| data      | ByRef  | array of<br>UnsignedInteger | The array in which pulse width information will be stored.  |
| count     | ByVal  | int16                       | The number of pulse widths to store. This should be no larger than the number of entries in the passed array.                                     |
| flags     | ByVal  | Byte                        | A value of zero requests that a falling edge begin the capture process while a value of 1 indicates a rising edge. All other values are reserved. |
| timeout   | ByVal  | Integral                    | If non-zero, this parameter specifies a timeout value that, if exceeded, will terminate the input capture process.                                |

## Discussion

Invoking this routine is equivalent to the call `InputCaptureEx(pin, data, count, flags)` or `InputCaptureEx(pin, data, count, flags, timeout)` where `pin` is the default input capture pin for the device as shown in the table below. See the description of `InputCaptureEx()` for more detailed information.

| Default Input Capture Pin              |         |
|--|---------|
| ZX Models                              | Pin     |
| ZX-24, ZX-24a, ZX-24p, ZX-24n          | 12, D.6 |
| ZX-40, ZX-40a, ZX-40p, ZX-40n          | 20, D.6 |
| ZX-44, ZX-44a, ZX-44p, ZX-44n          | 15, D.6 |
| ZX-24e, ZX-24ae, ZX-24pe, ZX-24ne      | 14, D.6 |
| ZX-1281, ZX-1281n                      | 29, D.4 |
| ZX-1280, ZX-1280n                      | 47, D.4 |
| ZX-128e, ZX-128ne, ZX-1281e, ZX-1281ne | 8, D.4  |

## Example

```
Dim pd(1 to 5) as UnsignedInteger

Call PutPin(12, zxInputTriState)
Call InputCapture(pd, UBound(pd), 1)
```

## Compatibility

The BasicX compiler erroneously allows any variable for the first parameter. This implementation requires the data type to be `UnsignedInteger` or `Integer` although it needn't be an array. For practical purposes, an array will almost always be used.

In BasicX compatibility mode, the use of the optional fourth parameter is not supported. Also, because the processor runs at twice the speed of the BX-24 processor, the default time unit is one half of that provided for by BasicX.

See the description of `InputCaptureEx()` for a discussion of which ISRs are required.

# InputCaptureEx

**Type** Subroutine

**Invocation** InputCaptureEx(pin, data, count, flags)  
InputCaptureEx(pin, data, count, flags, timeout)

| Parameter | Method | Type                     | Description   |
|-----------|--------|--------------------------|---|
| pin       | ByVal  | Byte                     | The input capture pin to use.   |
| data      | ByRef  | array of UnsignedInteger | The array in which pulse width information will be stored.  |
| count     | ByVal  | int16                    | The number of pulse widths to store. This should be no larger than the number of entries in the passed array.                                     |
| flags     | ByVal  | Byte                     | A value of zero requests that a falling edge begin the capture process while a value of 1 indicates a rising edge. All other values are reserved. |
| timeout   | ByVal  | integral                 | If non-zero, this parameter specifies a timeout value that, if exceeded, will terminate the input capture process.                                |

## Discussion

This routine collects timing data from a pulse train applied to the specified input capture pin and stores it in the specified array. The stored data reflects the width of the successive high and low portions of the pulse train. If any segment is longer than can be represented in a 16-bit value, the stored value will be 65535 (&Hffff) and the immediately following value, if any, will be meaningless.

Prior to commencing the input capture process all of the elements of the data array are initialized with the value 65534 (&Hfffe). This fact can be used to determine the actual number of timing data stored in the array during input capture.

The stored values represent the number of I/O Timer ticks (by default about 67.8ns) measured for each segment of the pulse train. However, the value of `Register.TimerSpeed1` may be changed to allow longer pulse widths to be measured. See the section on Timers for more information.

If the optional `timeout` parameter is specified and is non-zero, the Input Capture process will be terminated if  $N * 65536$  I/O Timer ticks occur (where  $N$  is the value of the `timeout` parameter) before the specified number of datapoints has been stored. This gives a range of possible timeout values from about 4.5mS to 290 seconds with a resolution of 4.5mS (using the default value of `Register.TimerSpeed1`).

The calling task will be suspended until the specified number of datapoints has been stored, the timeout value is exceeded or the task is resumed using `ResumeTask()`. Other tasks will be allowed to run but you must be careful to not call any routines that may disable interrupts for long periods of time because that could interfere with the accuracy of the input capture timing.

## Resource Usage

This routine utilizes a timer to collect the timing information of the pulse train. The table below indicates which timer is used and the corresponding input pin for the supported channels. The corresponding timer busy flag (e.g. `Register.Timer1Busy`) will be set `True` for the duration of the input capture operation. Also, on the ZX-24, ZX-24a, ZX-24p and ZX-24n the input capture pin is common with PortC bit 0. This means that you should set pin 12 to be an input (either tri-state or pull-up) when you want to use `InputCapture()` so that it doesn't interfere with the pulse train to be measured. This routine cannot be used at the same time as `OutputCapture()` or `OutputCaptureEx()` when that routine requires the same timer.

| Valid Input Capture Pins               |            |            |            |            |
|--|------------|------------|------------|------------|
| ZX Models                              | Timer1 Pin | Timer3 Pin | Timer4 Pin | Timer5 Pin |
| ZX-24, ZX-24a, ZX-24p, ZX-24n          | 12, D.6    | -          | -          | -          |
| ZX-40, ZX-40a, ZX-40p, ZX-40n          | 20, D.6    | -          | -          | -          |
| ZX-44, ZX-44a, ZX-44p, ZX-44n          | 15, D.6    | -          | -          | -          |
| ZX-24e, ZX-24ae, ZX-24pe, ZX-24ne      | 14, D.6    | -          | -          | -          |
| ZX-1281, ZX-1281n                      | 29, D.4    | 9, E.7     | -          | -          |
| ZX-1280, ZX-1280n                      | 47, D.4    | 9, E.7     | 35, L.0    | 36, L.1    |
| ZX-128e, ZX-128ne, ZX-1281e, ZX-1281ne | 8, D.4     | 13, E.7    | -          | -          |

For native code devices, the table below gives the ISRs that may be loaded if your program uses InputCaptureEx(). If the compiler cannot determine which specific timer ISR is required by analyzing the parameters used, all listed ISRs will be included.

| ISRs Required     |   |
|-------------------|---|
| Underlying CPU    | ISR Name  |
| mega644P          | Timer1_Capt, Timer1_OVF   |
| megal281, megal28 | Timer1_Capt, Timer1_OVF,<br>Timer3_Capt, Timer3_OVF   |
| megal280          | Timer1_Capt, Timer1_OVF,<br>Timer3_Capt, Timer3_OVF,<br>Timer4_Capt, Timer4_OVF,<br>Timer5_Capt, Timer5_OVF |

### Example

```
Dim pd(1 to 5) as UnsignedInteger

Call PutPin(D.6, zxInputTriState)
Call InputCaptureEx(D.6, pd, UBound(pd), 1)
```

### Compatibility

This routine is not available in BasicX compatibility mode.

# LBound

**Type**            Function returning Integer

**Invocation**    LBound(array) or  
                  LBound(array, dimension)

| Parameter | Method | Type      | Description  |
|-----------|--------|-----------|--|
| array     | ByRef  | any array | The array about which the bound information is desired.          |
| dimension | ByVal  | int16     | The dimension of interest. See the description for more details. |

## Discussion

This function returns the lower bound of the specified array. There are two forms. The first requires only the array to be specified. In this case, the lower bound of the first dimension of the array is returned. The second form specifies a dimension number, the valid range of which is 1 to the number of dimensions of the array. The array may be located in RAM, Program Memory or Persistent Memory.

Note that the use of this function instead of hard-coding values makes your code easier to maintain.

## Example

```
Dim ba(1 to 20) as Byte
Dim ma(3 to 5, -6 to 7) as Byte
Dim i as Integer

i = LBound(ba)           ' the result is 1
i = LBound(ma)           ' the result is 3
i = LBound(ma, 1)        ' the result is 3
i = LBound(ma, 2)        ' the result is -6
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        UBound



# LCase

---

**Type**            Function returning String

**Invocation**    LCase(str)

| Parameter | Method | Type   | Description                             |
|-----------|--------|--------|---|
| str       | ByVal  | String | The string to be changed to lower case. |

## Discussion

This function returns a new string containing the same characters as the passed string except that all upper case characters will be replaced with lower case characters.

## Example

```
Dim s as String, s1 as String
s = "Hello, world!"
s2 = LCase(s)                    ' the result will be "hello, world!"
```

**See Also**        UCase

# Left

**Type**            Function returning String

**Invocation**    Left(str, length)

| Parameter | Method | Type    | Description  |
|-----------|--------|---------|--|
| Str       | ByVal  | String  | The string from which to extract characters.         |
| length    | ByVal  | int8/16 | The number of characters to extract from the string. |

## Discussion

This function returns a string consisting of the leftmost characters of the given string. The maximum number of characters in the returned string is the smaller of 1) the number of characters in the string passed as the first parameter and 2) the value of the second parameter. Internally, the length is interpreted as a 16-bit signed value and negative values are treated as zero.

This function produces the same result as `Mid(str, 1, length)`.

## Example

```
Dim s as String, s2 as String

s = "Hello, world!"
s2 = Left(s, 5)           ' the result will be "Hello"
```

**See Also**        Mid, Right, Trim

# Len

---

**Type**            Function returning Integer

**Invocation**    Len(str)

| Parameter | Method | Type   | Description   |
|-----------|--------|--------|---|
| str       | ByVal  | String | The string of which the length is to be determined. |

## Discussion

This function returns the length of the given string, in bytes. Note that the length may be zero.

## Example

```
Dim s as String
Dim i as Integer

s = "Hello, world!"
i = Len(s)                ' the result will be 13
```

# LoByte

---

**Type**            Function returning Byte

**Invocation**    LoByte(val)

| Parameter | Method | Type    | Description                                 |
|-----------|--------|---------|---|
| val       | ByVal  | numeric | The value of which the low byte is desired. |

## Discussion

This function returns the least significant byte of the specified value.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        HiByte, HiWord, LoWord, MidWord

# LockTask

---

**Type**            Subroutine

**Invocation**    LockTask()

## Discussion

This routine causes the running task to become locked so that no other task can run. The one exception to this is a task that is awaiting an external interrupt or an interval interrupt. Note that a task may explicitly unlock itself by calling `UnlockTask()`. A task will also become unlocked if it calls any of the sleep or delay routines.

Note that multiple calls to `LockTask()` have the same effect as a single call to `LockTask()` assuming that no other calls are made that implicitly unlock the task.

## Compatibility

The BasicX documentation indicates that a locked task will yield to a task that is awaiting an interrupt when the interrupt occurs. However, testing indicates that this is, in fact, not the case. This implementation allows an interrupt task to have priority over a locked task.

**See Also**        UnlockTask, Delay, Sleep, WaitForInterrupt, WaitForInterval

# Log

---

**Type**            Function returning Single

**Invocation**    Log(arg)

| Parameter | Method | Type   | Description   |
|-----------|--------|--------|---|
| arg       | ByVal  | Single | The value of which the natural log is to be computed. |

## Discussion

This function returns the `Single` value corresponding to natural logarithm (base e) of the value provided. The transcendental value e, upon which the natural logarithm is based, is approximately 2.71828. This function is the inverse of the `Exp( )` function.

If the value of the argument provided is zero, the result is positive infinity. If the argument value is negative, the result is NaN.

**See Also**        Exp, Exp10, Log10

# Log10

---

**Type**            Function returning Single

**Invocation**    Log10(arg)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| arg       | ByVal  | Single | The value of which the common log is to be computed. |

## Discussion

This function returns the `Single` value corresponding to the common logarithm (base 10) of the value provided. This function is the inverse of the `Exp10()` function.

If the value of the argument provided is zero, the result is positive infinity. If the argument value is negative, the result is NaN.

**See Also**        Exp, Exp10, Log

# LongJump

**Type** Subroutine

**Invocation** LongJump(jmpbuf, val)

| Parameter | Method | Type          | Description   |
|-----------|--------|---------------|---|
| jmpbuf    | ByRef  | Array of Byte | A buffer holding the return context, see description below. |
| val       | ByVal  | int16         | The value to be returned to the original SetJump() caller.  |

## Discussion

This subroutine, in conjunction with `SetJump()`, provides a way to circumvent the normal call-return structure and return directly to a distant caller. It is the equivalent of a non-local Goto function and can be used, among other purposes, to handle exceptions in your programs. The first parameter specifies a `Byte` array that has been previously initialized by a call to `SetJump()`. The second parameter specifies a value that will be seen by the original `SetJump()` caller as the return value. This value, which should be non-zero, can indicate the nature of the condition that led to the `LongJump()` call. See the section on Exception Handling in the ZBasic Reference Manual for more details.

## Caution

Passing a jump buffer that has not been prepared by a call to `SetJump()`, one that has been modified after the `SetJump()` call, or one that was prepared by a subroutine/function that is no longer active will have unpredictable and almost certainly undesirable effects.

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** SetJump



# LoWord

---

**Type**            Function returning UnsignedInteger

**Invocation**    LoWord(val)

| Parameter | Method | Type    | Description                                 |
|-----------|--------|---------|---|
| val       | ByVal  | numeric | The value of which the low word is desired. |

## Discussion

This function returns the least significant word of the specified value. If the specified value is a Byte the return value will have zero in the high byte.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        HiByte, HiWord, LoByte, MidWord

# MakeDword

---

**Type**            Function returning UnsignedLong

**Invocation**    MakeDword(loWord, hiWord)

| Parameter | Method | Type  | Description   |
|-----------|--------|-------|---|
| loWord    | ByVal  | int16 | The value for the low word of the double word value.  |
| hiWord    | ByVal  | int16 | The value for the high word of the double word value. |

## Discussion

This function returns a value composed of the two word values.

## Example

```
Dim w1 as UnsignedInteger, w2 as UnsignedInteger
Dim ul as UnsignedLong

ul = MakeDword(w1, w2)
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        MakeWord

# MakeString

---

**Type**                Function returning String

**Invocation**        MakeString(address, length)

| Parameter | Method | Type    | Description   |
|-----------|--------|---------|---|
| address   | ByVal  | int16   | The address of bytes with which to populate the string. |
| length    | ByVal  | int8/16 | The number of characters to place in the string.        |

## Discussion

This function populates a string with an arbitrary byte stream. It is most useful for composing or modifying strings but may have other uses as well.

## Example

```
Dim ba(1 to 10) as Byte
Dim i as Integer
Dim s as String

For i = LBound(ba) to UBound(ba)
    ba(i) = &H60 + CByte(i)
Next i
s = MakeString(MemAddress(ba), SizeOf(ba))
```

## Compatibility

This function is not available in BasicX compatibility mode.

# MakeWord

---

**Type**            Function returning UnsignedInteger

**Invocation**    MakeWord(loByte, hiByte)

| Parameter | Method | Type | Description                                    |
|-----------|--------|------|--|
| loByte    | ByVal  | Byte | The value for the low byte of the word value.  |
| hiByte    | ByVal  | Byte | The value for the high byte of the word value. |

## Discussion

This function returns a value composed of the two byte values.

## Example

```
Dim b1 as Byte, b2 as Byte
Dim u as UnsignedInteger

u = MakeWord(b1, b2)
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        MakeDword

# Max

---

**Type**                Function (see discussion for the return type)

**Invocation**        Max(val1, val2)

| Parameter | Method | Type    | Description  |
|-----------|--------|---------|--|
| val1      | ByVal  | numeric | One of two values of which the largest is desired. |
| val2      | ByVal  | numeric | One of two values of which the largest is desired. |

## Discussion

This function returns the larger of the two supplied values, both of which must be of the same type. If the supplied values are signed, the determination of which is largest takes the sign of the values into account. The return value is the same type as the parameters.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            Min

# MemAddress

**Type**            Function returning Integer

**Invocation**    MemAddress(var)

| Parameter | Method | Type         | Description                                   |
|-----------|--------|--------------|---|
| var       | ByRef  | any variable | The variable of which the address is desired. |

## Discussion

This function returns the `Integer` representation of the RAM address of the specified variable. Note that for arrays, you may also specify subscript expressions for all of the array dimensions to yield the address of an individual array element. Without the subscript expressions, the resulting value will be the address of the first element of the array.

This function is useful for deriving the address to pass to the several functions that require a RAM address, e.g. `BitCopy()`, `RamPeek()`, `RamPoke()`, etc.

The address of any variable can also be obtained using the `DataAddress` property. For RAM-based variables, the `DataAddress` property is of type `UnsignedInteger`.

## Example

```
Dim addr as Integer
Dim ba(1 to 20) as Byte
Dim fval as Single

addr = MemAddress(fval)
addr = MemAddress(ba)
addr = MemAddress(ba(2))
addr = fval.DataAddress
addr = ba.DataAddress
addr = ba.DataAddress(2)
```

## Compatibility

BasicX only supports the `DataAddress` property for Program Memory data items.

**See Also**            MemAddressU, VarPtr

# MemAddressU

**Type**                      Function returning `UnsignedInteger`

**Invocation**            `MemAddressU(var)`

| Parameter | Method | Type         | Description                                   |
|-----------|--------|--------------|---|
| var       | ByRef  | any variable | The variable of which the address is desired. |

## Discussion

This function returns the `UnsignedInteger` representation of the RAM address of the specified variable. Note that for arrays, you may also specify subscript expressions for all of the array dimensions to yield the address of an individual array element. Without the subscript expressions, the resulting value will be the address of the first element of the array.

This function is useful for deriving the address to pass to the several functions that require a RAM address, e.g. `BitCopy()`, `RamPeek()`, `RamPoke()`, etc.

The `DataAddress` property may also be used to determine the address of a variable (except in BasicX compatibility mode). The type of the resulting value is `UnsignedInteger`. See the examples below.

## Examples

```
Dim addr as UnsignedInteger
Dim ba(1 to 20) as Byte
Dim fval as Single

addr = MemAddressU(fval)
addr = MemAddressU(ba)
addr = MemAddressU(ba(2))
addr = ba.DataAddress
addr = ba.DataAddress(2)
```

**See Also**                      `MemAddress`, `VarPtr`

# MemCmp

**Type**            Function returning Integer

**Invocation**    MemCmp(addr1, addr2, count)

| Parameter | Method | Type     | Description   |
|-----------|--------|----------|---|
| addr1     | ByVal  | integral | The address of the first block of memory to be compared.  |
| addr2     | ByVal  | integral | The address of the second block of memory to be compared. |
| count     | ByVal  | integral | The number of bytes to compare.                           |

## Discussion

This function can be used to compare two arbitrary sequences of data in RAM. If all of the bytes in the two blocks are the same (over the given number of bytes to compare) the value zero is returned. Otherwise, the return value will be greater than zero if at the position of the first mismatch the byte in the first block is greater than the corresponding byte in the second block. If the converse is true, the return value will be less than zero.

All three parameters are converted internally to `UnsignedInteger`.

## Example

```
Dim a1(1 to 10) as Byte
Dim a2(1 to 10) as Byte
Dim ival as Integer

ival = MemCmp(a1.DataAddress, a2.DataAddress, SizeOf(a1))
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        MemCopy, MemSet



# MemCopy

**Type** Subroutine

**Invocation** MemCopy(destination, source, count)

| Parameter   | Method | Type     | Description                              |
|-------------|--------|----------|--|
| destination | ByVal  | integral | The address to which to begin copying.   |
| source      | ByVal  | integral | The address from which to begin copying. |
| count       | ByVal  | integral | The number of bytes to copy.             |

## Discussion

This subroutine can be used to copy a block of data from one location in RAM to another location. An overlapping copy (when the destination is in the midst of the data being copied) is handled correctly so that the data to be copied is not overwritten.

All three parameters are converted internally to `UnsignedInteger`. Note that `MemCopy()` has the same functionality as `BlockMove()` but has a different parameter order; one that you may be accustomed to.

## Caution

This subroutine should be used with care because it is possible to overwrite important data on the stack or other areas of memory which may cause your program to malfunction.

## Example

```
Dim ba(1 to 10) as Byte
Dim ival as Integer

ba(3) = &H48
ba(4) = &H55
Call MemCopy(ival.DataAddress, ba(3).DataAddress, SizeOf(ival))
```

After execution, `ival` will have the value `&H5548`. Note the use of the `SizeOf()` function. This is a better programming practice than using a specific value because it makes the code easier to maintain.

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** BitCopy, MemCmp, MemSet

# MemSet

---

**Type**            Subroutine

**Invocation**    MemSet(addr, count, val)

| Parameter | Method | Type    | Description                           |
|-----------|--------|---------|---------------------------------------|
| addr      | ByVal  | int16   | The address of a block to initialize. |
| count     | ByVal  | int8/16 | The number of bytes to initialize.    |
| val       | ByVal  | Byte    | The initialization value.             |

## Discussion

This routine is useful for initializing arrays, buffers, etc. that reside in RAM.

## Example

```
Dim ba(1 to 20) as Byte

Call MemSet(MemAddress(ba), Sizeof(ba), &H55)
Call MemSet(ba.DataAddress, Sizeof(ba), 0)
```

## Caution

Using this routine to initialize data other than your own program variables may have detrimental effects.

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also**        MemCmp, MemCopy

# Mid

**Type** Function returning String

**Invocation** Mid(str, pos, length) or  
Mid(str, pos)

| Parameter | Method | Type    | Description   |
|-----------|--------|---------|---|
| str       | ByVal* | String  | The string from which to extract or modify a substring. |
| pos       | ByVal  | int8/16 | The position of the first character of the substring.   |
| length    | ByVal  | int8/16 | The length of the substring to extract or modify.       |

\* When used on the left hand side of an assignment, this parameter is passed ByRef.

## Discussion

This function can be used to extract a portion of a string or to modify a portion of a string, depending on how it is used. When it appears in the context of a function call, it returns a new string extracted from the string provided. The first character of the extracted substring will be the character at the position given by `pos` (where the first character of the string is position 1). The length of the returned string will be the number of characters in the source string beginning at the starting index through the end of the string or the specified length (if present), whichever is less. If the starting position is beyond the end of the string or if the specified length is less than or equal to zero, the returned string will be of zero length.

When used on the left hand side of an assignment operator, the `Mid()` function replaces a sequence of characters in a string with characters from the string value on the right hand side of the assignment operator.

```
Dim s as String
s = "abcdef"
Mid(str, 3) = "##" ' result is "ab##ef"
```

Note that when used in this way the first parameter is passed by reference so it cannot be a literal string or any other entity than cannot be passed by reference. Also, the length of the target string will never be changed. The number of characters overwritten in the destination string will be the lesser of a) the number of characters in the string on the right hand side of the assignment, b) the number of characters specified in the third parameter (if present), and c) the number of characters in the target string beginning at the position specified by the second parameter through the end of the string.

## Compatibility

In BasicX, the first parameter is pass-by-reference. This disallows any use of a string literal for the first parameter. Also, in BasicX the third parameter must always be provided.

The BasicX documentation suggests that using `Mid()` on the left hand side of an assignment might result in a change in the string length. Tests indicate that this is not the case. Moreover, execution of the code fragment below actually results in a garbage character being placed in the third character position.

```
Dim s as String
s = "abc"
Mid(s, 2, 2) = "!" ' result is "a!@" (@ is an indeterminate character)
```

**See Also** Left, Right, Trim

# MidWord

---

**Type**                Function returning UnsignedInteger

**Invocation**        MidWord(val)

| Parameter | Method | Type    | Description                                    |
|-----------|--------|---------|--|
| val       | ByVal  | numeric | The value of which the middle word is desired. |

## Discussion

This function returns the middle two bytes of a 4-byte value. If the specified value is a Byte the return value will be zero. If the specified value is contained in two bytes, the return value will have zero in the high byte.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            HiByte, HiWord, LoByte, LoWord

# Min

---

**Type**                Function (see discussion for the return type)

**Invocation**        Min(val1, val2)

| Parameter | Method | Type    | Description   |
|-----------|--------|---------|---|
| val1      | ByVal  | numeric | One of two values of which the smallest is desired. |
| val2      | ByVal  | numeric | One of two values of which the smallest is desired. |

## Discussion

This function returns the smaller of the two supplied values, both of which must be of the same type. If the supplied values are signed, the determination of which is smallest takes the sign of the values into account. The return value is the same type as the parameters.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            Max

# NoOp

---

**Type**            Subroutine

**Invocation**    NoOp()

## Discussion

This subroutine implements a delay of one CPU cycle, typically about 68nS.

## Compatibility

This function is only available for native code targets, e.g. the ZX-24n.

# OpenCom

**Type** Subroutine

**Invocation** OpenCom(channel, baud, inQueue, outQueue)

| Parameter | Method | Type          | Description                        |
|-----------|--------|---------------|------------------------------------|
| channel   | ByVal  | Byte          | The serial channel to open.        |
| baud      | ByVal  | Long          | The desired baud rate.             |
| inQueue   | ByRef  | array of Byte | The queue for incoming characters. |
| outQueue  | ByRef  | array of Byte | The queue for outgoing characters. |

## Discussion

This subroutine prepares a serial channel for use. If the specified channel is already open or if the channel number is invalid, it has no effect. The supported channel numbers are 1, 2 and 3-6 but you must have previously called `ComChannels()` in order to use channels 4-6.

The supported baud rates for Com1 (channel 1) and Com2 are the standard rates from 300 to 460,800 while the supported rates for Com3 to Com6 (channels 3-6) range from 300 to 19,200. However, if `ComChannels()` has been invoked, the maximum rate for channels 3-6 will be limited to that specified in the description of `ComChannels()`. Moreover, for channels 3-6 the baud rate for any given channel must be an integral divisor of the maximum rate.

The queues specified for the receive and transmit channels each must have been previously initialized by calling `OpenQueue()`. If you set up a transmit-only or receive-only serial channel you may use the value 0 for the unused queue. If you provide the value 0 for both queues, the channel will not be opened.

## Example

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

Call OpenQueue(outQueue, SizeOf(outQueue))
Call ComChannels(2, 9600)
Call DefineCom(4, 0, 12, &H08)
Call OpenCom(4, 9600, 0, outQueue)
```

The code above prepares Com4 as a transmit-only serial channel. If you wanted reception as well, you would have to declare and initialize a second queue and define the receive pin.

## Resource Usage

The hardware UARTs are assigned to channel numbers as shown in the table below.

| Hardware UART Assignment               |        |        |        |        |
|--|--------|--------|--------|--------|
| ZX Model                               | Com1   | Com2   | Com7   | Com8   |
| ZX-24, ZX40, ZX-44, ZX-24e             | USART0 | -      | -      | -      |
| ZX-24a, ZX40a, ZX-44a, ZX-24ae         | USART0 | -      | -      | -      |
| ZX-24p, ZX-40p, ZX-44p, ZX-24pe        | USART0 | USART1 | -      | -      |
| ZX-24n, ZX-40n, ZX-44n, ZX-24ne        | USART0 | USART1 | -      | -      |
| ZX-1281, ZX-1281n                      | USART1 | USART0 | -      | -      |
| ZX-1280, ZX-1280n                      | USART0 | USART1 | USART2 | USART3 |
| ZX-128e, ZX-128ne, ZX-1281e, ZX-1281ne | USART0 | USART1 | -      | -      |

It is important to note that on the ZX-24p and ZX-24n, the Com2 serial channel cannot be used at the same time as the hardware I2C channel because the pin 11 is shared between the TxD pin of Com2 and the SDA signal.

Com3 to Com6 are implemented as software UARTs using the Serial Timer (see table below) to regulate the bit timing. When one or more of the channels 3-6 are open the corresponding timer busy flag will be True indicating that Serial Timer is in use. When all of the channels 3-6 are closed, corresponding timer busy flag will again be False indicating that the Serial Timer is available for other uses.

| Serial Timer by Device                                    |              |
|---|--------------|
| ZX Model  | Serial Timer |
| ZX-24, ZX40, ZX-44, ZX-24e                                | Timer2       |
| ZX-24a, ZX40a, ZX-44a, ZX-24ae, ZX-24pe, ZX-24ne          | Timer2       |
| ZX-24p, ZX40p, ZX-44p, ZX-24n, ZX-40n, ZX-44n             | Timer2       |
| ZX-1281, ZX-1281n, ZX-1280, ZX-1280n, ZX-1281e, ZX-1281ne | Timer0       |
| ZX-128e, ZX-128ne   | Timer2       |

For native code devices, the table below indicates which ISRs may be loaded by using OpenCom() in your program. If the compiler cannot determine which specific channel is being opened, all of the listed ISRs will be included.

| ISRs Required  |             |                                   |
|----------------|-------------|-----------------------------------|
| Underlying CPU | Com Channel | ISR Name                          |
| mega644P       | Com1        | USART0_RX, USART0_TX, USART0_UDRE |
|                | Com2        | USART1_RX, USART1_TX, USART1_UDRE |
|                | Com3-Com6   | Timer2_CompA                      |
| mega128        | Com1        | USART0_RX, USART0_TX, USART0_UDRE |
|                | Com2        | USART1_RX, USART1_TX, USART1_UDRE |
|                | Com3-Com6   | Timer2_CompA                      |
| mega1281       | Com1        | USART1_RX, USART1_TX, USART1_UDRE |
|                | Com2        | USART0_RX, USART0_TX, USART0_UDRE |
|                | Com3-Com6   | Timer0_CompA                      |
| mega1280       | Com1        | USART0_RX, USART0_TX, USART0_UDRE |
|                | Com2        | USART1_RX, USART1_TX, USART1_UDRE |
|                | Com7        | USART2_RX, USART2_TX, USART2_UDRE |
|                | Com8        | USART3_RX, USART3_TX, USART3_UDRE |
|                | Com3-Com6   | Timer0_CompA                      |

Note, particularly, that the ISRs for Com1 are always included in every program even if OpenCom() is not explicitly invoked.

## Compatibility

In BasicX, the supported channel numbers are 1 to 3, depending on the particular target chip. Also, BasicX doesn't support the use of zero to indicate no queue is being supplied.

**See Also**      ComChannels, CloseCom, DefineCom, StatusCom



# OpenI2C

**Type** Subroutine

**Invocation** OpenI2C(channel, sdaPin, sclPin) or  
OpenI2C(channel, sdaPin, sclPin, bitRate)

| Parameter | Method | Type     | Description   |
|-----------|--------|----------|---|
| channel   | ByVal  | Byte     | The I2C channel to open (0-4).                        |
| sdaPin    | ByVal  | Byte     | The pin for the I2C data (SDA) signal.                |
| sclPin    | ByVal  | Byte     | The pin for the I2C clock (SCL) signal.               |
| bitRate   | ByVal  | integral | The optional clock speed designation, see discussion. |

## Discussion

This subroutine prepares an I2C channel for use. Five channels are supported, numbered 0 through 4. Channel zero uses the onboard hardware I2C controller while channels 1 through 4 are implemented in software. The I2C implementation does not support multi-master arbitration when operating in Master mode. Slave clock stretching is supported on both hardware and software channels.

For channel 0, the `sdaPin` and `sclPin` parameters are ignored since the hardware uses specific pins for the SDA and SCL signals (e.g. Port C, bits 1 and 0, respectively). For channels 1-4, the `sdaPin` and `sclPin` parameters specify the pins to use for the data and clock signals, respectively. In both cases, the clock and data pins are automatically configured for I2C operation. The I2C protocol requires pullup resistors on both of the lines, the value of which depends on characteristics of your system. A typical value is in the range of 1.5K to 4.7K.

The optional `bitRate` parameter allows you to control the speed of the data interchange. If the parameter is not given, the default speed is 100KHz. Each I2C device has a maximum clock rate at which it will operate reliably; check the datasheet of your selected device to determine the maximum rate.

The interpretation of the value of the `bitRate` parameter differs for channel 0 and channels 1-4. The tables below specify the values to use for several common clock speeds.

| I2C Channel 0 Clock Speeds |                         |   |
|----------------------------|-------------------------|---|
| bitRate Value              | Approximate Clock Speed | Notes                                   |
| 140                        | 50KHz                   |   |
| 66                         | 100KHz                  | Standard Low Speed, default speed       |
| 29                         | 200KHz                  |   |
| 11                         | 388KHz                  | Closest to Standard High Speed (400KHz) |
| 10                         | 410KHz                  | Highest supported speed                 |

| I2C Channels 1-4 Clock Speeds <sup>1</sup> |                         |                                   |
|--|-------------------------|-----------------------------------|
| bitRate Value                              | Approximate Clock Speed | Notes                             |
| 295  | 50KHz                   |                                   |
| 148  | 100KHz                  | Standard Low Speed, default speed |
| 74   | 200KHz                  |                                   |
| 59   | 250KHz                  | Highest supported speed           |

<sup>1</sup> The values given assume the default setting of `Register.TimerSpeed1`.

For channel 0, the `bitRate` parameter is a composite of two values: the value in the lower 8 bits is known as BR and is written to the processor's TWBR register. The low two bits of the high byte select a clock divisor according to the table below. The clock speed of the hardware channel is given by the

formula  $14.7456\text{MHz} / (16 + 2 * \text{BR} * \text{Divisor})$ . If the `bitRate` parameter is omitted or is zero the value of 66 is used by default.

| Channel 0 Prescaler Selector Value |         |
|------------------------------------|---------|
| Value                              | Divisor |
| 0                                  | 1       |
| 1                                  | 4       |
| 2                                  | 16      |
| 3                                  | 64      |

For channels 1-4 the `bitRate` parameter is interpreted as the number of I/O Timer ticks per bit. For I2C operations, the I/O Timer uses the prescaler specified by `Register.TimerSpeed1`. With the default prescaler of 1, each I/O Timer tick represents approximately 68nS. If the `bitRate` parameter is omitted or is zero the value of 74 is used by default. Due to processing overhead, the minimum attainable bit time is approximately 4µS.

For channel 0, the table below gives the pin numbers used for SDA and SCL.

| SDA and SCL Pins                       |         |         |
|--|---------|---------|
| ZX Models                              | SDA     | SCL     |
| ZX-24, ZX-24a, ZX-24p, ZX-24n          | 11, C.1 | 12, C.0 |
| ZX-40, ZX-40a, ZX-40p, ZX-40n          | 23, C.1 | 22, C.0 |
| ZX-44, ZX-44a, ZX-44p, ZX-44n          | 20, C.1 | 19, C.0 |
| ZX-24e, ZX-24ae, ZX-24pe, ZX-24ne      | 11, C.1 | 12, C.0 |
| ZX-1281, ZX-1281n                      | 26, D.1 | 25, D.0 |
| ZX-1280, ZX-1280n                      | 44, D.1 | 43, D.0 |
| ZX-128e, ZX-128ne, ZX-1281e, ZX-1281ne | 11, D.1 | 12, D.0 |

As noted above, the `sdaPin` parameter value is ignored when channel 0 is specified.

### Examples

```
Call OpenI2C(0, 0, 0)           ' open the hardware channel at 100KHz
Call OpenI2C(2, 19, 20)        ' open channel 2 using pins 19, 20
Call OpenI2C(1, C.3, A.1, 74)  ' open channel 1 at 200KHz
```

### Resource Usage

The I2C routines utilize the I/O Timer to regulate the bit timing for channels 1-4. While sending or receiving I2C data, the corresponding timer busy flag will be True indicating that the I/O Timer is in use. On the ZX-24p and ZX-24n, the hardware I2C channel cannot be used while Com2 is open since pin 11 is shared by the SDA signal and TxD for Com2.

### Compatibility

This subroutine is not available in BasicX compatibility mode.

**See Also**      `CloseI2C`, `I2CGetByte`, `I2CPutByte`, `I2CStart`, `I2CStop`, `I2CCmd`, `OpenI2CSlave`

# OpenI2CSlave

**Type** Subroutine

**Invocation** OpenI2CSlave(slaveAddr)

| Parameter | Method | Type | Description                                |
|-----------|--------|------|--|
| slaveAddr | ByVal  | Byte | The I2C slave address to which to respond. |

## Discussion

This subroutine immediately activates the I2C controller in slave mode. If you activate slave mode, you must also provide an interrupt handler for the `TWI` vector (aka the `I2C` vector). While slave mode is active, calls to `CmdI2C()` and the low level I2C commands are ineffective for I2C channel 0. Slave mode can be canceled by calling `CloseI2C()`.

While slave mode is active, the device will respond to reads and writes on the I2C bus referring to its slave address which is the value of the `slaveAddr` parameter with the least significant bit set to zero. If the least significant bit of the `slaveAddr` parameter is set, the slave can respond also to “general call” traffic on the bus.

The table below gives the pin numbers used for the SDA and SCL signals on various ZX devices.

| SDA and SCL Pins                       |         |         |
|--|---------|---------|
| ZX Models                              | SDA     | SCL     |
| ZX-24, ZX-24a, ZX-24p, ZX-24n          | 11, C.1 | 12, C.0 |
| ZX-40, ZX-40a, ZX-40p, ZX-40n          | 23, C.1 | 22, C.0 |
| ZX-44, ZX-44a, ZX-44p, ZX-44n          | 20, C.1 | 19, C.0 |
| ZX-24e, ZX-24ae, ZX-24pe, ZX-24ne      | 11, C.1 | 12, C.0 |
| ZX-1281, ZX-1281n                      | 26, D.1 | 25, D.0 |
| ZX-1280, ZX-1280n                      | 44, D.1 | 43, D.0 |
| ZX-128e, ZX-128ne, ZX-1281e, ZX-1281ne | 11, D.1 | 12, D.0 |

## Examples

```
Call OpenI2CSlave(&H50) ' activate I2C slave mode with address &H50
```

## Resource Usage

The I2C hardware channel cannot be opened by `OpenI2C()` while slave mode is active. On the ZX-24n, I2C slave mode cannot be used while `Com2` is open since pin 11 is shared by the SDA signal and `TxD` for `Com2`.

## Compatibility

This subroutine is only available for native mode devices.

**See Also** `CloseI2C`, `OpenI2C`

# OpenPWM

**Type** Subroutine

**Invocation** OpenPWM(channel, frequency, mode)

| Parameter | Method | Type   | Description                                  |
|-----------|--------|--------|--|
| channel   | ByVal  | Byte   | The channel to use for PWM generation.       |
| frequency | ByVal  | Single | The desired PWM frequency.                   |
| mode      | ByVal  | Byte   | The desired PWM mode (see discussion below). |

## Discussion

This subroutine prepares a PWM channel for generating a pulse width modulated (PWM) signal. PWM generation is performed using one of the CPU's 16-bit timers, the number of which varies depending on the ZX model. The table below indicates the available channels and the corresponding timer used. See the description of PWM() for additional details on the PWM channels.

| Supported PWM Channels  |         |         |         |            |
|---|---------|---------|---------|------------|
| ZX Models   | Timer1  | Timer3  | Timer4  | Timer5     |
| ZX-24, ZX-24a, ZX-24p, ZX-24n,<br>ZX-40, ZX-40a, ZX-40p, ZX-40n,<br>ZX-44, ZX-44a, ZX-44p, ZX-44n,<br>ZX-24e, ZX-24ae, ZX-24pe, ZX-24ne | 1, 2    | -       | -       | -          |
| ZX-1281, ZX-1281n,<br>ZX-128e, ZX-128ne, ZX-1281e, ZX-1281ne  | 1, 2, 3 | 4, 5, 6 | -       | -          |
| ZX-1280, ZX-1280n   | 1, 2, 3 | 4, 5, 6 | 7, 8, 9 | 10, 11, 12 |

The `frequency` parameter specifies the PWM base frequency in Hertz. Since the same frequency and generation mode will be used for all PWM channels based on the same timer, it is only necessary to call `OpenPWM()` once to prepare the timer for all of the PWM channels that are based on a given timer.

The `mode` parameter specifies the PWM generation mode. Two modes are supported: Fast PWM mode and Phase/Frequency Correct mode. The constants `zxFastPWM` and `zxCorrectPWM`, having the values 0 and 1 respectively, may be used to specify the mode. The Fast PWM mode has a maximum frequency of one-half of the CPU clock frequency and is intended for fixed-frequency applications. The Phase/Frequency Correct PWM mode has a maximum frequency of one-quarter of the CPU clock frequency and may be used when the PWM frequency will be changed in the midst of PWM signal generation. Frequency changes are effected by making additional calls to `OpenPWM()` and the change is synchronized so that it takes effect at the beginning of a cycle.

A side effect of calling `OpenPWM()` is that the timer busy flag for the underlying timer (e.g. `Register.Timer1Busy`) will be set to `True` irrespective of its prior state. It is recommended that the initial call to `OpenPWM()` be preceded by a call to acquire the semaphore for the timer. This will ensure that an existing timer operation will not be disturbed.

## Example

```
Call OpenPWM(1, 50.0, zxFastPWM) 'prepare for 50Hz Fast PWM using channel 1
```

## Compatibility

This subroutine is not available in BasicX compatibility mode.

**See Also** ClosePWM, PWM

# OpenQueue

**Type** Subroutine

**Invocation** OpenQueue(queue, size)  
OpenQueue(queue)

| Parameter | Method | Type          | Description                      |
|-----------|--------|---------------|----------------------------------|
| queue     | ByRef  | array of Byte | The queue to be initialized.     |
| size      | ByVal  | int16         | The size of the array, in bytes. |

## Discussion

This routine prepares a queue for use by initializing the management information contained in the queue data structure. The number of bytes of space available for data in a queue is the specified size less the queue management overhead (9 bytes). It may be convenient to use the built-in constant `System.MinQueueSize` in the definition of an array intended to hold a queue.

If the compiler can deduce the size of the array element, e.g. an explicitly dimensioned `Byte` array is specified, the second parameter may be omitted. In the case, the compiler utilizes the size of the array as the size parameter. Otherwise, the compiler will issue an error message indicating that the size must be explicitly specified.

## Caution

If you specify a size parameter that is larger than the actual size of the array, data following the array may be overwritten, usually with undesirable consequences. For this reason, it is recommended that you use the `SizeOf()` function to specify the queue size so that it will automatically track any changes that you make to the actual queue size. See the example below.

`OpenQueue()` should only be called for a queue that is not in use. Invoking it for a queue that is in use has undefined results.

## Example

```
Dim inQueue(1 to System.MinQueueSize + 20) as Byte

Call OpenQueue(inQueue, SizeOf(inQueue))
```

After the call to `OpenQueue()` the queue will ready to be used.

## Compatibility

BasicX allows any type for the first parameter. This implementation requires that it be an array of `Byte`. The second parameter must be supplied in BasicX mode.

# OpenSPI

**Type** Subroutine

**Invocation** OpenSPI(channel, flags, csPin)  
OpenSPI(channel, flags, csPin, rxDelay)

| Parameter | Method | Type | Description                                       |
|-----------|--------|------|---|
| channel   | ByVal  | Byte | The SPI channel to open (1-4).                    |
| flags     | ByVal  | Byte | Flags controlling the SPI communication.          |
| csPin     | ByVal  | Byte | The pin for the chip select signal to the device. |
| rxDelay   | ByVal  | Byte | The delay time prior to each received byte.       |

## Discussion

This subroutine prepares an SPI channel for use. Four channels are supported, numbered 1 through 4. It does not matter if the particular channel has been previously opened. The `flags` parameter specifies the characteristics of the SPI communication. They must be set to be compatible with the device with which you want to communicate. See the table below for details. The `csPin` parameter specifies the pin number that you wish to control the device's chip select input. The pin will be made an output and set to the inactive (high) state.

**SPI Channel Control Bits**

| Function              | Hex Value | Bit Mask    |
|-----------------------|-----------|-------------|
| Bit Rate f/4          | &H00      | xx xx xx 00 |
| Bit Rate f/16         | &H01      | xx xx xx 01 |
| Bit Rate f/64         | &H02      | xx xx xx 10 |
| Bit Rate f/128        | &H03      | xx xx xx 11 |
| Clock Phase False     | &H00      | xx xx x0 xx |
| Clock Phase True      | &H04      | xx xx x1 xx |
| Clock Low at Idle     | &H00      | xx xx 0x xx |
| Clock High at Idle    | &H08      | xx xx 1x xx |
| Bit Order – MSB first | &H00      | xx 0x xx xx |
| Bit Order – LSB first | &H20      | xx 1x xx xx |
| Double Speed          | &H80      | 1x xx xx xx |

The remaining bits are currently undefined but may be employed in the future. Bits 3 and 2 taken together specify the SPI mode 0-3, e.g. xx xx 00 xx specifies mode 0. Note that if the Double Speed bit is set, the SPI channel will run at twice the frequency specified by the two low order flag bits.

The `rxDelay` parameter, which defaults to zero if not present, specifies the amount of time to delay before beginning the SPI cycle for each byte received, if any, during the second half of the `SPICmd( )` process. See the description of `SPICmd( )` for more details.

## Caution

For ZX devices that use an external SPI EEPROM for user program storage, you must avoid doing anything that will interfere with the SPI commands to that device. SPI communication by direct manipulation of the processor SPI control registers is not supported and may cause your program to malfunction.

**Compatibility**

BasicX supports neither the double speed option nor the optional rxDelay parameter. Also, ZX devices based on the ATmega32 processor do not support the optional rxDelay parameter.

**See Also**      CloseSPI, OpenSPISlave, SPICmd

# OpenSPISlave

**Type** Subroutine

**Invocation** OpenSPISlave(flags)

| Parameter | Method | Type | Description                              |
|-----------|--------|------|--|
| flags     | ByVal  | Byte | Flags controlling the SPI communication. |

## Discussion

This subroutine, available only for native mode devices, immediately activates the SPI interface in slave mode. The `flags` parameter specifies the characteristics of the SPI communication. They must be set to be compatible with the SPI master with which you want to communicate. See the table below for details.

**SPI Slave Mode Configuration Bits**

| Function              | Hex Value | Bit Mask    |
|-----------------------|-----------|-------------|
| Clock Phase False     | &H00      | xx xx x0 xx |
| Clock Phase True      | &H04      | xx xx x1 xx |
| Clock Low at Idle     | &H00      | xx xx 0x xx |
| Clock High at Idle    | &H08      | xx xx 1x xx |
| Bit Order – MSB first | &H00      | xx 0x xx xx |
| Bit Order – LSB first | &H20      | xx 1x xx xx |

The chip select pin for an SPI slave is a dedicated pin; see the table below. If you activate slave mode, you must also provide an interrupt handler for the `SPI_STC` vector. While slave mode is active, `SPICmd()` calls are ineffective. Slave mode can be canceled by calling `CloseSPI()`.

**Slave Mode CS Pin**

| ZX Models           | CS Pin  |
|---------------------|---------|
| ZX-40n              | 5, B.4  |
| ZX-44n              | 44, B.4 |
| ZX-24ne             | 24, B.4 |
| ZX-1281n            | 10, B.0 |
| ZX-1280n            | 19, B.0 |
| ZX-128ne, ZX-1281ne | 28, B.0 |

Note that the SPI master sets the SPI clock speed. The highest SPI clock speed that can be used reliably is one quarter of the CPU clock speed of a ZX slave device. Depending on how much computation the slave must perform to prepare data for sending back to the master, a substantially slower SPI clock may need to be used. If a ZX device is being used as the master, it may be useful to set the `rxDelay` parameter on calls to `OpenSPI()` on the master to allow additional processing time.

## Compatibility

This subroutine is only supported for native mode devices.

**See Also** CloseSPI, OpenSPI



# OpenWatchDog

**Type** Subroutine

**Invocation** OpenWatchDog(timeout)

| Parameter | Method | Type | Description                                 |
|-----------|--------|------|---|
| timeout   | ByVal  | Byte | Specifies a timeout value (see discussion). |

## Discussion

This subroutine prepares the watchdog timer for use. Once it is opened, the `WatchDog()` routine must be called from time to time. If the period between `WatchDog()` calls exceeds the timeout value, the system will be reset.

The timeout value is 16.384 milliseconds times 2 to the N power where N is the value of the `timeout` parameter limited to the range shown in the table below. Note that the timeout value varies with processor voltage. It is slightly longer at 3.0 volts than at 5.0 volts. Consult the Atmel documentation for more specific information.

| WatchDog Timeout Parameter Range   |               |           |
|--|---------------|-----------|
| ZX Models  | Timeout Range | Max. Time |
| ZX-24, ZX-40, ZX-44  | 0-7           | 2 sec     |
| ZX-24a, ZX-40a, ZX-44a,<br>ZX-24p, ZX-40p, ZX-44p,<br>ZX-24n, ZX-40n, ZX-44n | 0-9           | 8 sec     |
| ZX-1281, ZX-1281n, ZX-1280, ZX-1280n   | 0-9           | 8 sec     |
| ZX-24e, ZX-128e, ZX-128ne  | 0-7           | 2 sec     |
| ZX-24ae, ZX-24pe, ZX-24ne, ZX-1281e, ZX-1281ne                               | 0-9           | 8 sec     |

When the processor is reset, the register value `Register.ResetFlags` contains bit flags indicating the source of the reset. It is important to note that the occurrence of a system fault (e.g. a stack overflow) will also cause a WatchDog reset as will calling `ResetProcessor()`. See the section on Run Time Stack Checking in the ZBasic Reference Manual for more information on stack overflow detection.

The watchdog timer can be turned off using `CloseWatchDog`.

## Compatibility

BasicX doesn't support `Register.ResetFlags` or `CloseWatchDog`.

**See Also** WatchDog, CloseWatchDog, ResetProcessor

# OpenX10

**Type** Subroutine

**Invocation** OpenX10(channel, inQueue, outQueue)

| Parameter | Method | Type          | Description                             |
|-----------|--------|---------------|---|
| channel   | ByVal  | Byte          | The X-10 communication channel to open. |
| inQueue   | ByRef  | array of Byte | The queue for incoming X-10 data.       |
| outQueue  | ByRef  | array of Byte | The queue for outgoing X-10 data.       |

## Discussion

This subroutine prepares an X-10 communication channel for use. After the channel is opened you can send arbitrary X-10 command bit streams, which you must create in low-level form, by simply adding the constituent bytes to the outgoing queue. Similarly, the incoming queue will receive raw X-10 data which you must decode. Each X-10 command begins with the bit sequence 1110 which is followed by additional bit pairs. The bit pair 01 represents a logic zero while the bit pair 10 represents a logic one. The bit pair 11 is invalid and the bit pair 00 signifies the end of a command bit stream and also represents the idle condition. Additional information on X-10 commands may be found in various places on the Internet.

If the specified channel is already open or if the channel number is invalid, it has no effect. The supported channel numbers are 1-2. The channel must have been previously configured by a call to `DefineX10()`. Also, the queues specified for the receive and transmit channels each must have been previously initialized by calling `OpenQueue()`. If you set up a transmit-only or receive-only serial channel you may use the value 0 for the unused queue. If you provide the value 0 for both queues, the channel will not be opened.

## Example

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

Call OpenQueue(outQueue, SizeOf(outQueue))
Call DefineX10(1, 0, 12, &H08)
Call OpenX10(1, 0, outQueue)
```

The code above prepares channel 1 as for transmit-only operation. If you wanted reception as well, you would have to declare and initialize a second queue and define the receive pin.

## Resource Usage

X-10 communication requires the use of the `Int0` line to which the X-10 zero-crossing signal must be connected. When one or more of the X-10 channels are open any task that waits for `Int0` using `WaitForInterrupt()` will be suspended indefinitely. When all X-10 channels are closed, `Int0` will again be available for `WaitForInterrupt()` use.

For native code devices, the following ISRs are required.

| ISRs Required  |                    |
|----------------|--------------------|
| Underlying CPU | ISR Name           |
| mega644P       | Timer0_CompB, INT0 |
| mega1281       | Timer2_CompB, INT0 |
| mega1280       | Timer2_CompB, INT0 |

**Compatibility**

This subroutine is not available on ZX models that are based on the ATmega32 processor (e.g. the ZX-24) or the ATmega128. Moreover, it is not available in BasicX compatibility mode.

**See Also**      CloseX10, DefineX10, StatusX10

# OutputCapture

**Type** Subroutine

**Invocation** OutputCapture(values, count, firstPulse)

| Parameter | Method | Type           | Description  |
|-----------|--------|----------------|--|
| intervals | ByRef  | array of int16 | The lengths of successive segments of the output waveform. |
| count     | ByVal  | int16          | The number of entries in the value array.                  |
| flags     | ByVal  | Byte           | Configuration bits controlling the generation process.     |

## Discussion

This subroutine produces a series of precisely timed logic levels on the OutputCapture pin (see table below) allowing you to produce an arbitrary waveform. Each entry in the `intervals` array specifies a time interval, in units of the I/O Timer period (by default, about 67.8ns), for each segment of the waveform. When called, the OutputCapture pin will be made an output and will be set to its initial state (the complement of the least significant bit of the `flags` parameter).

When waveform generation is begun, the OutputCapture pin will be changed to the opposite state for the interval specified by the first `intervals` element, changed to the opposite state again for the interval specified by the second `intervals` element, etc. for as many elements as specified. The final state of OutputCapture pin depends on whether the `count` parameter is odd or even. If it is odd the final state will be the complement of the least significant bit of the `flags` parameter; if it is even the final state will be the same as the least significant bit of the `flags` parameter.

The calling task will be suspended during the waveform generation process. If another task disables interrupts the accuracy of the generated waveform may suffer.

Due to processing overhead, the smallest pulse width that can be accommodated is about 6μS. This corresponds to a value of about 88 in the data array at the default timer speed. If the system has a heavy interrupt load (e.g. serial channels 3-6 are open) the minimum pulse width for reliable operation may be significantly larger. The maximum pulse width using the default timer speed is about 4.4mS. If you need to generate longer pulse widths, you may set the value of `Register.TimerSpeed1` so that a slower clock rate is used.

To avoid unwanted logic transitions on the OutputCapture pin during preparation for waveform generation, the OutputCapture pin should be configured as an input prior to the call. You'll probably need to employ a pullup or pulldown resistor on the pin to guarantee the desired logic state prior to the commencement of waveform generation.

## Resource Usage

This routine uses the I/O Timer. If the timer is already in use the routine will return immediately without performing the waveform generation. Also, this routine cannot be used at the same time as `InputCapture()`. See the description of `OutputCaptureEx()` for information about ISR requirements.

| Output Capture Pin                     |         |
|--|---------|
| ZX Models                              | Pin     |
| ZX-24, ZX-24a, ZX-24p, ZX-24n          | 27, D.4 |
| ZX-40, ZX-40a, ZX-40p, ZX-40n          | 18, D.4 |
| ZX-44, ZX-44a, ZX-44p, ZX-44n          | 13, D.4 |
| ZX-24e, ZX-24ae, ZX-24pe, ZX-24ne      | 16, D.4 |
| ZX-1281, ZX-1281n                      | 25, B.6 |
| ZX-1280, ZX-1280n                      | 16, B.6 |
| ZX-128e, ZX-1281e, ZX-128ne, ZX-1281ne | 22, B.6 |

## Compatibility

Since the CPU runs at twice the rate as the BasicX CPU, the units of the pulse width are half as long. If you need to generate longer pulse widths, you may set the value of `Register.TimerSpeed1` so that a slower clock rate is used. Also, the BasicX documentation indicates that if the I/O Timer is already in use, that use will be terminated and the waveform generation will be performed.

**See Also**      `OutputCaptureEx()`

# OutputCaptureEx

**Type** Subroutine

**Invocation** OutputCaptureEx(pin, intervals, count, flags)  
OutputCaptureEx(pin, intervals, count, flags, repeatCount)

| Parameter   | Method | Type           | Description  |
|-------------|--------|----------------|--|
| pin         | ByVal  | Byte           | Specifies the waveform output pin.                                   |
| intervals   | ByRef  | array of int16 | The lengths of successive segments of the output waveform.           |
| count       | ByVal  | any int        | The number of entries in the <code>intervals</code> array (1-65535). |
| flags       | ByVal  | Byte           | Configuration bits controlling the generation process.               |
| repeatCount | ByVal  | any int        | The number of times to repeat the pattern (1-65535).                 |

## Discussion

This subroutine produces a series of precisely timed logic levels on the specified pin allowing you to produce an arbitrary waveform. Each entry in the `intervals` array specifies a time interval, in units of the I/O Timer clock period (by default, about 67.8ns), for each segment of the waveform. When called, the specified pin will be made an output and will be set to its initial state (the complement of the least significant bit of the `flags` parameter).

When waveform generation is begun, the specified pin will be changed to the opposite state for the interval specified by the first `intervals` element, changed to the opposite state again for the interval specified by the second `intervals` element, etc. for as many elements as specified. The final state of the output pin depends on whether the `count` parameter is odd or even. If it is odd the final state will be the complement of the least significant bit of the `flags` parameter; if it is even the final state will be the same as the least significant bit of the `flags` parameter.

If the optional `repeatCount` parameter is not given a repeat count of 1 is assumed. If the repeat count is 1 the `intervals` array should generally have an odd number of values. This allows the output to end in the same state as it started. If the repeat count is greater than one the `intervals` array should generally have an even number of values. This allows the output waveform to repeat at the same logic levels. Also, when the waveform is repeated the last interval of the last cycle is omitted so that the output ends up in the same state as it started.

The calling task will be suspended during the waveform generation process. If another task disables interrupts, the accuracy of the generated waveform will suffer.

Due to processing overhead, the smallest pulse width that can be accommodated is about 6.8μS. This corresponds to a value of about 100 in the data array at the default timer speed. If the system has a heavy interrupt load (e.g. serial channels 3-6 are open) the minimum pulse width for reliable operation may be significantly larger. The maximum pulse width using the default timer speed is about 4.4mS. If you need to generate longer pulse widths, you may set the value of `Register.TimerSpeed1` so that a slower clock rate is used.

To avoid unwanted logic transitions on the output pin during preparation for waveform generation, the output pin should either be configured as an input or as an output in the desired starting state prior to the call. If you configure it as an input you'll probably need to employ a pullup or pulldown resistor on the pin to guarantee the desired logic state prior to the commencement of waveform generation.

Although this subroutine can be invoked specifying the hardware OutputCapture pin (see the table below) or any other I/O pin, the behavior when using a general I/O pin may be slightly different than when using the hardware OutputCapture pin. The hardware OutputCapture pin uses features of the hardware to toggle the I/O pin while for general I/O pins the pin is toggled in software by directly setting the corresponding PORTx bit. During periods of high interrupt load the hardware toggling will be more accurate.

## Resource Usage

This routine uses the I/O Timer. If the timer is already in use the routine will return immediately without performing the waveform generation. Also, this routine cannot be used at the same time as `InputCapture()` or `InputCaptureEx()` that requires the same timer.

**Hardware Output Capture Pin**

| <b>ZX Models</b>                       | <b>Timer 1 Pin</b> | <b>Timer 3 Pin</b> | <b>Timer 4 Pin</b> | <b>Timer 5 Pin</b> |
|--|--------------------|--------------------|--------------------|--------------------|
| ZX-24, ZX-24a, ZX-24p, ZX-24n          | 27, D.4            | -                  | -                  | -                  |
| ZX-40, ZX-40a, ZX-40p, ZX-40n          | 18, D.4            | -                  | -                  | -                  |
| ZX-44, ZX-44a, ZX-44p, ZX-44n          | 13, D.4            | -                  | -                  | -                  |
| ZX-24e, ZX-24ae, ZX-24pe, ZX-24ne      | 13, D.4            | -                  | -                  | -                  |
| ZX-1281, ZX-1281n                      | 16, B.6            | 6, E.4             | -                  | -                  |
| ZX-1280, ZX-1280n                      | 25, B.6            | 6, E.4             | 16, H.4            | 39, L.4            |
| ZX-128e, ZX-128ne, ZX-1281e, ZX-1281ne | 22, B.6            | 16, E.4            | -                  | -                  |

When performing an output capture on a general I/O pin, Timer1 will be used to generate the required timing. On ZX devices that have Timer3, it will be used if Timer1 is not available. If neither timer is available, the routine will return immediately.

For native code devices, the table below gives the ISRs that may be loaded if your program uses `OutputCapture()`. If the compiler cannot determine which specific timer ISR is required by analyzing the parameters used, all listed ISRs will be included.

**ISRs Required**

| <b>Underlying CPU</b> | <b>ISR Name</b>  |
|-----------------------|--|
| mega644P              | Timer1_CompB   |
| mega128               | Timer1_CompB,<br>Timer3_CompB  |
| mega1281              | Timer1_CompB,<br>Timer1_CompC,<br>Timer3_CompB                                   |
| mega1280              | Timer1_CompB,<br>Timer1_CompC,<br>Timer3_CompB,<br>Timer4_CompB,<br>Timer5_CompB |

## Compatibility

This routine is not available in BasicX compatibility mode.

# ParityCheck

---

**Type**            Function returning Boolean

**Invocation**    ParityCheck(data, oddParity)

| Parameter | Method | Type    | Description  |
|-----------|--------|---------|--|
| data      | ByVal  | Byte    | The data value for which to check the parity.                |
| oddParity | ByVal  | Boolean | The desired parity: True -> odd parity, False -> even parity |

## Discussion

This function computes the parity over the eight bits of the provided data value and compares that result to the desired result indicated by the `oddParity` parameter. The return value is a pass/fail indicator where True means that the parity matched the desired parity.

The data value has even parity if the number of one bits in the value is even.

## Example

```
Dim b as Byte

If Not ParityCheck(b, False) Then
    Debug.Print "Even parity check failed"
End If
```

## Compatibility

This routine is not available in BasicX compatibility mode.



# Pause

**Type** Subroutine

**Invocation** Pause(time)

| Parameter | Method | Type            | Description   |
|-----------|--------|-----------------|---|
| time      | ByVal  | Single or int16 | The amount of time to pause, in seconds (Single) or Timer 0 ticks (int16) |

## Discussion

This routine suspends execution of the current task for approximately the period of time specified. No other task is allowed to run during the pause period. The resolution of the time period is approximately 4.34µS with a maximum pause time of about 284mS. Note that the accuracy of the pause may be affected by the time required for the processor to service interrupts (RTC, serial channel, etc.). Also note that the resolution of the pause is similar to the minimum execution time for user instructions. This means that timing using `Pause()` calls of less than 20 to 50 units or so will be affected significantly by the succeeding instructions.

This routine should be used instead of `Sleep()` or `Delay()` when higher resolution timing is required or you don't want a task switch to occur. If you need longer pauses than can be produced by this routine, you can implement them using `Register.RTCStopWatch`.

## Example

```
Do
    Call PutPin(12, 0)
    Call Pause(0.010)      ' a 10 millisecond delay
    Call PutPin(12, 1)
    Call Pause(2304)       ' a 10 millisecond delay
Loop
```

This loop produces a square wave signal on pin 12 at approximately 50Hz (with some jitter due to handling interrupts).

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** Delay, DelayUntilClockTick, Sleep, WaitForInterval

# PeekQueue

**Type** Subroutine

**Invocation** PeekQueue(queue, var, count)

| Parameter | Method | Type          | Description                                 |
|-----------|--------|---------------|---|
| queue     | ByRef  | array of Byte | The queue from which to retrieve data.      |
| var       | ByRef  | any type      | The variable to receive the retrieved data. |
| count     | ByVal  | int16         | The number of bytes to retrieve.            |

## Discussion

This routine will copy the specified number of bytes from the queue to the indicated variable but it does not remove them from the queue. The routine will not return until it can copy the entire number of bytes specified. Because of this, you should usually check the number of bytes available in the queue using `GetQueueCount()` before calling `PeekQueue()`.

Note that if the calling task is locked and the queue contains insufficient data when this routine is called, the task will be unlocked to allow other tasks to run.

## Caution

If the requested number of bytes is larger than the queue capacity, the routine will never return. Likewise, if not enough data is placed in the queue, the routine will never return. Also, if the variable to receive the data is smaller than the number of bytes indicated, adjacent memory will be overwritten, usually with undesirable results.

## Example

## Compatibility

BasicX allows any type for the first parameter. This implementation requires that it be an array of `Byte`.

# PersistentPeek

---

**Type**            Function returning Byte

**Invocation**    PersistentPeek(address)

| Parameter | Method | Type  | Description                                       |
|-----------|--------|-------|---|
| address   | ByVal  | int16 | The persistent memory address from which to read. |

## Discussion

This function will return the content of the specified persistent memory address.

The address of any persistent variable can also be obtained using the `DataAddress` property. For persistent variables, the `DataAddress` property is of type `UnsignedInteger`.

## Example

```
Dim pi as PersistentInteger
Dim b as Byte
b = PersistentPeek(1000)
b = PersistentPeek(pi.DataAddress + 1)
```

The second use of `PersistentPeek()` demonstrates how you can use the `DataAddress` property to read a byte value from any part of a persistent variable of any type.

## Compatibility

BasicX does not support the use of the `DataAddress` property for persistent items.

The BasicX system has only 512 bytes of persistent memory. This implementation has 1024 bytes of persistent memory of which the first 32 are reserved for system use.

**See Also**            PersistentPoke

# PersistentPoke

---

**Type** Subroutine

**Invocation** PersistentPoke(value, address)

| Parameter | Method | Type  | Description                                      |
|-----------|--------|-------|--|
| value     | ByVal  | Byte  | The to write to persistent memory.               |
| address   | ByVal  | int16 | The persistent memory address to which to write. |

## Discussion

This routine will write the given value to the specified persistent memory address.

The address of any persistent variable can also be obtained using the `DataAddress` property. For persistent variables, the `DataAddress` property is of type `UnsignedInteger`.

## Caution

The first 32 bytes of persistent memory are reserved for the system. Modifying any of them may produce unpredictable results.

The persistent memory (on-board EEPROM) has a limit specified by the manufacturer of a million write cycles. When this limit is exceeded the memory may become unreliable.

## Example

```
Dim pi as PersistentInteger
Call PersistentPoke(&H55, 1000)
Call PersistentPoke(&H55, pi.DataAddress + 1)
```

The second use of `PersistentPoke()` demonstrates how you can use the `DataAddress` property to write a byte value to any part of a persistent variable of any type.

## Compatibility

BasicX does not support the use of the `DataAddress` property for persistent items.

The BasicX system has only 512 bytes of persistent memory. This implementation has 1024 bytes of persistent memory of which the first 32 are reserved for system use.

**See Also** PersistentPeek

# PlaySound

**Type** Subroutine

**Invocation** PlaySound(pin, address, length, rate, repeat)

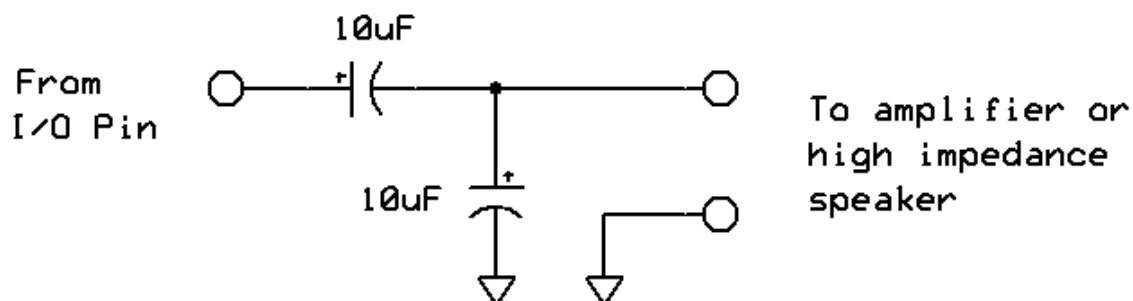
| Parameter | Method | Type  | Description                              |
|-----------|--------|-------|--|
| pin       | ByVal  | Byte  | The output pin.                          |
| address   | ByVal  | int16 | The EEPROM address of the sound data.    |
| length    | ByVal  | int16 | The number of bytes of sound data.       |
| rate      | ByVal  | int16 | The sample rate for the sound data.      |
| repeat    | ByVal  | int16 | The number of times to repeat the sound. |

## Discussion

This routine uses a pseudo-PWM technique to create an approximation to a sine wave on the specified output pin. The frequency of the sine wave is given by successive bytes in EEPROM (user program memory) beginning at the specified address and continuing for the given length. The `rate` parameter specifies the rate at which the data elements will be utilized. It is equivalent to the sample rate at which an original analog sound might have been digitized. Lastly, the `repeat` parameter tells how many times to repeat the production of the output using the supplied data. If zero is specified, the sound will be repeated 65,536 times.

The minimum supported sample rate is 250Hz. If a smaller value is specified, 250Hz will be used instead.

The actual output will be a pulse stream that has an average value that approximates the target analog signal. Because of the high frequency nature of the pulse train used to synthesize the waveform some filtering is required. The example circuit below may be used to couple the output to a high impedance speaker (> 40  $\Omega$ ) or an amplifier. Note, however, that the signal is too large to be fed to the microphone input of an amplifier. Instead, the Auxiliary or Line input should be used.



## Resource Usage

This routine uses the I/O Timer and disables interrupts during the generation process. In particular, this means that serial input that arrives during the generation will likely be missed and serial output on channels 3-6 will be disrupted.

Task switching is suspended and other interrupts are disabled while the sound is being produced. However, RTC ticks are accumulated during the process and the RTC is updated when the process has completed so that the RTC does not lose time.

### **Example**

```
Dim music as ByteVectorData("sound.txt")
```

```
Call PlaySound(12, LoWord(music.DataAddress), UBound(music), 11025, 1)
```

This example assumes that you have prepared the file "sound.txt" to contain the digitized music, sampled at 11025Hz.

### **Compatibility**

The BasicX documentation for `PlaySound()` does not explicitly indicate that a zero repeat count will result in 65,536 iterations. However, experimental evidence indicates that it does.

In the BasicX implementation the RTC will lose time if the duration is too long. It is not known if the BasicX implementation has a minimum sample rate.

# PortBit

**Type**                      Function returning Byte

**Invocation**            PortBit(portIdx, bitIdx)  
PortBit(pin)

| Parameter | Method | Type     | Description                              |
|-----------|--------|----------|--|
| portIdx   | ByVal  | integral | The I/O port designator (A=0, B=1, etc.) |
| bitIdx    | ByVal  | integral | The bit designator (0-7)                 |
| pin       | ByVal  | integral | A pin number                             |

## Discussion

This function returns a composite value that describes a specific bit in a specific I/O port. The fields of the Byte value are as shown in the table below.

| Bit(s) | Description                              |
|--------|--|
| 7      | Always 1                                 |
| 6-3    | The I/O port designator (A=0, B=1, etc.) |
| 2-0    | The bit designator (0-7)                 |

When invoked in the first form with the parameter values 2 and 6 (representing Port C, bit 6) the return value will have the bit pattern &B10010110.

The second form of invocation converts a physical pin number to the composite value representing the port and bit corresponding to that pin. When passed an invalid pin, the return value is zero.

Values returned by the PortBit() function may be used anywhere that a pin number may be used, e.g. as the first parameter to PutPin(). The primary advantage to using the composite port/bit designator is that the same value may be used unchanged on any ZX device.

Note that the special port/bit designators like `C.2` are converted by the compiler to the same type of composite port/bit designator described here if the compiler directive `Option PortPinEncoding On` is specified.

## Compatibility

This function is not available in BasicX compatibility mode.

# Pow

**Type**            Function returning Single

**Invocation**    Pow(mantissa, exponent)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| mantissa  | ByVal  | Single | The value to be raised to the power given by the exponent. |
| exponent  | ByVal  | Single | The exponent value.  |

## Discussion

This function returns the value of the first parameter raised to the power given by the second parameter. This is the same functionality as provided by the exponentiation operator ^.

Certain special cases are detected as shown in the table below.

| Mantissa  | Exponent           | Result    |
|-----------|--------------------|-----------|
| any value | 0.0                | 1.0       |
| negative  | non-integral value | NaN       |
| 0.0       | Negative           | +Infinity |

## Example

```
Dim r as Single, f as Single  
  
f = 10.0  
r = Pow(f, 2.0)    ' result is 100.0
```

**See Also**        Exp, Exp10



# PulseIn (subroutine form)

**Type** Subroutine

**Invocation** PulseIn(pin, level, var)

| Parameter | Method | Type   | Description                                    |
|-----------|--------|--------|--|
| pin       | ByVal  | Byte   | The pin on a pulse width will be measured.     |
| level     | ByVal  | Byte   | The expected pulse logic value (high = 1).     |
| var       | ByRef  | Single | The variable to receive the pulse width value. |

## Discussion

This routine waits for the input pin to be in the idle state (the opposite of that specified by the `level` parameter), waits for it to change to the specified logic level and then measures the time that it stays at that level. The pulse width is stored in the specified variable and has units of seconds with a default resolution of approximately 1.085µs.

The pin is made an input if it is not already so. If the awaited logic transition never occurs or if the pulse width exceeds the maximum representable width the stored result will be zero.

The timing resolution may be adjusted using `Register.TimerSpeed2`. However, if this is done, the resulting pulse width value will need to be scaled proportionally. Note that for compatibility with BasicX, the timing resolution is one half of the period of the selected I/O Timer frequency.

## Resource Usage

This routine uses the I/O Timer and interrupts are disabled during the pulse measurement. However, RTC ticks will be accumulated during the pulse measurement and the RTC will be updated when the process is complete.

## Example

```
Dim width as Single
Call PulseIn(12, 1,width)      ' measure a positive-going pulse
```

## Compatibility

The BasicX implementation does not support adjustable timing resolution.

# PulseIn (function form)

**Type**                Function returning Integer

**Invocation**        PulseIn(pin, level)

| Parameter | Method | Type | Description                                |
|-----------|--------|------|--|
| pin       | ByVal  | Byte | The pin on a pulse width will be measured. |
| level     | ByVal  | Byte | The expected pulse logic value (high = 1). |

## Discussion

This routine waits for the input pin to be in the idle state (the opposite of that specified by the `level` parameter), waits for it to change to the specified logic level and then measures the time that it stays at that level. The width of the pulse is returned by the function, the units of which are 2 times the I/O Timer period. At the default I/O Timer clock period of 0.54μS, the returned value has units of 1.08μS.

The pin is made an input if it is not already so. If the awaited logic transition never occurs or if the pulse width exceeds the maximum representable width the returned value will be zero.

The timing resolution may be adjusted using `Register.TimerSpeed2`. Note that for compatibility with BasicX, the timing resolution is one half of the period of the selected I/O Timer frequency.

## Resource Usage

This routine uses the I/O Timer and interrupts are disabled during the pulse measurement. However, RTC ticks will be accumulated during the pulse measurement and the RTC will be updated when the process is complete.

## Example

```
Dim width as Integer  
  
i = PulseIn(12, 1)            ' measure a positive pulse
```

## Compatibility

The BasicX implementation does not support adjustable timing resolution.

# PulseOut

**Type** Subroutine

**Invocation** PulseOut(pin, duration, level)

| Parameter | Method | Type            | Description  |
|-----------|--------|-----------------|--|
| pin       | ByVal  | Byte            | The pin on which a pulse width will be generated.  |
| duration  | ByVal  | int16 or Single | The width of the generated pulse.                  |
| level     | ByVal  | Byte            | The desired pulse logic value (low = 0, high = 1). |

## Discussion

This routine first makes the specified pin an output. (However, for practical purposes, you should generally make the pin an output and set it to the desired state before calling this routine.) Then it sets the pin to the active state (as indicated by the `level` parameter), waits the specified time and then sets the pin back to the inactive state. The pin will be left configured as an output.

The pulse width may be specified by a `Single` value with units of seconds and a resolution of approximately 1.085µs (however, due to processing overhead, the shortest pulse that can be generated is slightly less than 2µs). Alternately, the pulse width may be specified by an `Integer` or `UnsignedInteger` value with units of 2 x I/O Timer ticks (by default, 1.085µs). Note, however, that `Register.TimerSpeed2` may be modified to adjust the I/O Timer tick rate. If this is done, the `Single` value will have to be scaled proportionally.

If the output pin is specified as zero, this routine does not generate a pulse but will delay for approximately the specified period of time. This may be useful for generating a delay with better precision than can be obtained by using `Delay()` or `Sleep()`. Moreover, generating a delay in this manner does not cause the task to lose control.

## Resource Usage

This routine uses the I/O Timer and interrupts are disabled during the pulse generation. However, RTC ticks will be accumulated during the pulse generation and the RTC will be updated when the process is complete. If the pulse is too long characters being sent or received on serial channels 3-6 may be garbled. See the

## Example

```
Dim width as Integer
```

```
Call PutPin(12, zxOutputLow)
Call PulseOut(12, 2, 1)      ' generate a positive pulse about 2µS long
Call PulseOut(0, 1e-5, 0)    ' generate a delay of about 10µS
```

## Compatibility

In the BasicX implementation the RTC will lose time if the pulse is too long.

The BasicX implementation does not support adjustable timing resolution.

# Put1Wire

---

**Type** Subroutine

**Invocation** Put1Wire(pin, value)

| Parameter | Method | Type | Description                        |
|-----------|--------|------|------------------------------------|
| pin       | ByVal  | Byte | The pin to be used for 1-Wire I/O. |
| value     | ByVal  | Byte | The bit value to write.            |

## Discussion

This routine sends the LSB of the given value using the 1-Wire protocol. To perform a 1-Wire operation, this function along with related 1-Wire routines must be used in the proper sequence. See the specifications of your 1-Wire device for more information.

## Resource Usage

This routine uses the I/O Timer and disables interrupts for about 100µS.

## Example

```
Call Put1Wire(12, 1)
```

**See Also** Get1Wire, Get1WireByte, Get1WireData, Put1WireByte, Put1WireData, Reset1Wire

# Put1WireByte

---

**Type** Subroutine

**Invocation** Put1WireByte(pin, value)

| Parameter | Method | Type | Description                        |
|-----------|--------|------|------------------------------------|
| pin       | ByVal  | Byte | The pin to be used for 1-Wire I/O. |
| value     | ByVal  | Byte | The value to write.                |

## Discussion

This routine sends a byte (LSB first) using the 1-Wire protocol. To perform a 1-Wire operation, this function along with related 1-Wire routines must be used in the proper sequence. See the specifications of your 1-Wire device for more information.

## Example

```
Call Put1WireByte(12, &H55)
```

## Resource Usage

This routine uses the I/O Timer and disables interrupts for about 100µS for each bit sent.

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** Get1Wire, Get1WireByte, Get1WireData, Put1Wire, Put1WireData, Reset1Wire

# Put1WireData

---

**Type** Subroutine

**Invocation** Put1WireData(pin, data, count)

| Parameter | Method | Type     | Description                            |
|-----------|--------|----------|--|
| pin       | ByVal  | Byte     | The pin to be used for 1-Wire I/O.     |
| data      | ByRef  | any type | A variable holding the bytes to write. |
| count     | ByVal  | Byte     | The number of bytes to write.          |

## Discussion

This routine sends 1 or more bytes of data (each LSB first) using the 1-Wire protocol. To perform a 1-Wire operation, this function along with related 1-Wire routines must be used in the proper sequence. See the specifications of your 1-Wire device for more information.

## Example

```
Dim d(1 to 10) As Byte

Call Put1WireData(12, d, 5)
```

## Resource Usage

This routine uses the I/O Timer and disables interrupts for about 100µS for each bit sent.

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** Get1Wire, Get1WireByte, Get1WireData, Put1Wire, Put1WireByte, Reset1Wire

# PutBit

---

**Type**            Subroutine

**Invocation**    PutBit(var, bitNumber, val)

| Parameter | Method | Type     | Description                                    |
|-----------|--------|----------|--|
| var       | ByRef  | any type | The variable to which the bit will be written. |
| bitNumber | ByVal  | int8/16  | The bit number to write.                       |
| val       | ByVal  | Byte     | The bit value.                                 |

## Discussion

This routine writes a single bit to memory beginning at the location of the specified variable. Bit numbers 0-7 are written to the byte at the specified location, bit numbers 8-15 are written to the subsequent byte, etc. In each case, the lower bit number corresponds to the least significant bit of the byte while the highest bit number corresponds to the most significant bit of a byte.

Only the least significant bit of the `val` parameter is used; the remaining bits are ignored.

## Caution

If you specify a bit number beyond the number of bits in the specified variable, a byte in memory following the variable will be modified, perhaps with undesirable results.

## Compatibility

In BasicX compatibility mode, the `bitNumber` parameter may only be specified using a `Byte` value.

**See Also**        GetBit

# PutDAC

**Type** Subroutine

**Invocation** PutDAC(pin, dacValue, dacAccumulator)  
PutDAC(pin, dacValue, dacAccumulator, cycles)

| Parameter      | Method | Type    | Description  |
|----------------|--------|---------|--|
| pin            | ByVal  | Byte    | The output pin.  |
| dacValue       | ByVal  | numeric | The desired output value. See discussion below.            |
| dacAccumulator | ByRef  | Byte    | A value used in the DAC process. See the discussion below. |
| cycles         | ByVal  | Byte    | The number of PWM cycles to perform.                       |

## Discussion

This routine creates a digital approximation of an analog signal on the specified pin using a pseudo-PWM technique. When called, the specified pin is made an output, a pulse train is generated having an average value equal to the `dacValue` parameter and then, after a fixed number of iterations, the pin is placed in the high impedance input state. If the output is filtered with a low pass filter, the voltage will, immediately after the process is completed, be at a level between zero and the processor voltage (usually +5 volts). However, the voltage will begin to decay at a rate dependent on the load presented to the filter. The voltage can be refreshed from time to time by calling `PutDAC()` again.

The `dacValue` parameter may be specified by a `Single` value or an integral value. If a `Single` value is supplied, it should be in the range 0.0 to 1.0 corresponding to the output range of 0 to the processor voltage (usually +5 volts). If an integral value is supplied, it should be in the range of 0 to 255 corresponding to the same output voltage range as above.

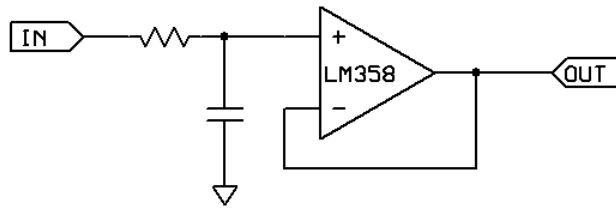
The `dacAccumulator` parameter is required to ensure continuity between successive calls to `PutDAC()`. The value of the parameter after the call should not be modified and the same parameter should be supplied on the next call. The initial value of the parameter is of no consequence. If your application uses `PutDAC()` to create an analog voltage on more than one pin at a time, a separate accumulator value must be used for each one.

If the `cycles` parameter is not specified, a single PWM cycle is performed. Each cycle will generate a burst of pulses for about 200 $\mu$ S during which time interrupts will be disabled. At the end of each cycle, the pin is put in high impedance mode and interrupts are re-enabled. The process is then repeated if the cycle count is greater than one. A cycle count of zero causes no cycles to be performed.

The selection of components for the required filter depends on several factors. A larger capacitor will allow the voltage to hold longer but also takes longer to bring up to the proper voltage. As a rule of thumb, the product of the resistance (in ohms) and the capacitance (in farads) should be on the order of the number of cycles times 50 $\mu$ S. For example, with a 100 resistor and a 1 $\mu$ F capacitor, the cycle count should probably be 2 in order to bring the capacitor up to the desired voltage level.

For best results, you should probably follow the filter with a high impedance buffer such as a unity gain op amp circuit, an example of which is shown below. The op amp chosen is not particularly critical, nearly any will do the job.





## Examples

```
Dim acc as Byte
```

```
Call PutDAC(12, 0.5, acc)
```

```
Call PutDAC(12, 128, acc, 5)
```

## Compatibility

In BasicX compatibility mode, the `dacValue` parameter may only be specified using a `Single` value. Also, the fourth parameter is not supported.

## Resource Usage

This routine disables interrupts for about 200µS during the generation process. Interrupts are reenabled between each successive cycle.

**See Also**      DACPin

# PutDate

---

**Type** Subroutine

**Invocation** PutDate(year, month, day)

| Parameter | Method | Type  | Description                 |
|-----------|--------|-------|-----------------------------|
| year      | ByVal  | int16 | The year value (1999-2177). |
| month     | ByVal  | Byte  | The month value (1-12).     |
| day       | ByVal  | Byte  | The day value (1-31).       |

## Discussion

This routine composes a new value for `Register.RTCDay` using the provided parameters. The month value of 1 corresponds to January while 12 corresponds to December. If the year or month is invalid or if the day number is invalid for the specified month and year, `Register.RTCDay` will be set to zero.

Note that `Register.RTCDay` is initialized to zero on power-up or reset. This corresponds to January 1, 1999.

**See Also**      `GetDate`

# PutEEPROM

---

**Type** Subroutine

**Invocation** PutEEPROM(addr, var, count)

| Parameter | Method | Type     | Description   |
|-----------|--------|----------|---|
| addr      | ByVal  | Long     | The Program Memory address at which to begin writing.         |
| var       | ByRef  | any type | The variable from which the data to be written will be taken. |
| count     | ByVal  | int16    | The number of bytes to write.                                 |

## Discussion

This routine is provided for compatibility with BasicX. The more aptly named PutProgMem() should be used by new applications.

**See Also** GetProgMem, PutProgMem

# PutNibble

---

**Type** Subroutine

**Invocation** PutNibble(var, nibbleNumber, val)

| Parameter    | Method | Type     | Description                                       |
|--------------|--------|----------|---|
| var          | ByRef  | any type | The variable to which the nibble will be written. |
| nibbleNumber | ByVal  | int8/16  | The nibble number to write.                       |
| val          | ByVal  | Byte     | The nibble value.                                 |

## Discussion

This routine writes a single nibble (four bits) to memory beginning at the location of the specified variable. Nibble numbers 0-1 are written to the byte at the specified location, nibble numbers 2-3 are written to the subsequent byte, etc. In each case, the lower nibble number corresponds to the least significant four bits of the byte while the higher nibble number corresponds to the most significant four bits of the byte.

Only the least significant four bits of the `val` parameter is used; the remaining bits are ignored.

## Caution

If you specify a nibble number beyond the number of nibbles in the specified variable, a byte in memory following the variable will be modified, perhaps with undesirable results.

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** GetNibble

# PutPersistent

---

**Type** Subroutine

**Invocation** PutPersistent(addr, var, count)

| Parameter | Method | Type     | Description                                      |
|-----------|--------|----------|--|
| addr      | ByVal  | int16    | The Persistent Memory address to which to write. |
| var       | ByRef  | any type | The variable from which data will be taken.      |
| count     | ByVal  | int8/16  | The number of bytes to write.                    |

## Discussion

This routine reads one or more bytes from RAM and writes them to Persistent Memory beginning at the address given.

## Caution

Persistent Memory 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.

A block of Persistent Memory starting at address zero is reserved for system use. When the compiler assigns addresses to persistent variables defined in your program, the lowest address used is the first address above this reserved block. The .map file generated by the compiler contains a section indicating the addresses assigned to persistent variables defined in your program. The built-in values `Register.PersistentStart`, `Register.PersistentSize` and `Register.PersistentUsed` may be useful for determining the allocated and unallocated portions of Persistent Memory.

This routine will write to any address in Persistent Memory. Generally, you should avoid writing to the reserved area of Persistent Memory.

## Example

```
Dim pvar(1 to 10) as PersistentByte
Dim var(1 to 10) as Byte

Call PutPersistent(pvar.DataAddress, var, SizeOf(pvar))
```

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** GetPersistent

# PutPin

**Type** Subroutine

**Invocation** PutPin(pin, mode)

| Parameter | Method | Type | Description                         |
|-----------|--------|------|-------------------------------------|
| pin       | ByVal  | Byte | The pin to configure.               |
| mode      | ByVal  | Byte | The configuration mode (see below). |

## Discussion

This routine is used to configure a pin to be an input or an output or to effect a change in the output logic level. If the pin is configured as an input, it may be configured to be in “tri-state” mode or “pull-up” mode. If the pin is configured to be an output, the output level may be set to zero or 1. The table below gives the values for each of the possible modes. If an invalid mode is specified or an invalid pin is specified, the routine has no effect.

**Values for the mode Parameter**

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

Note that for modes 4 and 5 to be useful, the pin must have been previously set to be an output. Mode 4 (zxOutputToggle) will change the output to the opposite logic level. Mode 5 (zxOutputPulse) will change the output to the opposite level for a short period of time and then change it back to the original level. The duration of the pulse will be about 8 CPU cycles (approximately 0.5uS at 14.7456MHz).

## Example

```
Call PutPin(12, zxOutputLow) ' pin 12 will be at logic zero
```

## Compatibility

Modes 4 and 5 are not available in BasicX.

**See Also** GetPin

# PutProgMem

---

**Type** Subroutine

**Invocation** PutProgMem(addr, var, count)

| Parameter | Method | Type     | Description   |
|-----------|--------|----------|---|
| addr      | ByVal  | Long     | The Program Memory address to which to begin writing.         |
| var       | ByRef  | any type | The variable from which the data to be written will be taken. |
| count     | ByVal  | int16    | The number of bytes to write.                                 |

## Discussion

This routine writes one or more bytes to Program Memory (where the user program is stored) taking the data from RAM beginning at the location of the specified variable. Note that if a number of bytes is specified that is larger than the given variable, adjacent memory will be read.

## Caution

Program Memory has a write cycle limit specified by the manufacturer of a million cycles. Writing to a particular address in excess of this limit may result in unreliable operation.

**See Also** GetProgMem

# PutQueue

**Type** Subroutine

**Invocation** PutQueue(queue, var, count)

| Parameter | Method | Type          | Description  |
|-----------|--------|---------------|--|
| queue     | ByRef  | array of Byte | The queue to which to write data.                                |
| var       | ByRef  | any type      | The variable from which to read data to be written to the queue. |
| count     | ByVal  | int16         | The number of bytes to write to the queue.                       |

## Discussion

This routine reads data from the variable and writes it to the specified queue. If there is insufficient space in the queue, the calling task will suspend until space becomes available. Note, particularly, that no data will be written until there is room for all the data to be written. This has two important ramifications.

Firstly, if the number of bytes to be written is larger than the data capacity of the queue, the write will never complete. Secondly, if data is never taken out of the queue thus making room for the additional data, the write will also never complete.

Note that the number of bytes to write may be larger than the named variable. If this is the case, data will be taken from subsequent memory locations until the write count is satisfied. This may or may not be what you intended to occur.

Note, also, that before any queue operations are performed, the queue data structure must be initialized. See the discussion of `OpenQueue()` for more details.

## Example

```
Dim outQueue(1 to 40) as Byte
Dim lval as Long

Call OpenQueue(outQueue, SizeOf(outQueue))
lval = &H55aa
Call PutQueue(outQueue, lval, SizeOf(lval))
```

## Compatibility

BasicX allows any type for the first parameter. This implementation requires that it be an array of Byte.

**See Also** PutQueueByte, PutQueueStr



# PutQueueByte

**Type** Subroutine

**Invocation** PutQueueByte(queue, val)

| Parameter | Method | Type          | Description                                |
|-----------|--------|---------------|--|
| queue     | ByRef  | array of Byte | The queue to which to write data.          |
| val       | ByVal  | Byte          | The byte value to be written to the queue. |

## Discussion

This routine writes the given byte value to the specified queue. If there is insufficient space in the queue, the calling task will suspend until space becomes available. This means that if data is never taken out of the queue thus making room for additional data, the process will never complete.

Note that before any queue operations are performed, the queue data structure must be initialized. See the discussion of `OpenQueue()` for more details.

## Example

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

Call OpenQueue(outQueue, SizeOf(outQueue))
Call PutQueueByte(outQueue, &H55)
```

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** OpenQueue, PutQueue, PutQueueStr

# PutQueueStr

**Type** Subroutine

**Invocation** PutQueueStr(queue, str)

| Parameter | Method | Type          | Description                            |
|-----------|--------|---------------|--|
| queue     | ByRef  | array of Byte | The queue to which to write data.      |
| str       | ByVal  | String        | The string to be written to the queue. |

## Discussion

This routine writes the characters from the string to the specified queue. If there is insufficient space in the queue, the calling task will suspend until space becomes available. Note, particularly, that no data will be written until there is room for all the data to be written. This has two important ramifications. Firstly, if the number of bytes to be written is larger than the data capacity of the queue, the write will never complete. Secondly, if data is never taken out of the queue thus making room for the additional data, the write will also never complete.

Note that before any queue operations are performed, the queue data structure must be initialized. See the discussion of `OpenQueue( )` for more details.

## Example

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

Call OpenQueue(outQueue, SizeOf(outQueue))
Call PutQueueStr(outQueue, "Hello, world!")
```

## Compatibility

BasicX allows any type for the first parameter. This implementation requires that it be an array of Byte.

**See Also** PutQueueByte, PutQueue, OpenQueue

# PutTime

---

**Type** Subroutine

**Invocation** PutTime(hour, minute, seconds)

| Parameter | Method | Type   | Description                       |
|-----------|--------|--------|-----------------------------------|
| hour      | ByVal  | Byte   | The hour value (0-23).            |
| minute    | ByVal  | Byte   | The minutes value (0-59).         |
| seconds   | ByVal  | Single | The seconds value (0.0 to 59.999) |

## Discussion

This routine combines the given values into the corresponding RTC tick count and stores the result in `Register.RTCTick`. Each parameter that is outside its corresponding legal range is considered to be zero.

Note that `Register.RTCTick` is initialized to zero on power-up or reset. This corresponds to 0:00:00.

**See Also** GetTime

# PutTimeStamp

---

**Type** Subroutine

**Invocation** PutTimeStamp(year, month, day, hour, minute, seconds)

| Parameter | Method | Type   | Description                 |
|-----------|--------|--------|-----------------------------|
| year      | ByVal  | int16  | The year value (1999-2177). |
| month     | ByVal  | Byte   | The month value (1-12).     |
| day       | ByVal  | Byte   | The day value (1-31).       |
| hour      | ByVal  | Byte   | The hour value (0-23).      |
| minute    | ByVal  | Byte   | The minutes value (0-59).   |
| seconds   | ByVal  | Single | The seconds value.          |

## Discussion

This routine combines the given date values into the corresponding `Register.RTCDay` value and combines the given time values into the corresponding RTC tick count and stores the result in `Register.RTCTick`. The effect is the same as if `PutDate()` and `PutTime()` had been called with their respective parameters.

Note that `Register.RTCDay` and `Register.RTCTick` are initialized to zero on power-up or reset.

# PWM

**Type** Subroutine

**Invocation** PWM(channel, dutyCycle)

| Parameter | Method | Type               | Description                            |
|-----------|--------|--------------------|--|
| channel   | ByVal  | Byte               | The channel to use for PWM generation. |
| dutyCycle | ByVal  | Single or integral | The desired duty cycle.                |

## Discussion

This subroutine begins or modifies the generation of a PWM signal on the specified channel. The channel must have been previously prepared for PWM generation by calling `OpenPWM( )`. PWM generation is performed using one of the CPU's 16-bit timers, the number of which varies depending on the ZX model. For ZX models based on the mega32, mega644 and mega644P CPUs, there is one 16-bit timer that jointly supports two PWM channels, numbered 1 and 2. For ZX models based on the mega128 and mega1281 CPUs, there are two 16-bit timers and each jointly supports three PWM channels, numbered 1 through 3 and 4 through 6 respectively. For ZX models based on the mega1280 CPU, there are four 16-bit timers and each jointly supports three PWM channels. The table below indicates the output pin for each PWM supported channel.

**Output Pins for PWM Channel Numbers**

| ZX Models                                 | 1       | 2       | 3       | 4       | 5       | 6       |
|---|---------|---------|---------|---------|---------|---------|
| ZX-24, ZX-24a, ZX-24p, ZX-24n             | 26, D.5 | 27, D.4 | -       | -       | -       | -       |
| ZX-40, ZX-40a, ZX-40p, ZX-40n             | 19, D.5 | 18, D.4 | -       | -       | -       | -       |
| ZX-44, ZX-44a, ZX-44p, ZX-44n             | 14, D.5 | 13, D.4 | -       | -       | -       | -       |
| ZX-24e, ZX-24ae, ZX-24pe, ZX-24ne         | 15, D.5 | 16, D.4 | -       | -       | -       | -       |
| ZX-1281, ZX-1281n                         | 15, B.5 | 16, B.6 | 17, B.7 | 5, E.3  | 6, E.4  | 7, E.5  |
| ZX-1280, ZX-1280n                         | 24, B.5 | 25, B.6 | 26, B.7 | 5, E.3  | 6, E.4  | 7, E.5  |
| ZX-128e, ZX-128ne,<br>ZX-1281e, ZX-1281ne | 23, B.5 | 22, B.6 | 21, B.7 | 17, E.3 | 16, E.4 | 15, E.5 |

| ZX Models         | 7       | 8       | 9       | 10      | 11      | 12      |
|-------------------|---------|---------|---------|---------|---------|---------|
| ZX-1280, ZX-1280n | 38, L.3 | 39, L.4 | 40, L.5 | 16, H.4 | 17, H.5 | 18, H.6 |

The `dutyCycle` parameter specifies the desired duty cycle of the generated signal, expressing the percentage of time that the PWM signal will be at the logic 1 state. If the supplied parameter is of type `Single`, the value is in percent with a resolution of 0.01%. If the supplied parameter is integral, the units are percent, i.e., the value 100 means 100%. Specifying a `Single` value that is negative or any value greater than 100 will have an undefined effect.

If this subroutine is called without a preceding call to `OpenPWM( )` to prepare the timer, the call will have no effect. This subroutine may be called multiple times to effect changes to the PWM signal's duty cycle while the signal is being generated. The change in duty cycle is synchronized so that it takes effect at the beginning of the next PWM pulse.

## Example

```
Call OpenPWM(2, 50.0, zxFastPWM) ' prepare for 50Hz Fast PWM using channel 2
Call PWM(2, 7.5)                  ' generate PWM with 7.5% duty cycle (1.5mS)
Call Delay(1.0)
Call PWM(2, 6.25)                  ' generate PWM with 6.25% duty cycle (1.25mS)
```

**Compatibility**

This subroutine is not available in BasicX compatibility mode.

**See Also**      ClosePWM, OpenPWM

# RadToDeg

---

**Type**              Function returning Single

**Invocation**      RadToDeg(angle)

| Parameter | Method | Type   | Description                                   |
|-----------|--------|--------|---|
| angle     | ByVal  | Single | The angle, in radians, to convert to degrees. |

## Discussion

The trigonometric functions in the System Library all use radian angle measure. Depending on the programming task, it is sometimes more convenient to think of angles in terms of degrees. This function and its companion `DegToRad()` facilitate the conversion between the two systems.

Depending on optimization settings, if the parameter supplied to this function is known to be constant at compile time, the compiler will convert the value at compile time. Otherwise, code is generated to perform the conversion (multiplication by a conversion factor) at run time.

## Example

```
Dim f as Single
Dim theta as Single      ' the angle in degrees

theta = RadToDeg(Asin(f))
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**              DegToRad

# RamPeek

---

**Type**            Function returning Byte

**Invocation**    RamPeek(address)

| Parameter | Method | Type  | Description                         |
|-----------|--------|-------|-------------------------------------|
| address   | ByVal  | int16 | The RAM address from which to read. |

## Discussion

This function will return the content of the specified RAM address.

## Example

```
Dim b as Byte
Dim i as Integer

b = RamPeek(MemAddress(i))
b = RamPeek(i.DataAddress)
```

**See Also**            RamPeekDword, RamPeekWord



# RamPeekDword

---

**Type**            Function returning UnsignedLong

**Invocation**    RamPeekDword(address)

| Parameter | Method | Type  | Description                         |
|-----------|--------|-------|-------------------------------------|
| address   | ByVal  | int16 | The RAM address from which to read. |

## Discussion

This function will return the 4-byte value at the specified RAM address. The first byte will be the low order byte and the last will be the high order byte.

## Example

```
Dim ul as UnsignedLong  
  
ul = RamPeekDWord(200)
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        RamPeek, RamPeekWord

# RamPeekWord

---

**Type**            Function returning UnsignedInteger

**Invocation**    RamPeekWord(address)

| Parameter | Method | Type  | Description                         |
|-----------|--------|-------|-------------------------------------|
| address   | ByVal  | int16 | The RAM address from which to read. |

## Discussion

This function will return the 2-byte value at the specified RAM address. The first byte will be the low order byte and the following will be the high order byte.

## Example

```
Dim u as UnsignedInteger  
  
u = RamPeekWord(200)
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        RamPeek, RamPeekDword

# RamPoke

---

**Type** Subroutine

**Invocation** RamPoke(value, address)

| Parameter | Method | Type  | Description                        |
|-----------|--------|-------|------------------------------------|
| value     | ByVal  | Byte  | The value to write to RAM.         |
| address   | ByVal  | int16 | The RAM address to which to write. |

## Discussion

This routine will write the given value to the specified RAM address.

## Caution

Modifying user variables in this way may cause your program to malfunction. Writing to areas of RAM used by the system may cause your program to malfunction.

## Examples

```
Dim b as Byte
```

```
Call RamPoke(&H55, MemAddress(b))
```

```
Call RamPoke(&H55, b.DataAddress)
```

**See Also** RamPokeDword, RamPokeWord

# RamPokeDword

**Type** Subroutine

**Invocation** RamPokeDword(value, address)

| Parameter | Method | Type       | Description                        |
|-----------|--------|------------|------------------------------------|
| value     | ByVal  | any 32-bit | The value to write to RAM.         |
| address   | ByVal  | int16      | The RAM address to which to write. |

## Discussion

This routine will write the given value to the four bytes at the specified RAM address, least significant byte first.

## Caution

Modifying user variables in this way may cause your program to malfunction. Writing to areas of RAM used by the system may cause your program to malfunction.

## Example

```
Dim ul as UnsignedLong

Call RamPokeDword(&H117355aa, MemAddress(ul))
Call RamPokeDword(&H117355aa, ul.DataAddress)
```

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** RamPoke, RamPokeWord

# RamPokeWord

---

**Type** Subroutine

**Invocation** RamPokeWord(value, address)

| Parameter | Method | Type  | Description                        |
|-----------|--------|-------|------------------------------------|
| value     | ByVal  | int16 | The value to write to RAM.         |
| address   | ByVal  | int16 | The RAM address to which to write. |

## Discussion

This routine will write the given value to the two bytes at the specified RAM address, least significant byte first.

## Caution

Modifying user variables in this way may cause your program to malfunction. Writing to areas of RAM used by the system may cause your program to malfunction.

## Example

```
Dim u as UnsignedInteger  
  
Call RamPokeWord(&H55aa, MemAddress(u))
```

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** RamPoke, RamPokeDword

# Randomize

---

**Type**            Subroutine

**Invocation**     Randomize()

## Discussion

This routine seeds the random number generator with the value of Register.RTCTick. This is can be used to introduce some randomness into the sequence of values returned by `Rnd ( )` especially if the time that `Randomize()` gets called has some uncertainty due to external events, e.g. the time that a user takes to press a key.

**See Also**        Rnd

# RCTime (subroutine form)

**Type** Subroutine

**Invocation** RCTime(pin, level, interval)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| pin       | ByVal  | Byte   | The pin to use.  |
| level     | ByVal  | Byte   | The expected initial logic level of the pin.                   |
| interval  | ByRef  | Single | The variable in which to return the charge/discharge interval. |

## Discussion

This routine measures how long the specified pin stays at the given logic level after it is made a tri-state input. The return value is expressed in seconds with a resolution of about 1.085 $\mu$ s by default but this can be changed using `Register.TimerSpeed2`. If the maximum time elapses (32,767 units times the resolution) and the pin has not changed logic levels, the return value will be zero. If the pin is not at the specified level when called, the routine returns immediately with a value of approximately 1.085e-6. The pin will be left in the input tri-state mode.

This function can be used with an external resistor-capacitor circuit to measure the value of one element when the other one is known. The charge/discharge time depends on the values of R and C as well as the initial and final voltages. Before calling this routine, you should make the specified pin an output and set it to the level specified.

## Resource Usage

This routine uses the I/O Timer. If the timer is already in use when this routine is called, it will return immediately with a zero value. The same is true if the specified pin is invalid.

Task switching is suspended and interrupts are disabled while the charge/discharge time is being measured. However, RTC ticks are accumulated during the process and the RTC is updated when the process has completed so that the RTC does not lose time.

## Example

See the function form of this routine for more information.

## Compatibility

In BasicX, the ability to change the resolution using `Register.TimerSpeed2` is not supported.

The BasicX documentation indicates that the maximum value that can be returned is about 71ms. In this implementation, the maximum value that can be returned is about 35.6 corresponding to 35.6ms at standard resolution. The resolution can be changed by modifying `Register.TimerSpeed2` which will affect the maximum time value.

The BasicX implementation will miss RTC ticks if the charge/discharge time is too long.

# RCTime (function form)

**Type**            Function returning Integer

**Invocation**    RCTime(pin, level)

| Parameter | Method | Type | Description                                  |
|-----------|--------|------|--|
| pin       | ByVal  | Byte | The pin to use.                              |
| level     | ByVal  | Byte | The expected initial logic level of the pin. |

## Discussion

This function measures how long the specified pin stays at the given logic level after it is made a tri-state input. The return value has units of about 1.085 $\mu$ S by default but this can be changed using `Register.TimerSpeed2`. If the maximum time elapses (32,767 units) and the pin has not changed logic levels, the return value will be zero. If the pin is not at the specified level when called, the routine returns immediately with a value of 1. The pin will be left in the input tri-state mode.

As an example, this function can be used with an external resistor-capacitor circuit to measure the value of one element when the other one is known. The charge/discharge time depends on the values of R and C as well as the initial and final voltages. Before calling this routine, you should make the specified pin an output and set it to the level specified.

## Example

```
Const pin as Byte = 12
```

```
Call PutPin(pin, 1)    ' make the pin an output high to start charging
Call Delay(1.4e-4)    ' delay a bit to allow nearly full charging
i = RCTime(pin, 1)    ' measure the time to reach logic zero level
```

## Resource Usage

This routine uses the I/O Timer. If the timer is already in use when this routine is called, it will return immediately with a zero value. The same is true if the specified pin is invalid.

Task switching is suspended and interrupts are disabled while the charge/discharge time is being measured. However, RTC ticks are accumulated during the process and the RTC is updated when the process has completed so that the RTC does not lose time.

## Compatibility

In BasicX, the ability to change the resolution using `Register.TimerSpeed2` is not supported.

The BasicX implementation will miss RTC ticks if the charge/discharge time is too long.



# Reset1Wire

**Type**            Function returning Byte

**Invocation**    Reset1Wire(pin)

| Parameter | Method | Type | Description                        |
|-----------|--------|------|------------------------------------|
| pin       | ByVal  | Byte | The pin to be used for 1-Wire I/O. |

## Discussion

This function generates a reset signal on the given pin using the 1-Wire protocol. The return value is the “presence” bit sent by the attached 1-Wire device(s), if any. It will be zero if a 1-Wire device responded, 1 otherwise.

To perform a 1-Wire operation, this function along with related 1-Wire routines must be used in the proper sequence. See the specifications of your 1-Wire device for more information.

## Resource Usage

This routine uses the I/O Timer and disables interrupts for approximately 1mS.

## Example

```
Dim b as Byte  
  
b = Reset1Wire(12)
```

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also**            Get1Wire, Get1WireByte, Get1WireData,  
Put1Wire, Put1WireByte, Put1WireData

# ResetProcessor

---

|                   |                  |
|-------------------|------------------|
| <b>Type</b>       | Subroutine       |
| <b>Invocation</b> | ResetProcessor() |

## Discussion

Calling this routine will cause a WatchDog reset of the processor within 20ms. When the processor begins running again, the value of `Register.ResetFlags` will indicate that a WatchDog reset has occurred. If you need to be able to distinguish between an actual WatchDog reset and a call to `ResetProcessor()` it is recommended that you define a persistent variable and set its value to indicate the source of the reset.

## Compatibility

BasicX does not support `Register.ResetFlags`.

# ResumeTask

**Type** Subroutine

**Invocation** ResumeTask(taskStack)  
ResumeTask()

| Parameter | Method | Type          | Description                       |
|-----------|--------|---------------|-----------------------------------|
| taskStack | ByRef  | array of Byte | The stack for a task of interest. |

## Discussion

This routine attempts to change the status of a task to a ready-to-run state. If no task stack is explicitly given, the task stack for the `Main()` routine is assumed. The table below shows the effect for various task states (as returned by `StatusTask()`).

**Effect of Resuming a Task in Various States**

| Status | State                                   | Effect   |
|--------|---|--|
| 0      | Ready to run.                           | None, the task is already ready to run.                                |
| 1      | Sleeping.                               | The task is awakened.  |
| 2      | Awaiting <code>InputCapture()</code> .  | The task resumes as if the <code>InputCapture()</code> had completed.  |
| 3      | Awaiting interrupt 0.                   | The task resumes as if the interrupt had occurred.                     |
| 4      | Awaiting interrupt 1.                   | The task resumes as if the interrupt had occurred.                     |
| 5      | Awaiting interrupt 2.                   | The task resumes as if the interrupt had occurred.                     |
| 6      | Awaiting interval expiration.           | The task resumes as if the interval had expired.                       |
| 7      | Awaiting analog compare.                | The task resumes as if the comparison interrupt had occurred.          |
| 8      | Awaiting pin change event 0.            | The task resumes as if the pin change had occurred.                    |
| 9      | Awaiting pin change event 1.            | The task resumes as if the pin change had occurred.                    |
| 10     | Awaiting pin change event 2.            | The task resumes as if the pin change had occurred.                    |
| 11     | Awaiting pin change event 3.            | The task resumes as if the pin change had occurred.                    |
| 12     | Awaiting <code>OutputCapture()</code> . | The task resumes as if the <code>OutputCapture()</code> had completed. |
| 13     | Awaiting interrupt 3.                   | The task resumes as if the interrupt had occurred.                     |
| 14     | Awaiting interrupt 4.                   | The task resumes as if the interrupt had occurred.                     |
| 15     | Awaiting interrupt 5.                   | The task resumes as if the interrupt had occurred.                     |
| 16     | Awaiting interrupt 6.                   | The task resumes as if the interrupt had occurred.                     |
| 18     | Awaiting interrupt 7.                   | The task resumes as if the interrupt had occurred.                     |
| 254    | Task exiting.                           | None, exiting tasks can't be resumed.                                  |
| 255    | Terminated.                             | None, halted tasks can't be resumed.                                   |

If this routine is invoked using an array other than one that is or was being used for a task stack the result is undefined. See the section on Task Management in the ZBasic Reference Manual for additional information regarding task management.

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** ExitTask, RunTask, StatusTask

# Right

**Type**            Function returning String

**Invocation**    Right(str, length)

| Parameter | Method | Type    | Description  |
|-----------|--------|---------|--|
| str       | ByVal  | String  | The string from which to extract characters.         |
| length    | ByVal  | int8/16 | The number of characters to extract from the string. |

## Discussion

This function returns a string consisting of the rightmost characters of the string passed as the first parameter. The maximum number of characters in the returned string is the smaller of 1) the number of characters in the passed string and 2) the value of the second parameter. Internally, the length is interpreted as a 16-bit signed value and negative values are treated as zero.

This function produces the same result as `Mid(str, Len(str) - length + 1, length)` assuming that the passed string is at least `length` characters long.

## Example

```
Dim s as String, s2 as String

s = "Hello, world!"
s2 = Right(s, 6)           ' the result will be "world!"
```

**See Also**        Left, Mid, Trim

# Rnd

---

**Type**                Function returning Single

**Invocation**        Rnd()

## Discussion

This function will return a pseudo-random value in the range of 0.0 to 1.0. The first time that `Rnd()` is called after the processor starts up the pseudo-random number generator is initialized with a seed value. The sequence of values returned will be repeatable when starting from the same seed.

You can alter the sequence of returned values in two ways. Firstly, you can set the value of `Register.SeedPRNG`. The next call to `Rnd()` will initialize the pseudo-random number generator with that seed value before returning the first random value. The second way to modify the sequence is to call the `Randomize()` subroutine. Doing so will initialize the pseudo-random number generator with the current value of `Register.RTCTick`. This provides a way to introduce some non-repeatability into the sequence of values returned by `Rnd()`. It is especially effective if the time at which `Randomize()` is called is controlled by some external, unpredictable event like a user pressing a key.

## Example

```
Dim i as Integer

' print 10 random values
For i = 1 to 10
    Debug.Print CStr(Rnd())
Next
```

## Compatibility

BasicX does not support `Register.seedPRNG`. Instead, it has a system global variable named `seedPRNG`. This built-in variable is also supported in ZBasic for compatibility.

**See Also**            Randomize

# RunTask

---

**Type** Subroutine

**Invocation** RunTask(taskStack)  
RunTask()

| Parameter | Method | Type          | Description                       |
|-----------|--------|---------------|-----------------------------------|
| taskStack | ByRef  | array of Byte | The stack for a task of interest. |

## Discussion

Calling this routine alters the normal task rotation regimen by immediately attempting to run the specified task or, if no task stack is explicitly given, the `Main()` task. If the specified task cannot run (because it is sleeping, waiting for InputCapture, etc.) the list of tasks is examined in order beginning with the task immediately following the specified task and the first ready-to-run task that is found will be run.

Because this routine interferes with the normal task rotation it must be used carefully to avoid starving out one or more tasks. If this routine is invoked using an array other than one that is or was being used for a task stack the result is undefined.

See the section on Task Management in the ZBasic Reference Manual for additional information regarding task management.

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** ExitTask, ResumeTask, StatusTask

# Semaphore

**Type** Function returning Boolean

**Invocation** Semaphore(var)

| Parameter | Method | Type    | Description                     |
|-----------|--------|---------|---------------------------------|
| var       | ByRef  | Boolean | A variable used as a semaphore. |

## Discussion

This function will test the provided variable and if it is already True, the function will return False. Otherwise, if the semaphore variable is False, the call will set it to True and return True. This is referred to in computer science as an “atomic test and set” operation.

A semaphore is a signaling and synchronization mechanism used in multi-tasking systems. The idea is that if two or more tasks each want to use a particular resource they first request ownership of a semaphore. The request mechanism ensures that even if multiple requests occur near the same time, one and only one request will be satisfied. Therefore, the task that is granted the semaphore will have exclusive access to the resource until it has completed its objective. Subsequently, other tasks can request the semaphore and, if they receive it, they can perform their objective. Thus you can see that a particular semaphore can control access to some set of resources that you define. Your system may have multiple semaphores, each controlling access to a set of resources. Note, however, that if multiple semaphores are required to complete an operation the possibility of deadlock exists. This problem will occur if one task obtains one semaphore, another task obtains another semaphore and then both tasks wait for the other semaphore to be available.

In order for this mechanism to be effective, the same semaphore variable must be used by each task for gaining access to a particular set of resources. For this reason, the semaphore variable passed to `Semaphore()` will almost always be a global variable but it may be public or private as suits your application. The semaphore variable must be initially False, otherwise no `Semaphore()` request on that semaphore can ever succeed. Also, after a task has successfully gotten the semaphore and has finished using the related resources, the semaphore must be set False again so that a future `Semaphore()` call will succeed.

## Example

```
Dim serSem as Boolean

serSem = False

' wait until we get the semaphore
Do While (Not Semaphore(serSem))
    Call Delay(0.5)
Loop

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

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

# SerialNumber

---

**Type**            Subroutine

**Invocation**    SerialNumber(serNum)

| Parameter | Method | Type          | Description   |
|-----------|--------|---------------|---|
| serNum    | ByRef  | array of Byte | The array to which the serial number will be written. |

## Discussion

A call to this routine will copy six bytes of serial number information to the provided array. At present, only three of the bytes are defined, representing the version number of the system firmware (for VM mode devices) or the ZX library code (for native mode devices). The first byte is the major version number, the second is the minor version number and the third byte is the variant number. The remaining bytes are undefined.

## Caution

If the array provided is less than 6 bytes long, subsequent memory will be overwritten, possibly with detrimental results.

## Compatibility

The serial number of this implementation may be different than that of BasicX.



# SetBits

**Type** Subroutine

**Invocation** SetBits(target, mask, value)

| Parameter | Method | Type | Description                               |
|-----------|--------|------|---|
| target    | ByRef  | Byte | The byte to be modified.                  |
| mask      | ByVal  | Byte | The mask indicating which bits to modify. |
| value     | ByVal  | Byte | The value of the bits to store.           |

## Discussion

This subroutine allows you to set the value of one or more bits in a byte while leaving others unchanged. Effectively, the result is the same as using the statement below.

```
target = (target And Not mask) Or (value And mask)
```

The `mask` parameter governs which bits will get updated. For each bit of the `mask` parameter that is a 1, the corresponding bit of the `target` will be set to the state of the corresponding bit of the `value` parameter. Bits of the `target` that correspond to zero bits of the `mask` parameter will remain unchanged.

The advantage to using the `SetBits()` subroutine instead of the equivalent statement is twofold. Firstly, it is more efficient, resulting in less code and faster execution time. Secondly, and perhaps more importantly, it performs the action as an atomic operation, i.e. one that is guaranteed, once begun, to complete without an intervening task switch. This characteristic makes `SetBits()` useful for modifying I/O ports and other `Byte` values in a multi-tasking environment.

## Example

```
' set the middle 4 bits of Port C to the binary value &B0110  
Call SetBits(Register.PortC, &H3C, &H18)
```

## Compatibility

This routine is not available in BasicX compatibility mode. Also, it is only supported by ZX firmware later than v1.0.0.

**See Also** ToggleBits

# SetInterval

---

**Type** Subroutine

**Invocation** SetInterval(interval)

| Parameter | Method | Type            | Description   |
|-----------|--------|-----------------|---|
| interval  | ByVal  | Single or int16 | The interval counter period, in RTC ticks (if an integral value is specified) or seconds (if a <code>Single</code> value is given). |

## Discussion

This routine sets the period of the built-in interval counter. On each RTC tick, the interval counter will be decremented. When it gets to zero, it is reloaded with the specified value and it begins to count down again. Furthermore, if a task is awaiting the interval expiration, it is immediately scheduled for execution (unless a higher priority task requires service). If no task is awaiting the interval expiration, the fact that the interval counter expired is recorded. Subsequently, a task may request a wait on the interval and, depending on the nature of the request, the task may be immediately triggered or it may await the next interval expiration.

Internally the interval period is stored as a 16-bit unsigned integer value. This limits the interval period to a maximum of slightly less than 128 seconds. Of course, longer interval periods may be effectively implemented by maintaining a counter and taking action after the expiration of a number interval periods.

## Example

```
Call SetInterval(200)      'about 391 milliseconds
Call SetInterval(10.0)    'about 10 seconds
```

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also**      WaitForInterval

# SetJump

---

**Type**                Function returning Integer

**Invocation**        SetJump(jmpbuf)

| Parameter | Method | Type          | Description   |
|-----------|--------|---------------|---|
| jmpbuf    | ByRef  | array of Byte | A buffer to hold the return context, see description below. |

## Discussion

This function, in conjunction with `LongJump()`, provides a way to circumvent the normal call-return structure and return directly to a distant caller. It is the equivalent of a non-local Goto function and can be used, among other things, to handle exceptions in your programs. The parameter specifies a `Byte` array that will be initialized with context information to allow a direct return from deeply nested calls. The array must be a minimum size (either 6 bytes or 24 bytes for VM mode and native mode, respectively) to hold the context information for unwinding the call stack. You can use the built-in constant `System.JumpBufSize` to ensure that it is the proper size.

On the initial call to `SetJump()` the return value will always be zero. When control is returned via a call to `LongJump()`, the return value will be the value supplied as the second parameter to the `LongJump()` call. Generally, you should choose this value to indicate the nature of the exception condition and in most cases it should be non-zero.

The jump buffer needs to be accessible to the `LongJump()` caller. Often, this is realized by making it a global or module-level variable. If you want it to be a local variable, you'll have to pass the buffer as a parameter down the call chain. See the section on Exception Handling in the ZBasic Reference Manual for more details.

## Caution

If the provided array is less than minimum required size, adjacent memory locations will be modified usually with undesirable results. Your application should not directly modify the contents of the array. Doing so may cause unpredictable behavior.

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also**            `LongJump`

# ShiftIn

**Type**                      Function returning Byte

**Invocation**            ShiftIn(dataPin, clkPin, bitCnt)

| Parameter | Method | Type | Description                               |
|-----------|--------|------|---|
| dataPin   | ByVal  | Byte | The pin used to input data.               |
| clkPin    | ByVal  | Byte | The pin used to output a clocking signal. |
| bitCnt    | ByVal  | Byte | The number of bits to read in (1 to 8).   |

## Discussion

This function can be used to input data from a synchronous serial device like a shift register. The pin specified for input will be made an input but the pin specified for the clock signal must already be an output and be at the desired initial logic level.

For each of the number of bits specified, then the clock line will be pulsed by changing its logic level twice. The data line will be sampled approximately 2 CPU clock cycles after the leading edge of the clock pulse. With a 14.7456MHz CPU clock, this equates to about 135nS after the leading edge.

The returned value consists of the data bits read with the bit first read in the most significant bit position. This is referred to as MSB first. If fewer than 8 bits are read, the low order bits will be zero.

## Resource Usage

This subroutine uses the I/O Timer. If the I/O Timer is already in use, the function returns immediately and the return value is zero. No other use of this resource should be attempted while the shifting is in progress. Interrupts are disabled during the shifting process. However, RTC ticks are accumulated during the shifting process so the RTC should not lose time.

## Compatibility

For compatibility with I2C/TWI devices the clock rate is approximately 200kHz with `Register.TimerSpeed1` at its default value of 1. If the value of `Register.TimerSpeed1` is changed, the bit rate will be slower.

**See Also**                ShiftInEx, ShiftOut, ShiftOutEx

# ShiftInEx

**Type** Function returning UnsignedInteger

**Invocation** ShiftInEx(dataPin, clkPin, bitCnt, flags)  
ShiftInEx(dataPin, clkPin, bitCnt, flags, bitTime)

| Parameter | Method | Type  | Description   |
|-----------|--------|-------|---|
| dataPin   | ByVal  | Byte  | The pin used to input data.                                   |
| clkPin    | ByVal  | Byte  | The pin used to output a clocking signal.                     |
| bitCnt    | ByVal  | Byte  | The number of bits to read in (1 to 16).                      |
| flags     | ByVal  | Byte  | Flag bits controlling the operation.                          |
| bitTime   | ByVal  | int16 | The optional duration of each bit in ticks (see description). |

## Discussion

This function can be used to input data from a synchronous serial device like a shift register. The pin specified for input will be made an input but the pin specified for the clock signal must already be an output and be at the desired initial logic level. The `flags` parameter controls how the shifting process is performed as described in the table below.

**Control Flag Definitions**

| Function   | Hex Value | Bit Mask    |
|--|-----------|-------------|
| MSB first  | &H00      | xx xx xx x0 |
| LSB first  | &H01      | xx xx xx x1 |
| Sample the input after the active clock edge     | &H00      | xx xx xx 0x |
| Sample the input before the active clock edge    | &H02      | xx xx xx 1x |
| Fastest possible bit time                        | &H00      | xx xx x0 xx |
| Use the provided <code>bitTime</code> parameter  | &H04      | xx xx x1 xx |
| The active clock edge is the leading clock edge  | &H00      | xx xx 0x xx |
| The active clock edge is the trailing clock edge | &H08      | xx xx 1x xx |

The remaining bits are currently undefined but may be employed in the future.

For each of the number of bits specified, either the state of the data pin will be read and saved first or the clock line will be changed to the opposite state first depending on bit 1 of the `flags` parameter. Finally, the clock line will be returned to the original state thus completing one bit cycle.

If the `flags` parameter so specifies, the `bitTime` parameter value will be used to control the bit rate of the shifting process. The units of the `bitTime` parameter are, by default, approximately 67.8ns. However, `Register.TimerSpeed1` may be changed to adjust the controlling clock speed. If the `bitTime` parameter is not provided or if the value given is zero, the shifting will occur at the maximum rate.

Due to processing overhead the minimum bit time in the controlled speed mode is approximately 4µS. Attempting faster bit times in the controlled speed mode will produce undefined results. Without speed control, the bit time is approximately 2.5µS. Note that the duty cycle of the clock signal will be closer to 50% in the controlled speed mode. Without speed control, the active clock phase can be as little as 20% of the period.

The returned value consists of the data bits read arranged in MSB or LSB order as specified by the `flags` parameter. If MSB order is specified, the first bit read will be in the most significant bit position of the result. If LSB order is specified, the first bit read will be in the least significant bit position. If fewer than 16 bits are read, the remaining bits will be zero.

For reference purposes, the `ShiftIn()` function is roughly equivalent to `ShiftInEx(dpin, cpin, bitCnt, &H04, 74)`. However, the value read will be in the high order 8 bits of the returned value.

## Resource Usage

This subroutine uses the I/O Timer if the `flags` parameter has bit 2 on. If the I/O Timer is already in use, the function returns immediately and the return value is zero. No other use of this resource should be attempted while the shifting is in progress. Interrupts are disabled during the shifting process. However, RTC ticks are accumulated during the shifting process so the RTC should not lose time.

## Timing

Bit 3 of the `flags` parameter specifies the active edge of the clock pulse, i.e. whether the data line will be sampled relative to the leading edge or the trailing edge of the clock pulse. Bit 1 of the `flags` parameter controls whether the sampling will be done before or after the active edge. When bit 1 of the `flags` parameter is zero, the data line will be sampled approximately 2 CPU clock cycles after the active edge of the clock pulse. When bit 1 of the `flags` parameter is one, the data line will be sampled approximately 5 CPU clock cycles before the active edge of the clock pulse. With a 14.7456MHz CPU clock, these intervals are approximately 135nS and 340nS, respectively.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**      `ShiftIn`, `ShiftOut`, `ShiftOutEx`

# ShiftOut

**Type** Subroutine

**Invocation** ShiftOut(dataPin, clkPin, bitCnt, val)

| Parameter | Method | Type | Description                               |
|-----------|--------|------|---|
| dataPin   | ByVal  | Byte | The pin used to output data.              |
| clkPin    | ByVal  | Byte | The pin used to output a clocking signal. |
| bitCnt    | ByVal  | Byte | The number of bits to shift out (1 to 8). |
| val       | ByVal  | Byte | The value to shift out.                   |

## Discussion

This function can be used to output data to a synchronous serial device like a shift register. The pin specified for output will be made an output but the pin specified for the clock signal must already be an output and be at the desired initial logic level.

For each of the number of bits specified, the data pin will be set to the state of the corresponding bit in the `val` parameter beginning with the most significant bit first. Then the clock line will be pulsed by changing its logic level twice.

If a data width of fewer than 8 data bits is specified, the state of the remaining bits in the provided value is of no consequence.

## Resource Usage

This subroutine uses the I/O Timer. If the I/O Timer is already in use, the subroutine returns immediately. No other use of this resource should be attempted while the shifting is in progress. Interrupts are disabled during the shifting process. However, RTC ticks are accumulated during the shifting process so the RTC should not lose time.

## Compatibility

For compatibility with I2C/TWI devices the clock rate is approximately 200kHz with `Register.TimerSpeed1` at its default value of 1. If the value of `Register.TimerSpeed1` is changed, the bit rate will be slower.

**See Also** ShiftIn, ShiftInEx, ShiftOutEx

# ShiftOutEx

**Type** Subroutine

**Invocation** ShiftOutEx(dataPin, clkPin, bitCnt, val, flags)  
ShiftOutEx(dataPin, clkPin, bitCnt, val, flags, bitTime)

| Parameter | Method | Type    | Description   |
|-----------|--------|---------|---|
| dataPin   | ByVal  | Byte    | The pin used to output data.                                  |
| clkPin    | ByVal  | Byte    | The pin used to output a clocking signal.                     |
| bitCnt    | ByVal  | Byte    | The number of bits to shift out (1 to 16).                    |
| val       | ByVal  | int8/16 | The value to shift out.                                       |
| flags     | ByVal  | Byte    | Flag bits controlling the operation.                          |
| bitTime   | ByVal  | int16   | The optional duration of each bit in ticks (see description). |

## Discussion

This function can be used to output data to a synchronous serial device like a shift register. The pin specified for output will be made an output but the pin specified for the clock signal must already be an output and be at the desired initial logic level. The `flags` parameter controls how the shifting process is performed as described in the table below.

| Control Flag Definitions                        |           |             |
|---|-----------|-------------|
| Function  | Hex Value | Bit Mask    |
| MSB first                                       | &H00      | xx xx xx x0 |
| LSB first                                       | &H01      | xx xx xx x1 |
| Fastest possible bit time                       | &H00      | xx xx x0 xx |
| Use the provided <code>bitTime</code> parameter | &H04      | xx xx x1 xx |
| Normal data pin output                          | &H00      | xx xx 0x xx |
| Open drain data pin output                      | &H08      | xx xx 1x xx |

The remaining bits are currently undefined but may be employed in the future. For compatibility, the undefined bits should always be zero.

For each of the number of bits specified, the data pin will be set to the state of the corresponding bit in the `val` parameter beginning with the either the most significant bit first or the least significant bit first depending on bit 0 of the `flags` parameter. Then the clock line will be pulsed by changing its logic level twice.

Note that if a data width of fewer than 16 data bits is specified, the bits to be shifted out must be properly aligned in the value provided. If MSB order is specified, the data bits must be positioned in the most significant bits of the value provided. If LSB order is specified, the data bits must be positioned in the least significant bits of the value provided.

If the `flags` parameter so specifies, the `bitTime` parameter value will be used to control the bit rate of the shifting process. The units of the `bitTime` parameter are, by default, approximately 67.8ns. However, `Register.TimerSpeed1` may be changed to adjust the controlling clock speed. If the `bitTime` parameter is not provided or if the value given is zero, the shifting will occur at the maximum rate.

Due to processing overhead the minimum bit time in the controlled speed mode is approximately 4µS. Attempting faster bit times in the controlled speed mode will produce undefined results. Without speed control, the bit time is approximately 2.2µS. Note that the duty cycle of the clock signal will be closer to 50% in the controlled speed mode. Without speed control, the active clock phase can be as little as 20% of the period.



Normally, the data pin will be driven high or low according to the data bits being shifted out. For compatibility with certain data bus interfaces, the `flags` parameter bit 3 can be used to specify that the data pin should be put in high impedance input mode when outputting a one bit and actively pulled to ground for a zero bit. In this mode, an external pullup resistor will need to be used to obtain a voltage level corresponding to a logic one.

For reference purposes, the `ShiftOut()` routine is roughly equivalent to `ShiftOutEx(dpin, cpin, bitCnt, Shl(CInt(val), 8), &H04, 74)`.

### Resource Usage

This subroutine uses the I/O Timer if the `flags` parameter has bit 2 on. If the I/O Timer is already in use, the subroutine returns immediately. No other use of this resource should be attempted while the shifting is in progress. Interrupts are disabled during the shifting process. However, RTC ticks are accumulated during the shifting process so the RTC should not lose time.

### Compatibility

This function is not available in BasicX compatibility mode.

**See Also**      `ShiftIn`, `ShiftInEx`, `ShiftOut`

# Shl

**Type**                Function returning the same type as the first parameter.

**Invocation**        Shl(val, shiftCnt)

| Parameter | Method | Type     | Description                                  |
|-----------|--------|----------|--|
| val       | ByVal  | integral | The value to be shifted.                     |
| shiftCnt  | ByVal  | int8/16  | The number of bit positions to shift (0-16). |

## Discussion

This function returns the value provided as the first parameter but shifted left the number of bit positions specified by the second parameter. If the `shiftCnt` is zero, the value is returned unchanged. If the `shiftCnt` is greater than or equal to the number of bits in the value provided, the return value will be zero. For signed types, the sign of the result will be the same as that of the provided value.

The type of the return value will be the same as the type of the first parameter.

## Example

```
Dim i as Integer, j as Integer

i = 23
j = Shl(i, 5)      ' result will be 736
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            Shr

# Shr

**Type**                Function returning the same type as the first parameter.

**Invocation**        Shr(val, shiftCnt)

| Parameter | Method | Type     | Description                                  |
|-----------|--------|----------|--|
| val       | ByVal  | integral | The value to be shifted.                     |
| shiftCnt  | ByVal  | int8/16  | The number of bit positions to shift (0-16). |

## Discussion

This function returns the value provided as the first parameter but shifted right the number of bit positions specified by the second parameter. If the `shiftCnt` is zero, the value is returned unchanged. If the `shiftCnt` is greater than or equal to the number of bits in the value provided, the return value will be zero. For signed types, the sign of the result will be the same as that of the provided value.

The type of the return value will be the same as the type of the first parameter.

## Example

```
Dim i as Integer, j as Integer

i = 23
j = Shr(i, 2)            ' result will be 5
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            Shl

# Signum

---

**Type**                Function returning the same type as the first parameter.

**Invocation**        Signum(val)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| val       | ByVal  | signed | The value to be tested for positive, zero, negative. |

## Discussion

This function returns +1, 0 or −1 depending on whether the value provided is positive, zero or negative. The type of the return value will be the same as the type of the parameter value.

## Example

```
Dim i as Integer, j as Integer

i = -23
j = Signum(i)            ' result will be -1
```

## Compatibility

This function is not available in BasicX compatibility mode.

# Sin

---

**Type**            Function returning Single

**Invocation**    Sin(arg)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| arg       | ByVal  | Single | The angle, in radians, of which the sine will be computed. |

## Discussion

The return value will be the sine of the supplied value, in the range  $-1.0$  to  $1.0$ .

## Example

```
Const pi as Single = 3.14159
Dim val as Single

val = Sin(pi / 2.0)       ' result is approximately 1.0
```

**See Also**        Asin, DegToRad, RadToDeg

# SizeOf

**Type**            Function returning Integer

**Invocation**    SizeOf(var)

| Parameter | Method | Type     | Description                                    |
|-----------|--------|----------|--|
| var       | ByRef  | any type | The variable whose size, in bytes, is desired. |

## Discussion

This function returns the number of bytes constituting the supplied variable.

The primary purpose of this function is to allow writing code that is more easily maintained. For example, instead of hard coding the size value to pass to the `OpenQueue()` subroutine, you can use `SizeOf(queue)` instead. When you change the size of the queue there will be no need to update the `OpenQueue()` calls.

When used with arrays, you may give the array name without any index parameters and `SizeOf()` will return the total number of bytes occupied by the array. Alternately, you may specify constant expressions for all of the array dimensions and `SizeOf()` will return the number of bytes occupied by a single element of the array. This function is not particularly useful with sub-byte types (Bit and Nibble).

The `SizeOf()` function also allows the argument to name one of the fundamental data type (except `String`). In this case it returns the number of bytes comprising the type. For example, `Sizeof(Integer)` returns the value 2.

## Example

```
Dim cnt as Integer
Dim val as Single
Dim ia(1 to 20) as Integer

cnt = SizeOf(val)            ' result is 4
cnt = SizeOf(ia)            ' result is 40
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        SizeOfU

# SizeOfU

**Type**                      Function returning UnsignedInteger

**Invocation**            SizeOfU(var)

| Parameter | Method | Type     | Description                                    |
|-----------|--------|----------|--|
| var       | ByRef  | any type | The variable whose size, in bytes, is desired. |

## Discussion

This function returns the number of bytes constituting the supplied variable.

The primary purpose of this function is to allow writing code that is more easily maintained. For example, instead of hard coding the size value to pass to the `OpenQueue()` subroutine, you can use `SizeOfU(queue)` instead. When you change the size of the queue there will be no need to update the `OpenQueue()` calls.

When used with arrays, you may give the array name without any index parameters and `SizeOfU()` will return the total number of bytes occupied by the array. Alternately, you may specify constant expressions for all of the array dimensions and `SizeOfU()` will return the number of bytes occupied by a single element of the array. This function is not particularly useful with sub-byte types (Bit and Nibble).

The `SizeOfU()` function also allows the argument to name one of the fundamental data type (except `String`). In this case it returns the number of bytes comprising the type. For example, `SizeofU(Integer)` returns the value 2.

## Example

```
Dim cnt as UnsignedInteger
Dim val as Single
Dim ia(1 to 20) as Integer

cnt = SizeOfU(val)                ' result is 4
cnt = SizeOfU(ia)                ' result is 40
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**                SizeOf

# Sleep

**Type** Subroutine

**Invocation** Sleep(time)

| Parameter | Method | Type            | Description   |
|-----------|--------|-----------------|---|
| time      | ByVal  | Single or int16 | The amount of time to delay, in seconds (Single) or RTC ticks (int16) |

## Discussion

This routine suspends the current task for a period of time up to as long as specified. The actual delay depends on what other tasks actually do that may run in the interim. It is possible that the task will be suspended indefinitely depending on what another task might do.

Note that if the current task is locked, this call will unlock it.

There is a subtle difference between `Delay()` and `Sleep()` when the arguments are non-zero. For `Delay()` the specified time is the minimum amount of delay that the task will experience assuming that no other task is ready to run. The actual delay could be up to 1.95ms longer than the specified delay. For `Sleep()`, the specified time is the maximum amount of delay that the task will experience assuming that no other task is ready to run. The actual delay could be up to 1.95ms less than the specified delay.

## Example

```
Do
    Call PutPin(25, 0)
    Call Sleep(0.5)      ' a half-second delay
    Call PutPin(25, 1)
    Call Sleep(256)     ' a half-second delay
Loop
```

This loop causes the red LED to turn on and off alternately for a half second each.

## Compatibility

The BasicX documentation specifically indicates that `Sleep()` will unlock a locked task. However, tests indicate that this only happens if the parameter to `Sleep()` is non-zero. This implementation unlocks a task on any `Sleep()` call.

**See Also** Delay, DelayUntilClockTick, Pause, WaitForInterval, Register.RTCStopWatch



# SngClass

**Type**            Function returning Byte

**Invocation**    SngClass(arg)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| arg       | ByVal  | Single | The value of which to determine the floating point classification. |

## Discussion

The IEEE 754 standard floating point format used by ZBasic specifies a set of classifications for floating point values. This function returns a numeric value indicating the class to which the passed `Single` value belongs. The table below enumerates the return values and describes the meaning of each.

| Floating Point Value Classes |       |   |
|------------------------------|-------|---|
| Class                        | Value | Description   |
| ClassNormal                  | 1     | Normalized - This class represents "normal" floating point values such as 1.537 but does not include 0.0.   |
| ClassZero                    | 2     | Zero - This class represents the zero value (positive and negative).  |
| ClassInfinity                | 3     | Infinity - This class represents positive and negative infinity. Dividing a positive value by zero results in positive infinity.  |
| ClassDenormal                | 4     | Denormalized - This class represents an internal form known as denormalized values. Such values should never be generated as a result of a floating point operation. However if you copy some random bytes into a floating point variable the result may be a denormalized value. |
| ClassNaN                     | 5     | NaN - This class represents values that are "Not A Number". Taking the square root of a negative value or the logarithm of zero results in a NaN.   |

The names in the first column are available as built-in constants. Except for `ClassNaN`, the return value may include the flag `&H80` to indicate a negative value. For example, `SngClass(-1.0)` returns the value `&H81` to indicate a negative `ClassNormal` value. The built-in constant representing the negative flag is `ClassNegative`. The built-in constant `ClassMask` may be used to remove the negative flag from the return value, e.g. `SngClass(fval) And ClassMask`.

## Examples

```
Dim class as Byte
```

```
class = SngClass(1.0)           ' result is 1
class = SngClass(-1.0)          ' result is &H81
class = SngClass(-1.0) And ClassMask ' result is 1
class = SngClass(Sqr(-1.0))      ' result is 5
class = SngClass(1.0 / 0.0)      ' result is 3
```

## Compatibility

This function is not available in BasicX compatibility mode.

# SPICmd

**Type** Subroutine

**Invocation** SPICmd(channel, writeCnt, writeData, readCnt, readData)

| Parameter | Method | Type     | Description                                   |
|-----------|--------|----------|---|
| channel   | ByVal  | Byte     | The SPI channel number (1-4).                 |
| writeCnt  | ByVal  | integral | The number of bytes to write (0 – 65535).     |
| writeData | ByRef  | any type | The variable containing the data to write.    |
| readCnt   | ByVal  | integral | The number of bytes to read (0 – 65535).      |
| readData  | ByRef  | any type | The variable in which to place the data read. |

## Discussion

The routine allows you to send and/or receive data from a device connected to the processor's SPI bus (the holes on the end of the ZX device between pins 1 and 24). The specified channel must have been previously opened with a call to `OpenSPI()`. If the channel has not been opened, the results are undefined.

If both `writeCnt` and `readCnt` are zero the routine returns immediately without doing anything. You may specify the value 0 for `writeData` or `readData` if no data is being provided. If the value of `readCnt` exceeds the size of the `readData` variable, the additional bytes will be written to subsequent memory locations, possibly with undesirable results.

The execution of the SPI command occurs in four phases:

- Chip select is asserted by setting the previously specified pin to a logic zero level.
- If the `writeCnt` parameter is non-zero, the data bytes at `writeData` are written sequentially to the SPI interface. The data returned by the SPI device during this phase is discarded.
- If the `readCnt` parameter is non-zero, the existing data beginning at `readData` are written to the SPI device and the returned bytes are stored sequentially in the specified variable. That is, the byte at `readData(1)` is sent to the device and the byte that it sends back is stored at `readData(1)`. The same occurs for `readData(2)`, etc.
- Finally, chip select is deasserted by setting the previously specified pin to a logic one level.

Whether you use `writeData` or `readData` or both depends on the particulars of the device you're using. In some cases, you'll need to populate `readData` and in other cases not. Careful study of the datasheet of the target device will be required to determine how `SPICmd()` can be used to interface with it.

For an SPI channel that is opened with a non-zero `rxDelay` parameter specified (see `OpenSPI()`), a delay is implemented prior to each SPI cycle for which the data read is placed in the `readData` buffer, i.e., the third phase described above. The delay value specified is interpreted as the number of cycles of the SPI clock frequency (but ignoring the SPI2X configuration bit). Of course, during the delay time the SPI clock signal (SCK) will be idle. This delay is useful when communicating with slave devices that must compute data values to return, for example, a ZX-24n operating in SPI slave mode.

## Example

```
Dim odata(1 to 2) as Byte, idata(1 to 10) as Byte
Call OpenSPI (1, 0, 12)
Odata(1) = &H06
Odata(2) = &H00
Call SPICmd(1, 2, odata(1), 10, idata(1))
```

In this example `idata` is not initialized before calling `SPICmd()`. If your SPI device needs specific data written to it during the read phase `idata` would need to be initialized before the call.

### Compatibility

The use of a zero value to indicate that no data buffer is being supplied is not supported in BasicX compatibility mode. Also, in BasicX compatibility mode, both `writeCnt` and `readCnt` are Byte values and, thus, limited to a maximum of 255.

**See Also**      `CloseSPI`, `OpenSPI`

# Sqr

---

**Type**                Function returning Single

**Invocation**        Sqr(arg)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| arg       | ByVal  | Single | The value of which the square root will be computed. |

## Discussion

The return value will be the square root of the supplied value. Note that the `Sqr( )` function will return NaN if the argument is negative.

## Example

```
Dim val as Single  
  
val = Sqr(2.0)        ' result is approximately 1.414
```

# StackCheck

---

**Type** Subroutine

**Invocation** StackCheck(enable)

| Parameter | Method | Type    | Description                       |
|-----------|--------|---------|-----------------------------------|
| enable    | ByVal  | Boolean | The enable/disable state desired. |

## Discussion

This subroutine enables or disables stack checking. See the section on Run Time Stack Checking in the ZBasic Reference Manual for more information.

## Example

```
Call StackCheck(true)
```

## Compatibility

This routine is not available in BasicX compatibility mode nor is it available for native mode devices.

# StatusCom

---

**Type**                Function returning Byte

**Invocation**        StatusCom(chan)

| Parameter | Method | Type | Description                     |
|-----------|--------|------|---------------------------------|
| chan      | ByVal  | Byte | The serial channel of interest. |

## Discussion

This function returns a set of flag bits that indicate the status of the specified serial channel. The bits and their meanings are shown in the table below.

| Serial Channel Status Bit Values |   |
|----------------------------------|---|
| Value                            | Meaning   |
| &H01                             | The channel number is valid but may or may not be open. |
| &H02                             | The channel is open.                                    |
| &H04                             | The channel has data yet to be transmitted.             |

The remaining bits are currently undefined but may convey additional information in the future.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            CloseCom, ComChannels, DefineCom, OpenCom

# StatusQueue

---

**Type**            Function returning Boolean

**Invocation**    StatusQueue(queue)

| Parameter | Method | Type          | Description            |
|-----------|--------|---------------|------------------------|
| queue     | ByRef  | array of Byte | The queue of interest. |

## Discussion

This function returns `True` if there data bytes in the queue, otherwise `False`.

Note that before any queue operations are performed, the queue data structure must be initialized. See the discussion of `OpenQueue ( )` for more details.

## Compatibility

BasicX allows any type for the first parameter. This implementation requires that it be an array of `Byte`.

**See Also**            `GetQueueCount`, `OpenQueue`

# StatusTask

**Type** Function returning Byte

**Invocation** StatusTask(taskStack)  
StatusTask()

| Parameter | Method | Type          | Description                       |
|-----------|--------|---------------|-----------------------------------|
| taskStack | ByRef  | array of Byte | The stack for a task of interest. |

## Discussion

This function returns a value indicating the status of the task associated with the given task stack. If no task stack is explicitly given, the task stack for the `Main()` routine is assumed. The return values and their respective meanings are shown in the table below.

| Task Status Values    |       |   |
|-----------------------|-------|---|
| Constant              | Value | Meaning   |
| TaskReady             | 0     | The task is running or ready to run.                    |
| TaskSleeping          | 1     | The task is sleeping.                                   |
| TaskWaitInputCapture  | 2     | The task is waiting for InputCapture() to complete.     |
| TaskWaitInt0          | 3     | The task is awaiting Interrupt 0.                       |
| TaskWaitInt1          | 4     | The task is awaiting Interrupt 1.                       |
| TaskWaitInt2          | 5     | The task is awaiting Interrupt 2.                       |
| TaskWaitInterval      | 6     | The task is waiting for the interval counter to expire. |
| TaskWaitAnalogCompare | 7     | The task is waiting for an analog comparator event.     |
| TaskWaitPinChange0    | 8     | The task is waiting for a pin change event 0            |
| TaskWaitPinChange1    | 9     | The task is waiting for a pin change event 1            |
| TaskWaitPinChange2    | 10    | The task is waiting for a pin change event 2            |
| TaskWaitPinChange3    | 11    | The task is waiting for a pin change event 3            |
| TaskWaitOutputCapture | 12    | The task is waiting for OutputCapture() to complete.    |
| TaskWaitInt3          | 13    | The task is awaiting Interrupt 3.                       |
| TaskWaitInt4          | 14    | The task is awaiting Interrupt 4.                       |
| TaskWaitInt5          | 15    | The task is awaiting Interrupt 5.                       |
| TaskWaitInt6          | 16    | The task is awaiting Interrupt 6.                       |
| TaskWaitInt7          | 17    | The task is awaiting Interrupt 7.                       |
| TaskHalting           | 254   | The task is in the process of terminating.              |
| TaskHalted            | 255   | The task has terminated.                                |

| Pin Change Event Ports |            |            |            |            |
|------------------------|------------|------------|------------|------------|
| Underlying CPU         | PinChange0 | PinChange1 | PinChange2 | PinChange3 |
| mega644P               | PortA      | PortB      | PortC      | PortD      |
| mega1281               | PortB      | PortE      |            |            |
| mega1280               | PortB      | PortJ      | PortK      |            |

If this function is invoked using an array other than one that is or was being used for a task stack the result is undefined. See the section on Task Management in the ZBasic Reference Manual for additional information regarding task management.

**See Also** ExitTask, ResumeTask, RunTask, TaskIsValid



# StatusX10

**Type**            Function returning Byte

**Invocation**    StatusX10(chan)

| Parameter | Method | Type | Description                                 |
|-----------|--------|------|---|
| chan      | ByVal  | Byte | The X-10 communication channel of interest. |

## Discussion

This function returns a set of flag bits that indicate the status of the specified X-10 channel. The bits and their meanings are shown in the table below. The return value may comprise zero or more of the status bits.

| X-10 Channel Status Bit Values |   |
|--------------------------------|---|
| Value                          | Meaning   |
| &H01                           | The channel number is valid but may or may not be open. |
| &H02                           | The channel is open.                                    |
| &H04                           | The channel has data yet to be transmitted.             |

The remaining bits are currently undefined but may convey additional information in the future.

## Compatibility

This function is not available on ZX models that are based on the ATmega32 processor (e.g. the ZX-24). Moreover, it is not available in BasicX compatibility mode.

**See Also**        CloseX10, DefineX10, OpenX10

# StrAddress

**Type**                      Function returning UnsignedInteger

**Invocation**            StrAddress(str)

| Parameter | Method | Type   | Description  |
|-----------|--------|--------|--|
| str       | ByVal  | String | The string variable whose string address is desired. |

## Discussion

This function returns the memory address of the first character of a string stored in a string variable. Note that for dynamically allocated strings, the string address will be zero if the string is empty and the returned address may refer to RAM, Program Memory or Persistent memory. The function `StrType()` can be used to determine which address space contains the string's characters. For statically allocated strings, the string address will always be non-zero even if the string is empty.

See the section on Strings in the ZBasic Reference Manual for more details about dynamically vs. statically allocated strings.

## Example

```
Dim str as String
Dim addr as UnsignedInteger
Dim b as Byte

str = "Hello, world!"
addr = StrAddress(str)
b = RamPeek(addr)                      ' result will be 72, the letter H
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**                      StrType

# StrCompare

---

**Type**                Function returning Integer

**Invocation**        StrCompare(str1, str2)  
                         StrCompare(str1, str2, ignoreCase)

| Parameter  | Method | Type    | Description  |
|------------|--------|---------|--|
| str1       | ByVal  | String  | The first string to compare.                               |
| str2       | ByVal  | String  | The second string to compare.                              |
| ignoreCase | ByVal  | Boolean | A flag controlling whether alphabetic case is significant. |

## Discussion

This function returns a value indicating the “sort order” of the two strings. If the returned value is negative, the first string precedes the second in sort order, i.e. the first string would appear before the second in a list sorted alphabetically. If the returned value is zero, the strings have the same sort order and if it is greater than zero, the second string has a higher sort order. If the optional `ignoreCase` parameter is given, the comparison is done either observing or ignoring differences in alphabetic case depending on the value of the parameter. For the purposes of this parameter only the characters A-Z and a-z (&H41 to &H5a and &H61 to &H7a) are considered to be alphabetic. If the `ignoreCase` parameter is omitted, the comparison is performed observing case differences.

## Example

```
Dim str1 as String
Dim str2 as String

If (StrCompare(str1, str2, true) = 0) Then
    Debug.Print "The strings match"
End If
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            StrFind

# StrFind

**Type**            Function returning Byte

**Invocation**    StrFind(inStr, findStr)  
                  StrFind(inStr, findStr, startIdx)  
                  StrFind(inStr, findStr, startIdx, ignoreCase)

| Parameter  | Method | Type     | Description  |
|------------|--------|----------|--|
| inStr      | ByVal  | String   | The string to be searched.                                 |
| findStr    | ByVal  | String   | The string being sought.                                   |
| startIdx   | ByVal  | integral | The index of inStr at which to begin the search.           |
| ignoreCase | ByVal  | Boolean  | A flag controlling whether alphabetic case is significant. |

## Discussion

This function attempts to find the first occurrence of the `findStr` string within the `inStr` string. If it is found, the return value gives the 1-based index where the sought string was found within the searched string. If the sought string is not found, zero is returned. If the optional `startIdx` parameter is not given, the search begins at the first character of the searched string, equivalent to specifying 1 for `startIdx`. If the optional `ignoreCase` parameter is not given, the search is performed observing alphabetic case differences, otherwise alphabetic case differences are significant or not depending on the value specified for `ignoreCase`. For the purposes of this parameter only the characters A-Z and a-z (&H41 to &H5a and &H61 to &H7a) are considered to be alphabetic.

Searching for a zero length string will always be successful and the return value will be the specified or implied starting index. Searching for a non-zero length string within a zero length string will always fail, returning 0.

## Examples

```
Dim idx as Byte
```

```
idx = StrFind("haystack", "needle")           ' returns 0
idx = StrFind("haystack with needle", "needle") ' returns 15
idx = StrFind("foo bar foo", "foo", 2)         ' returns 9
idx = StrFind("foo bar foo", "", 2)            ' returns 2
idx = StrFind("foo bar FOO", "FOO")            ' returns 9
idx = StrFind("foo bar FOO", "FOO", 1, true)   ' returns 1
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**        StrCompare

# StrReplace

**Type**                      Function returning String

**Invocation**            StrReplace(str, findStr, replStr)  
                             StrReplace(str, findStr, replStr, startIdx)  
                             StrReplace(str, findStr, replStr, startIdx, replCount)  
                             StrReplace(str, findStr, replStr, startIdx, replCount, ignoreCase)

| Parameter  | Method | Type     | Description  |
|------------|--------|----------|--|
| str        | ByVal  | String   | The subject string in which to perform replacement.        |
| findStr    | ByVal  | String   | The sought string.   |
| replStr    | ByVal  | String   | The replacement string.                                    |
| startIdx   | ByVal  | integral | The index of 'str' at which to begin the replacement.      |
| replCount  | ByVal  | integral | The number of replacements to perform.                     |
| ignoreCase | ByVal  | Boolean  | A flag controlling whether alphabetic case is significant. |

## Discussion

This routine produces a new string by replacing occurrences of the sought string with the replacement string in the subject string. If the optional `startIdx` parameter is not given, the search begins at the first character of the subject string, equivalent to specifying 1 for `startIdx`. If the optional `replCount` parameter is not given, all occurrences of the sought string will be replaced. If the optional `ignoreCase` parameter is not given, the search is performed observing alphabetic case differences, otherwise alphabetic case differences are significant or not depending on the value specified for `ignoreCase`. For the purposes of this parameter, only the characters A-Z and a-z (&H41 to &H5a and &H61 to &H7a) are considered to be alphabetic.

If the subject string contains no occurrences of the sought string, or if the sought string is zero length, or if the replacement count is zero, the returned string will be identical to the subject string. The replacement count and the start index are treated internally as signed 16-bit values. If the value of the start index is less than 1, a starting index of 1 is assumed.

## Compatibility

This function is not available in BasicX compatibility mode.

# StrType

**Type** Function returning Byte

**Invocation** StrType(str)

| Parameter | Method | Type   | Description                                       |
|-----------|--------|--------|---|
| str       | ByVal  | String | The string variable whose string type is desired. |

## Discussion

This function returns a value indicating the nature of a string variable. The values returned have the meaning shown in the table below.

| Type | Meaning   |
|------|---|
| &H00 | The string is a standard statically allocated string or a bounded string. The value returned by StrAddress() is a RAM address and can be read using RamPeek() or MemCopy().   |
| &He0 | The string is dynamically allocated. The value returned by StrAddress() is a RAM address (which may be zero) and can be read using RamPeek() or MemCopy().  |
| &He2 | The string is in Program Memory. The value returned by StrAddress() is a Program Memory address and can be read using GetEEPROM().  |
| &He3 | The string is in Persistent Memory. The value returned by StrAddress() is a Persistent Memory address and can be read using GetPersistent().  |
| &He4 | The string is in RAM. The value returned by StrAddress() is a RAM address (which may be zero) and can be read using RamPeek() or MemCopy().   |
| &He5 | The string is in RAM and is limited to 1 or 2 characters. The value returned by StrAddress() is a RAM address and can be read using RamPeek() or MemCopy().   |
| &He6 | The string is in RAM. The value returned by StrAddress() is a RAM address and can be read using RamPeek() or MemCopy(). This special string type is used for native-mode code to pass a bounded string or fixed-length string to a subroutine/function ByVal. |
| &Hff | The string is a statically allocated fixed-length string. The value returned by StrAddress() is a RAM address and the data can be read using RamPeek() or MemCopy().  |

See the section on strings in the ZBasic Reference Manual for more details about dynamically vs. statically allocated strings.

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also** StrAddress

# System.Alloc

---

**Type**                Function returning UnsignedInteger

**Invocation**        System.Alloc(numBytes)

| Parameter | Method | Type     | Description                           |
|-----------|--------|----------|---------------------------------------|
| numBytes  | ByVal  | integral | The size of the requested allocation. |

## Discussion

This function allocates a block of memory from the heap of the specified size and returns the address of the first byte of the block. If a block of the specified size cannot be allocated, zero is returned. The block can be returned to the heap using the subroutine `System.Free()`.

This function and the block of memory it returns must be used with great care. If your program fails to deallocate the block using `System.Free()` when it is no longer needed, the heap may eventually be exhausted. Since space for strings is also allocated from the heap, exhaustion may cause string operations to fail. Moreover, if your program writes to memory outside of the bounds of the block, the heap data structures may be corrupted. This may cause future heap allocation requests to fail.

For native mode devices (e.g. the ZX-24n) a heap allocation may fail if the heap size is set too small compared to the needs of your application. To aid in determining a sufficient heap size the System Library function `System.HeapHeadRoom()` may be used to discover the amount of space in the heap that has not yet been used at the time of the call.

## Example

```
Dim addr as UnsignedInteger
addr = System.Alloc(50)
[other code here that uses the allocated block]
Call System.Free(addr)
addr = 0
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            System.Free

# System.DeviceID

---

**Type** Subroutine

**Invocation** System.DeviceID(buffer)

| Parameter | Method | Type          | Description   |
|-----------|--------|---------------|---|
| buffer    | ByRef  | array of Byte | The array to which the identification characters will be written. |

## Discussion

A call to this routine will copy up to 10 bytes to the buffer provided. The data copied to the buffer comprise characters of a string that identify the ZX device on which the program is executing. The last byte of the identification is followed by a zero byte that serves to mark the end of the identification characters. The example below illustrates how the data can be used to create a string.

Although this subroutine is primarily intended for manufacturing test purposes, it may be useful for other purposes as well.

## Caution

If the array provided is less than 10 bytes long, subsequent memory may be overwritten, possibly with detrimental results.

## Example

```
Dim buf(1 to 10) as Byte
Dim idStr as String
Dim idx as Byte

Call System.DeviceID(buf)
idStr = MakeString(buf.DataAddress, SizeOf(buf))
idx = StrFind(idStr, Chr(0))
If (idx <> 0) Then
    idStr = Left(idStr, idx - 1)
End If
Debug.Print idStr ' Displays "ZX24" on a ZX-24
```

## Compatibility

This routine is not available in BasicX compatibility mode.



# System.Free

---

**Type** Subroutine

**Invocation** System.Free(addr)

| Parameter | Method | Type            | Description                       |
|-----------|--------|-----------------|-----------------------------------|
| addr      | ByVal  | UnsignedInteger | The address of the block to free. |

## Discussion

This subroutine returns a block of allocated memory to the heap so that it may be later re-used. The `addr` parameter must be the value returned by an earlier call to `System.Alloc()` that has not yet been freed. Invoking this subroutine with `addr` equal to 0 is a special case that is benign.

This function and its companion, `System.Alloc()`, must be used with great care. If `System.Free()` is called with a non-zero value that is not one returned by `System.Alloc()` or a value that has already been freed, the heap management data structures will almost certainly be corrupted and future allocations will likely fail. It is a good practice to set an address to zero after it has been freed as illustrated in the example below.

## Example

```
Dim addr as UnsignedInteger
addr = System.Alloc(50)
[other code here that uses the allocated block]
Call System.Free(addr)
addr = 0
```

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also** System.Alloc

# System.HeapHeadRoom

---

**Type**            Function returning UnsignedInteger

**Invocation**    System.HeapHeadRoom()

## Discussion

This function determines the amount of space in the string heap that has never been used irrespective of the current end-of-heap position. The primary use for it is to determine the amount of heap space used by an application in order to balance the requirements of the heap and the various task stacks.

## Compatibility

This function is only available for native code targets, e.g. the ZX-24n.

**See Also**        System.TaskHeadRoom

# System.HeapSize

---

**Type**            Function returning UnsignedInteger

**Invocation**    System.HeapSize()

## Discussion

This function determines the amount of space reserved for the string heap. This value may be of use in special circumstances such as allocating extra buffers or dynamic task stacks.

**See Also**        System.HeapHeadRoom

# System.TaskHeadRoom

---

**Type**            Function returning UnsignedInteger

**Invocation**     System.TaskHeadRoom(taskStack)  
                     System.TaskHeadRoom()

| Parameter | Method | Type          | Description                       |
|-----------|--------|---------------|-----------------------------------|
| taskStack | ByRef  | array of Byte | The stack for a task of interest. |

## Discussion

This function determines the amount of space in task stack of the specified task that has never been used, irrespective of the current position of the task stack pointer. The primary use for it is to determine the amount of task stack space used by a task in order to balance the requirements of the heap and the various task stacks. If the supplied parameter does not refer to a valid task stack (i.e. a stack for a task that is in the task list), the return value will be &Hffff.

For the second form, with no task stack specified, the stack of the calling task is examined. In either case, if zero is returned it is nearly certain that the task stack has overflowed, possibly overwriting adjacent data.

## Compatibility

For VM mode devices, calling this function for the Main() task will always return &HFFFF unless you have specified a stack limit for Main(). See the `Option StackLimit` directive and related directives for more information.

**See Also**        System.HeapHeadRoom

# Tan

---

**Type**            Function returning Single

**Invocation**    Tan(arg)

| Parameter | Method | Type   | Description   |
|-----------|--------|--------|---|
| arg       | ByVal  | Single | The angle, in radians, of which the tangent will be computed. |

## Discussion

The return value will be the tangent of the supplied value. Note that the `Tan( )` function may return positive or negative infinity values.

## Example

```
Const pi as Single = 3.14159
Dim val as Single

val = Tan(pi / 4.0)            ' result is approximately 1.0
```

**See Also**            Atn, Atn2, DegToRad, RadToDeg

# TaskIsLocked

---

**Type**            Function returning Boolean

**Invocation**    TaskIsLocked()

## Discussion

This function will return `True` if the calling task is locked, `False` otherwise.

**See Also**        LockTask, UnlockTask

# TaskIsValid

---

**Type**                Function returning Boolean

**Invocation**        TaskIsValid(taskStack)

| Parameter | Method | Type          | Description                       |
|-----------|--------|---------------|-----------------------------------|
| taskStack | ByRef  | array of Byte | The stack for a task of interest. |

## Discussion

This function will return `True` if the specified task stack is currently in the task list, `False` otherwise. This function can be used with allocated task stacks to determine when it is safe to deallocate the task stack memory.

**See Also**            StatusTask

# Timer

---

**Type**                Function returning Single

**Invocation**        Timer()

## Discussion

This function returns the current RTC time represented as the number of seconds since midnight with a best-case resolution of 1.95ms. Note that `Register.RTCTick` gives you the equivalent information albeit in the form of a 32-bit value representing the number of 1.95ms ticks since midnight. Depending on your needs, one or the other may be more efficient to use.



# To<enum>

**Type**            Function returning an Enum member

**Invocation**    To<enum>(val)

| Parameter | Method | Type     | Description                             |
|-----------|--------|----------|---|
| val       | ByVal  | integral | The value to convert to an Enum member. |

## Discussion

This page describes a set of functions that convert the given value to a member of a specific enumeration. For each enumeration that you define in your program the compiler automatically provides a conversion function whose name is the name of the enumeration with the prefix *To*.

To use this conversion function, replace the <enum> portion of the function name as shown above with the actual enumeration name for which value-to-member conversion is desired. See below for an example of how this is done.

See the section on enumerations for more information.

## Compatibility

This function is provided for backward compatibility. It is recommended to use `CType( )` for new applications.

## Example

```
Enum Color
    Red
    Green
    Blue
End Enum

Dim c as Color

c = ToColor(1)    ' c will have the value Green
```

**See Also**        CType

# ToggleBits

---

**Type** Subroutine

**Invocation** ToggleBits(target, mask)

| Parameter | Method | Type | Description                               |
|-----------|--------|------|---|
| target    | ByRef  | Byte | The byte to be modified.                  |
| mask      | ByVal  | Byte | The mask indicating which bits to modify. |

## Discussion

This subroutine allows you to change the state of one or more bits in a byte while leaving others unchanged. Effectively, the result is the same as using the statement below.

```
target = target Xor mask
```

The `mask` parameter governs which bits will get changed. For each bit of the `mask` parameter that is a 1, the corresponding bit of the `target` will be set to the opposite of its current state. Bits of the `target` that correspond to zero bits of the `mask` parameter will remain unchanged.

The advantage to using the `ToggleBits()` subroutine instead of the equivalent statement is twofold. Firstly, it is more efficient, resulting in less code and faster execution time. Secondly, and perhaps more importantly, it performs the action as an atomic operation, i.e. one that is guaranteed, once begun, to complete without an intervening task switch. This characteristic makes `ToggleBits()` useful for modifying I/O ports and other `Byte` values in a multi-tasking environment.

## Example

```
' change the state of the two least significant bits of Port C
Call ToggleBits(Register.PortC, &H03)
```

## Compatibility

This routine is not available in BasicX compatibility mode. Also, it is only supported by ZX firmware later than v1.0.0.

**See Also** SetBits

# Trim

---

**Type**            Function returning String

**Invocation**    Trim(str)

| Parameter | Method | Type   | Description                                    |
|-----------|--------|--------|--|
| str       | ByVal  | String | The string from which blanks will be stripped. |

## Discussion

This function returns a new string containing the same characters as the passed string except that leading and trailing spaces will be removed. If the string consists solely of spaces, the resulting string will be zero length.

## Example

```
Dim s as String, s1 as String
s = "  Hello, world!  "
s2 = Trim(s)                       ' the result will be "Hello, world!"
```

**See Also**        Left, Mid, Right

# UBound

**Type**                Function returning Integer

**Invocation**        UBound(array) or  
                         UBound(array, dimension)

| Parameter | Method | Type      | Description   |
|-----------|--------|-----------|---|
| array     | ByRef  | any array | The array about which the bound information is desired.         |
| dimension | ByVal  | int16     | The dimension of interest. See the discussion for more details. |

## Discussion

This function returns the upper bound of the specified array. There are two forms. The first requires only the array to be specified. In this case, the upper bound of the first dimension of the array is returned. The second form specifies a dimension number, the valid range of which is 1 to the number of dimensions of the array. The array may be located in RAM, Program Memory or Persistent Memory.

In contrast to `LBound( )`, a parameter that is an array cannot be passed to `UBound( )` since the return value of `UBound( )` is computed at compile-time and many different sized arrays may be passed as a parameter.

Note that the use of this function instead of hard-coding values makes your code easier to maintain.

## Example

```
Dim ba(1 to 20) as Byte
Dim ma(3 to 5, -6 to 7) as Byte
Dim i as Integer

i = UBound(ba)           ' the result is 20
i = UBound(ma)           ' the result is 5
i = UBound(ma, 1)        ' the result is 5
i = UBound(ma, 2)        ' the result is 7
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also**            LBound

# UCase

---

**Type**            Function returning String

**Invocation**    UCase(str)

| Parameter | Method | Type   | Description                             |
|-----------|--------|--------|---|
| str       | ByVal  | String | The string to be changed to upper case. |

## Discussion

This function returns a new string containing the same characters as the passed string except that all lower case characters will be replaced with upper case characters.

## Example

```
Dim s as String, s1 as String
s = "Hello, world!"
s2 = UCase(s)                    ' the result will be "HELLO, WORLD!"
```

**See Also**        LCase

# UnlockTask

---

**Type**            Subroutine

**Invocation**    UnlockTask()

## Discussion

This routine causes the running task to become unlocked so that other tasks can run. Calling `UnlockTask()` when a task is not actually locked has no effect.

**See Also**        LockTask

# UpdateRTC

**Type** Subroutine

**Invocation** UpdateRTC(fastTicks)

| Parameter | Method | Type  | Description                                 |
|-----------|--------|-------|---|
| fastTicks | ByVal  | int16 | The number of fast ticks to add to the RTC. |

## Discussion

This subroutine can be used to update the RTC with the number of fast ticks missed during a long operation performed with interrupts disabled. In order to determine the number of fast ticks that are missed, your code must periodically check the interrupt flag of the RTC timer and, if it is set, increment a local counter value and then reset the interrupt flag.

## Example

```
' This example is for ZX devices that use Timer0 for the RTC timer.
Atomic
    Dim missedTicks as UnsignedInteger
    Const TickFlag as Byte = &H02
    missedTicks = 0
    Do
        ' place code here that performs one iteration of a
        ' long process and eventually exits the loop

        ' check the RTC flag, reset it
        If (CBool(Register.TIFR0 And TickFlag)) Then
            missedTicks = missedTicks + 1
            Register.TIFR0 = TickFlag
        End If
    Loop
    Call UpdateRTC(missedTicks)
    Call Yield()
End Atomic
```

**See Also** Yield

# ValueI

**Type** Subroutine

**Invocation** ValueI(str, val, flag)

| Parameter | Method | Type    | Description  |
|-----------|--------|---------|--|
| str       | ByVal  | String  | The string from which to extract an Integer value. |
| val       | ByRef  | int16   | The variable to receive the value.                 |
| flag      | ByRef  | Boolean | The variable to receive a success indicator.       |

## Discussion

This routine converts a character representation of an integral number, contained in the `str` parameter, to an `Integer` value returned in the `val` parameter. If the string is in an acceptable format, the `flag` parameter is set to `True`. Otherwise, the `flag` parameter is set to `False` and the `val` parameter will be 0.

The string may contain any number of leading and/or trailing spaces. The value itself may consist of an optional plus or minus sign, an optional radix indicator, and one or more digits. The supported radix indicators are `&H` for hexadecimal, `&O` for octal and `&B` or `&X` for binary (all case insensitive). If no radix indicator is present, decimal is assumed.

If the provided string has the proper format but represents a value that is too large or too small to be represented as an `Integer`, the result will be invalid but no such indication will be given.

Examples of integral values accepted by `ValueI()` are:

```
103
+123
&H55
-&B01101
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also** ValueL, ValueS



# ValueL

**Type** Subroutine

**Invocation** ValueL(str, val, flag)

| Parameter | Method | Type    | Description                                     |
|-----------|--------|---------|---|
| Str       | ByVal  | String  | The string from which to extract an Long value. |
| Val       | ByRef  | int32   | The variable to receive the value.              |
| Flag      | ByRef  | Boolean | The variable to receive a success indicator.    |

## Discussion

This routine converts a character representation of an integral number, contained in the `str` parameter, to a Long value returned in the `val` parameter. If the string is in an acceptable format, the `flag` parameter is set to `True`. Otherwise, the `flag` parameter is set to `False` and the `val` parameter will be 0.

The string may contain any number of leading and/or trailing spaces. The value itself may consist of an optional plus or minus sign, an optional radix indicator, and one or more digits. The supported radix indicators are &H for hexadecimal, &O for octal and &B or &X for binary (all case insensitive). If no radix indicator is present, decimal is assumed.

If the provided string has the proper format but represents a value that is too large or too small to be represented as a Long, the result will be invalid but no such indication will be given.

Examples of integral values accepted by `ValueL()` are:

```
103
+123
&H55
-&B01101
```

## Compatibility

This function is not available in BasicX compatibility mode.

**See Also** ValueI, ValueS

# ValueS

**Type** Subroutine

**Invocation** ValueS(str, val, flag)

| Parameter | Method | Type    | Description  |
|-----------|--------|---------|--|
| str       | ByVal  | String  | The string from which to extract a floating point value. |
| val       | ByRef  | Single  | The variable to receive the value.                       |
| flag      | ByRef  | Boolean | The variable to receive a success indicator.             |

## Discussion

This routine converts a character representation of a floating pointer number, contained in the `str` parameter, to a `Single` value returned in the `val` parameter. If the string is in an acceptable format, the `flag` parameter is set to `True`. Otherwise, the `flag` parameter is set to `False` and the `val` parameter will be 0.0.

The string may contain any number of leading and/or trailing spaces. The value itself may consist solely of decimal digits or may have a leading plus or minus sign. The value may include a decimal point, with or without preceding digits. However, there must be a digit either preceding the decimal point or following it, or both. Optionally, there may be a multiplier value consisting of the letter E (upper or lower case), optionally followed by a plus or minus sign, followed by one or more digits. Note that the range of acceptable input is wider than that for real values in ZBasic statements.

If the provided string has the proper format but represents a value that is too large or too small to be represented as a `Single`, the result will be invalid but no such indication will be given.

Examples of floating point numbers accepted by `ValueS()` are:

```
.30103
3.14159
-200.
1e05
+6.02E+23
123
```

**See Also** ValueI, ValueL

# VarPtr

**Type**            Function returning `UnsignedInteger`

**Invocation**    `VarPtr(var)`

| Parameter | Method | Type         | Description                                   |
|-----------|--------|--------------|---|
| var       | ByRef  | any variable | The variable of which the address is desired. |

## Discussion

This function returns the `UnsignedInteger` representation of the RAM address of the specified variable. Note that for arrays, you may also specify subscript expressions for all of the array dimensions to yield the address of an individual array element. Without the subscript expressions, the resulting value will be the address of the first element of the array.

This function is useful for deriving the address to pass to the several functions that require a RAM address, e.g. `BitCopy()`, `RamPeek()`, `RamPoke()`, etc.

This function is identical to `MemAddressU()` and is provided for BasicX compatibility.

**See Also**        `MemAddress`, `MemAddressU`

# WaitForInterrupt

**Type** Subroutine

**Invocation** WaitForInterrupt(mode)  
WaitForInterrupt(mode, intNum)

| Parameter | Method | Type | Description  |
|-----------|--------|------|--|
| mode      | ByVal  | Byte | A value specifying what action will trigger the interrupt. See the discussion below. |
| intNum    | ByVal  | Byte | A designator for the interrupt to await (see discussion below).                      |

## Discussion

This routine allows a task to suspend itself and wait for an interrupt. The particular interrupt awaited depends on the `intNum` designator combined with the `mode` value. There are three general sources of interrupts that can be awaited: external interrupts, analog comparator interrupts and pin change interrupts.

## External Interrupts 0-7

A task may await an external interrupt by specifying the value 0 through 7 (corresponding to external interrupts 0-7, respectively) for the `intNum` parameter. In this case, the allowable values for the `mode` parameter and their respective meanings are given in the table below. Note, however, that some devices support only a subset of the hardware interrupt channels. See the table in the Resource Usage section for details of the supported interrupt channels and the interrupt input pin for each device.

**Hardware Interrupt Mode Values**

| Value | Built-in Constant             | Interrupt Trigger                              |
|-------|-------------------------------|--|
| &H10  | <code>zxPinLow</code>         | A low level on the interrupt pin.              |
| &H14  | <code>zxPinChange</code>      | Any logic level change on the interrupt pin.   |
| &H18  | <code>zxPinFallingEdge</code> | A high to low transition on the interrupt pin. |
| &H1C  | <code>zxPinRisingEdge</code>  | A low to high transition on the interrupt pin. |

All other values are reserved for future use. For compatibility with BasicX, there are similarly named built-in constants that begin with the prefix `bx` instead of `zx` except that there is no equivalent for `zxPinChange`. Additionally, on mega32-based devices, Interrupt 2 is not capable of the first two trigger modes; it can only be triggered on a rising edge or a falling edge.

The built-in constants `WaitInt0` through `WaitInt7` may be used to specify the `intNum` parameter. If no `intNum` parameter is given, Interrupt 1 is assumed (for compatibility with BasicX). This is equivalent to using `WaitForInterrupt(mode, 1)`.

## Analog Comparator Interrupt

A task may await an analog comparator interrupt by specifying the value &H10 for `intNum`. The corresponding built-in constant is `waitAnalogComp`. In this case, the `mode` parameter specifies the comparator output transition that will cause the interrupt to occur.

**Analog Comparator Interrupt Mode Values**

| Value | Built-in Constant                | Interrupt Trigger                              |
|-------|----------------------------------|--|
| &H00  | <code>zxAnalogCompChange</code>  | Comparator output rising edge or falling edge. |
| &H02  | <code>zxAnalogCompFalling</code> | Comparator output falling edge.                |
| &H03  | <code>zxAnalogCompRising</code>  | Comparator output rising edge.                 |

With all of the `mode` values in the table above, the analog comparator's positive input is AIN0 (Port B, bit 2) and the comparator's negative input is either AIN1 (Port B, bit 3) or, if the ACME bit is set in a CPU register (see below), the analog input specified by the multiplexor select bits in Register.ADMUX. On the ZX-24 models, AIN0 is common with Port A, bit 2 so the latter I/O pin will need to be configured to be an input in high-impedance mode. Also, on the ZX-24 models built using boards earlier than Rev 5 (see the bottom side of the board), AIN1 has no external connection so the negative input must be supplied via the analog multiplexor. The ACME bit is in Register.SFIOR for the mega32 and mega128-based ZX models and in Register.ADCSRB in the mega644, mega644P, mega1280 and mega1281-based ZX models.

Another option for the positive comparator input is to select the internal "band gap" voltage. This voltage level (approximately 1.23 volts) is selected by adding &H40 to the mode values in the table above. The built-in constant `zxAnalogReference` has this value.

See the section in the Atmel microcontroller documentation describing the analog comparator for further details.

## Pin Change Interrupts

For ZX models based on the mega644, mega644P, mega1280 and mega1281 CPUs, a task may await a state change on one or more pins of an I/O port. This mode is selected by specifying a special value for the `intNum` parameter according to the tables below for the respective CPU types.

| intNum Values for Pin Change Interrupts |                             |                      |                                     |                                   |
|---|-----------------------------|----------------------|-------------------------------------|-----------------------------------|
| Value                                   | Built-in Constant           | Trigger – mega644,P  | Trigger – mega1281                  | Trigger – mega1280                |
| &H20                                    | <code>WaitPinChangeA</code> | Pin change on Port A |                                     |                                   |
| &H21                                    | <code>WaitPinChangeB</code> | Pin change on Port B | Pin change on Port B                | Pin change on Port B              |
| &H22                                    | <code>WaitPinChangeC</code> | Pin change on Port C |                                     |                                   |
| &H23                                    | <code>WaitPinChangeD</code> | Pin change on Port D |                                     |                                   |
| &H24                                    | <code>WaitPinChangeE</code> |                      | Change on Port Bit E.0 <sup>1</sup> |                                   |
| &H29                                    | <code>WaitPinChangeJ</code> |                      |                                     | Pin change on Port J <sup>2</sup> |
| &H2a                                    | <code>WaitPinChangeK</code> |                      |                                     | Pin change on Port K              |

<sup>1</sup>Not available on the ZX-1281e.

<sup>2</sup>Bits 0-6 only.

For each of the `intNum` values in the table above, the `mode` parameter specifies pin change interrupt enable bits corresponding to each pin of the port. For example, if the `mode` value is &H21, a pin change interrupt will be generated if either bit 0 or bit 5 of the specified port changes state. Clearly, a `mode` value of zero is useless since no pin change interrupt can ever occur in that case.

When the trigger condition occurs an interrupt will be generated and the task awaiting the interrupt will rise to the highest priority. This will cause an immediate task switch meaning that the next instruction that executes will be the one following the `WaitForInterrupt()` invocation. Note that if another task performs an action that causes interrupts to be disabled, response to the interrupt will be delayed until interrupts are re-enabled. The fact that the current task is locked does not prevent the interrupt task from executing next.

## Interrupt Priority

If two or more interrupts occur simultaneously, the task awaiting the highest priority interrupt is activated first. The priorities of the various interrupts are given in the table below.

| Interrupt Priority (highest to lowest) |
|--|
| Interrupt 0                            |
| Interrupt 1                            |
| Interrupt 2                            |
| Analog Comparator Interrupt            |
| Interrupt 3                            |
| Interrupt 4                            |
| Interrupt 5                            |
| Interrupt 6                            |
| Interrupt 7                            |
| Pin Change Interrupt, Port A           |
| Pin Change Interrupt, Port B           |
| Pin Change Interrupt, Port C           |
| Pin Change Interrupt, Port D           |
| Pin Change Interrupt, Port E           |
| Pin Change Interrupt, Port J           |
| Pin Change Interrupt, Port K           |

Note that a task awaiting an interrupt will exhibit some latency between the occurrence of the interrupt and when the waiting task begins execution. The latency depends on a number of factors including the specific instruction being executed at the time of the interrupt and the number and frequency of system interrupts that need to be handled. Instructions that may take a long time to execute such as `OutputCapture()`, `ShiftIn()`, `ShiftOut()`, `X10Cmd()`, etc. will introduce more latency than simple instructions like assigning a value to a variable.

## Examples

```
Call WaitForInterrupt(zxPinChange)
Call WaitForInterrupt(zxPinRisingEdge, WaitInt2)
Call WaitForInterrupt(&H40, WaitPinChangeA) ' await a change on Port A, bit 6
```

## Resource Usage

Only one task can be awaiting each interrupt at any particular time. If a task is already awaiting the specified interrupt, another call to `WaitForInterrupt()` for that same interrupt will return immediately.

Also, on the ZX-24 the interrupt pins are common with I/O pins as shown in the table below. This means that you should set the corresponding pin to be an input (either tri-state or pull-up) when you want to use `WaitForInterrupt()`. Note, however, that if the pin is an output and a task is awaiting an interrupt, a transition on the corresponding output can generate the interrupt for the waiting task. This may be of use in special situations as a “software interrupt”.

| Interrupt and I/O Pin Sharing for<br>ZX-24, ZX-24a, ZX-24p, ZX-24n |               |     |
|--|---------------|-----|
| Interrupt  | Port/Bit      | Pin |
| 0  | Port C, Bit 6 | 6   |
| 1  | Port C, Bit 1 | 11  |
| 2  | Port A, Bit 2 | 18  |

| ZX Models                                 | Interrupt Input Pins |      |      |      |      |      |      |      |      |      |
|---|----------------------|------|------|------|------|------|------|------|------|------|
|   | INT0                 | INT1 | INT2 | INT3 | INT4 | INT5 | INT6 | INT7 | AIN0 | AIN1 |
| ZX-24, ZX-24a, ZX-24p, ZX-24n             | 6                    | 11   | 18   | -    | -    | -    | -    | -    | 18   | -    |
| ZX-40, ZX-40a, ZX-40p, ZX-40n             | 16                   | 17   | 3    | -    | -    | -    | -    | -    | 3    | 4    |
| ZX-44, ZX-44a, ZX-44p, ZX-44n             | 11                   | 12   | 42   | -    | -    | -    | -    | -    | 42   | 43   |
| ZX-24e, ZX-24ae,<br>ZX-24pe, ZX-24ne      | 18                   | 17   | 26   | -    | -    | -    | -    | -    | 26   | 25   |
| ZX-1281, ZX-1281n                         | 25                   | 26   | -    | -    | 6    | 7    | 8    | 9    | 4    | 5    |
| ZX-1280, ZX-1280n                         | 43                   | 44   | 45   | 46   | 6    | 7    | 8    | 9    | 4    | 5    |
| ZX-128e, ZX-128ne,<br>ZX-1281e, ZX-1281ne | 12                   | 11   | 10   | 9    | 16   | 15   | 14   | 13   | 18   | 17   |

In the Interrupt Input Pin table above, the columns for INT2 and INT3 indicate that these interrupts are not available for the ZX-1281. That is because the corresponding I/O pins are used for serial channel Com1. For the ZX-1280, ZX-128e and ZX-1281e, INT2 and INT 3 will not be available if serial channel Com2 is in use. Also, INT0 and INT1 are not available on mega128, mega1281 and mega1280-based devices when I2C channel 0 is in use since the same pins are used for the SCL and SDA signals.

For native code devices, the following table lists the ISRs that may be included in your program if it invokes WaitForInterrupt(). The compiler will attempt to include only those ISRs that are required based on what it can determine from analysis of the various invocations. If the compiler is unable to determine which specific ISR is require, all those listed will be included.

| ISRs Required  |                    |  |
|----------------|--------------------|--|
| Underlying CPU | Int Type           | ISR Name                                       |
| mega644P       | External Interrupt | INT0, INT1, INT2                               |
|                | Analog Comparator  | Analog_Comp                                    |
|                | Pin Change Int.    | PCINT0, PCINT1, PCINT2, PCINT3                 |
| mega128        | External Interrupt | INT0, INT1, INT2, INT3, INT4, INT5, INT6, INT7 |
|                | Analog Comparator  | Analog_Comp                                    |
| mega1281       | External Interrupt | INT0, INT1, INT2, INT3, INT4, INT5, INT6, INT7 |
|                | Analog Comparator  | Analog_Comp                                    |
|                | Pin Change Int.    | PCINT0, PCINT1, PCINT2                         |
| mega1280       | External Interrupt | INT0, INT1, INT2, INT3, INT4, INT5, INT6, INT7 |
|                | Analog Comparator  | Analog_Comp                                    |
|                | Pin Change Int.    | PCINT0, PCINT1, PCINT2                         |

## Compatibility

The second parameter is not supported in BasicX compatibility mode. The built-in constant `zxPinChange` is not available in BasicX. It is not known if the capability is supported or not.

# WaitForInterval

**Type** Subroutine

**Invocation** WaitForInterval(flags)

| Parameter | Method | Type | Description  |
|-----------|--------|------|--|
| flags     | ByVal  | Byte | A set of flag bits that control the operation. See the discussion below. |

## Discussion

This routine allows a task to suspend itself and wait for an interval timer to expire. The length of the interval is set by the routine `SetInterval()`. Note that there is only one interval timer that is shared by all tasks. This means that at most one task may be awaiting the expiration of an interval at any one time. If another task is already awaiting an interval, calls to `WaitForInterval()` will return immediately.

The bit values for the `flags` parameter are described in the table below.

| Interval Timer Flag Values |   |
|----------------------------|---|
| Value                      | Description                                       |
| &H01                       | Wait until the next interval expires.             |
| &H02                       | Reset the interval counter to its original value. |

The remaining bits are currently undefined but may be employed in the future.

After a call to `SetInterval()` the interval counter is decremented on every RTC tick. When it reaches zero, if a task is awaiting the expiration of the interval, that task will be scheduled to run immediately. If no task is awaiting the expiration of the interval, the fact that the interval expired is recorded and the interval counter is reset to the original value.

If the `flags` value is zero when a task calls `WaitForInterval()`, and an interval expiration has previously been recorded (with no waiting task), the call will return immediately. Otherwise, the task will be suspended until the interval expiration. If the `flags` value is &H01, the task will be suspended until the next expiration of the interval. If the `flags` value is &H03, interval counter will be reloaded and then the task will be suspended until the interval expires. The last mode of operation is similar to a task calling `Sleep()`. The difference is that when the interval expires, the task is immediately reactivated. With a `Sleep()` call, the task will execute again when its sequential turn comes up.

A task awaiting the expiration of an interval has lower priority than one awaiting an interrupt. Note that a task awaiting the expiration of an interval will exhibit some latency between the expiration of the interval and when the waiting task begins execution. The latency depends on a number of factors including the specific instruction being executed at the time and the number and frequency of system interrupts that need to be handled. Instructions that may take a long time to execute such as `OutputCapture()`, `ShiftIn()`, `ShiftOut()`, `X10Cmd()`, etc. will introduce more latency than simple instructions like assigning a value to a variable.

## Example

```
Call SetInterval(1.0)
Do
    Call WaitForInterval(0)
    <other code here>
Loop
```



## Resource Usage

The interval counter is driven off of the real time clock. If interrupts are disabled for long periods of time, the timing won't be accurate. I/O routines that disable interrupts typically track RTC ticks and then update the RTC when the I/O process has completed. At this same time, the interval counter will be updated as well accounting for, at most, one missed expiration.

There is a single, system-wide interval timer. Only one task can be awaiting an interval at a time. If a task is already waiting, another call to `WaitForInterval()` will return immediately.

## Compatibility

This routine is not available in BasicX compatibility mode.

**See Also**      `SetInterval`

# WatchDog

---

**Type** Subroutine

**Invocation** WatchDog()

## Discussion

This routine resets the watchdog timer, preventing it from resetting the system. A watchdog timer is useful to ensure that your program continues to operate normally.

To implement a watchdog timer you first call `OpenWatchDog ( )` to prepare the watchdog timer for use. Thereafter, if your program doesn't call `WatchDog ( )` often enough, the watchdog will eventually time out and cause a system reset.

**See Also** CloseWatchDog, OpenWatchDog

# X10Cmd

**Type** Subroutine

**Invocation** X10Cmd(outPin, syncPin, house, devCmd, count)  
X10Cmd(outPin, syncPin, house, devCmd, count, flags)

| Parameter | Method | Type | Description   |
|-----------|--------|------|---|
| outPin    | ByVal  | Byte | The pin on which the X10 signal will be generated.      |
| syncPin   | ByVal  | Byte | The pin on which the 60Hz sync signal will be received. |
| house     | ByVal  | Byte | The house code.   |
| devCmd    | ByVal  | Byte | The device code or command code.                        |
| count     | ByVal  | Byte | The number of times to repeat the transmission.         |
| flags     | ByVal  | Byte | Flag bits to control the operation of the command.      |

## Discussion

This routine produces an X-10 compatible signal on the pin specified by `outPin`. The signal is synchronized to the zero-crossing signal on the pin specified by `syncPin`. The generated signal will include the specified house code and command/device code and will be repeated the specified number of times without any spacing between the code sequences. The X-10 specification indicates that most commands should be repeated twice and that successive commands should be separated by at least 3 power line cycles (~50 milliseconds). The exception is for bright and dim commands that can be repeated any number of times.

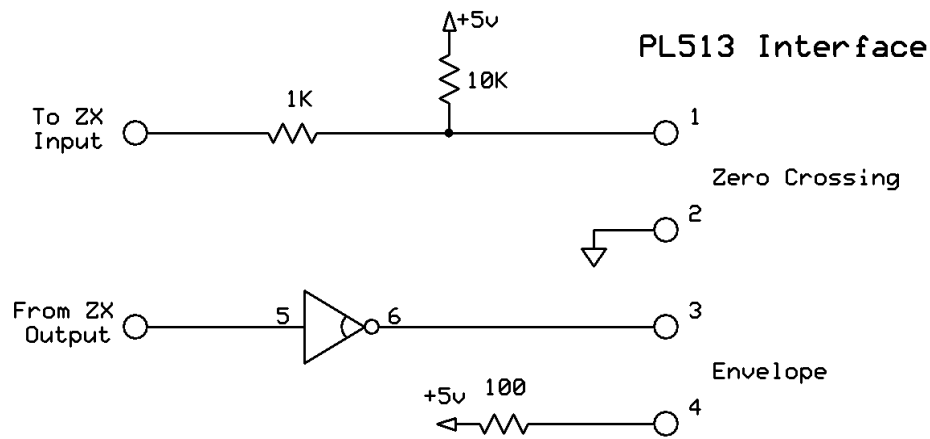
If the `flags` parameter is not present, the transmission is implemented as a single 1millisecond pulse near the edge of the zero crossing signal. If the `flags` parameter is present it has the effects shown in the table below depending on the value of the parameter.

| Function           | Hex Value | Bit Mask    |
|--------------------|-----------|-------------|
| Three-phase output | &H01      | xx xx xx x1 |
| 50Hz timing        | &H02      | xx xx xx 1x |

If the three-phase output flag bit is asserted, three 1 millisecond pulses will be output during each half-cycle. The 50Hz timing flag is used to control the phase timing of the three-phase output. If the flag bit is not asserted, 60Hz timing is utilized. This flag bit is only used when generating three-phase output.

## External Circuitry

In order to control X-10 devices, you will need a power line interface device such as the PL513 or the TW523, both of which are available from a variety of sources. The technical documentation for both interface devices is available on the Internet. A simplified interface between the ZX and the PL513 is shown below. Note that this circuit will not work for the TW523. The suggested OEM circuit in the X10 Technical Note, or something similar, should be used.



**Simple PL513 Interface**

### Example

The code below sends the commands to turn on device A1.

```
Const HouseCodeA as Byte = &H06
Const DeviceCode1 as Byte = &H0c
Const DeviceOn as Byte = &H05

Call X10Cmd(20, 19, HouseCodeA, DeviceCode1, 2)
Call Delay(0.50)
Call X10Cmd(20, 19, HouseCodeA, DeviceOn, 2)
```

### Compatibility

The BasicX documentation indicates that the transmission process is done in the background. On this implementation X10Cmd( ) will not return until the transmission is complete. In BasicX compatibility mode the flags parameter is not supported.

# Yield

---

**Type** Subroutine

**Invocation** Yield()

## Discussion

This routine is can be called whenever it is desirable to allow another task to run that is ready to run. One particular situation in which it is useful is at the end of a long process during which UpdateRTC() has been called one or more times. Normally, when an RTC interrupt occurs a task switch is performed immediately if the current task's time slice has expired or if a task is awaiting the expiration of an interval and the interval period has elapsed. However, if interrupts are disabled this automatic task switch cannot be performed. A call to UpdateRTC() will prepare the system for an eventual task switch which is then triggered by a call to Yield().

## Example

See the example at UpdateRTC.

**See Also** UpdateRTC

# ZXCmdMode

---

|                   |                                     |
|-------------------|-------------------------------------|
| <b>Type</b>       | Subroutine                          |
| <b>Invocation</b> | ZXCmdMode()<br>ZXCmdMode(highSpeed) |

| Parameter | Method | Type    | Description   |
|-----------|--------|---------|---|
| highSpeed | ByVal  | Boolean | A flag controlling the communication speed in command mode. |

## Discussion

This routine causes the ZX to stop executing your application and enter “command mode”. When in command mode, the ZX will respond to download commands and other special commands. When invoked with no parameters, command mode is invoked at the default Com1 baud rate (typically 19.2K baud). If the `highSpeed` parameter is specified and it is `True`, command mode is invoked at the download baud rate (115.2K baud).

You can use this routine in your application to facilitate downloading triggered by some particular event, e.g. receipt of a certain character or sequence of characters, the occurrence of an external signal, etc. You can use the downloader DLL source code (installed as part of ZBasic) to construct a special purpose downloader for your application. Alternately, if your application detects receipt of an “ATN character” and then invokes `ZXCmdMode( )` in slow speed mode, you can use the ZLoad command line utility or the ZBasic IDE to perform downloading without needing to have DTR connected to the device.

## Example

```
Call ZXCmdMode( )
```