ADC Library

Module and Filename

ADC
ADC.bas
Interface

Subroutines and Functions
function Read (pChannel **as byte**) **as word**
function ReadMean (pChannel **as byte**, pRange **as byte** = 64) **as word**
function ReadMedian (pChannel **as byte**) **as word**
sub SetAcqTime (pATime **as byte**)
sub SetConfig (pConfig **as byte**)
sub SetConvTime (pValue **as byte**)
Variables
ADResult **as word**
Convert **as boolean**
Enabled **as boolean**
RightJustify **as boolean**

Overview
Access the the PIC® microcontroller Analog To Digital (ADC) converter. The AD conversion will take an analog signal and convert into a 10 bit number. The number of channels available will depend on the device used. You should refer to corresponding datasheet for more information.

Example Code

```
// LCD options...
#option LCD_DATA = PORTD.4
#option LCD_RS = PORTE.0
#option LCD_EN = PORTE.1
// uses LCD and AD libraries...
include "LCD.bas"
include "ADC.bas"
include "convert.bas"
// read the AD port and scale for 0 - 5 volts...
function ADInAsVolt() as word
result = (ADC.Read(0) + 1) * 500 / 1024
end function
// sampled AD value...
dim ADVal as word
// initialise and clear LCD...
ADCON1 = $07 // PORTE as digital (LCD)
TRISA.0 = 1 // configure AN0 as an input
ADCON1.7 = 1 // set analogue input on PORTA.0
delayms (500)
LCD.Cls
// main program loop...
while true
ADVal = ADInAsVolt
LCD.MoveCursor (1,1)
LCD.Write("DC Volts = ", DecToStr(ADVal / 100), ".", DecToStr(ADVal, 2), " ")
delayms(250)
wend
```

Interface

function Read(pChannel **as byte**) **as word**

- *pChannel* - the channel number to read. Valid arguments are AN0, AN1..ANx. You should refer to your particular device datasheet to obtain the upper limit for ANx.

Read the ADC and return 10 bit result. Read uses a delay for the acquisition time, which has the following minimum overheads for values passed to SetAcqTime that are greater than zero.
4Mhz - 24us
8Mhz - 12us
10Mhz - 8us
16Mhz - 5us
20Mhz plus - 2us

function ReadMean(pChannel **as byte**, pRange **as byte** = 64) **as word**

- *pChannel* - the channel number to read. Valid arguments are AN0, AN1..ANx. You should refer to your particular device datasheet to obtain the upper limit for ANx.
- *pRange* - the number of samples to make. The default is 64.

Takes pRange ADC samples and returns the mean average. Much quicker than median average, but can be influenced by extreme high and low sample values.

function ReadMedian(pChannel **as byte**) **as word**

- *pChannel* - the channel number to read. Valid arguments are AN0, AN1..ANx. You should refer to your particular device datasheet to obtain the upper limit for ANx.

Takes 64 ADC samples and returns the median average. Although computationally expensive, this routine is quite useful as the result is not influenced by extreme high and low sample values.

sub SetAcqTime(pATime **as byte**)

- *pATime* - required acquisition time

Sets the ADC acquisition time, in microseconds (us).

sub SetConfig(pConfig **as byte**)

- *pConfig* - ADC configuration control bits. You should refer to your particular device datasheet for further information.

Set configuration control bits.

sub SetConvTime(pValue **as byte**)

- *pValue* - ADC conversion time. Valid arguments are FOSC_2, FOSC_4, FOSC_8, FOSC_16, FOSC_32, FOSC_64 and FRC. You should refer to your particular device datasheet for further information.

Set the ADC conversion time

ADResult **as word**

After making a call to Read, ADResult will hold the last ADC sample value.

Convert **as boolean**

Assigning true will start the ADC conversion process. You should not normally have to access this variable directly. To obtain an ADC reading, use Read instead.

Enabled **as boolean**
Assigning true will enable the ADC module. You should not normally have to access this variable directly. To obtain an ADC reading, use Read instead.

RightJustify **as boolean**
The ADC result is a 10 bit value which held in a 16 bit word. Setting RightJustify to true will right justify the 10 bit result. Setting RightJustify to false will left justify the 10 bit result. The ADC result is left justified by default.

Convert Library

Module and Filename

Convert
Convert.bas
Interface
Subroutines and Functions

function StrToDec (pValue **as string**) **as longword**
function HexToDec (pValue **as string**) **as longword**
function BinToDec (pValue **as string**) **as longword**
function IsDecValid (pValue **as string**) **as boolean**
function IsHexValid (pValue **as string**) **as boolean**
function IsBinValid (pValue **as string**) **as boolean**
function DecToStr (pValue **as type** [, pPad **as byte**, pPadChar **as char** = "0"]) **as string**
function HexToStr (pValue **as type** [, pPad **as byte**, pPadChar **as char** = "0"]) **as string**
function BinToStr (pValue **as type** [, pPad **as byte**, pPadChar **as char** = "0"]) **as string**
function FloatToStr (pValue **as float**, pNumDigits **as byte** = 3) **as string**
function FloatToStr (pValue **as float**, pPad, pNumDigits **as byte**, pPadChar **as char** = "0") **as string**
function DecToBCD (pValue **as byte**) **as byte**
function BCDToDec (pValue **as byte**) **as byte**

Overview
Number conversion library.
Interface

function StrToDec(pValue **as string**) **as longword**

- *pValue* - String to convert

Converts a string representation of a decimal number and converts it to an ordinal type.

function HexToDec(pValue **as string**) **as longword**

- *pValue* - String to convert

Converts a string representation of a hexademimal number and converts it to an ordinal type.

function BinToDec(pValue **as string**) **as longword**

- *pValue* - String to convert

Converts a string representation of a binary number and converts it to an ordinal type.

function IsDecValid(pValue **as string**) **as boolean**

- *pValue* - String to check

Checks to see if a string representation of a decimal number is valid. The function returns true if it is, false otherwise. It is sometimes useful to make a call to this

routine before a call to StrToDec.

function IsHexValid(pValue **as string**) **as boolean**

- *pValue* - String to check

Checks to see if a string representation of a hexadecimal number is valid. The function returns true if it is, false otherwise. It is sometimes useful to make a call to this routine before a call to HexToDec.

function IsBinValid(pValue **as string**) **as boolean**

- *pValue* - String to check

Checks to see if a string representation of a binary number is valid. The function returns true if it is, false otherwise. It is sometimes useful to make a call to this routine before a call to BinToDec.

function DecToStr(pValue **as type** [, pPad **as byte**, pPadChar **as char** = "0"]) **as string**

- *pValue* - The number to convert. The argument can be of type boolean, byte, shortint, word, integer, longword or longint.
- *pPad* - An optional parameter which will pack the left hand side of the returned string with (pPad - length of numeric string) zeros.
- *pPadChar* - An optional parameter which changes the default packing character.

This function will convert a number and return a string decimal string representation of the number passed. Optional *pPad* and *pPadChar* arguments can be used to format the returned string value. For example,
MyString = DecToStr(42) *// result is "42"*
MyString = DecToStr(42,5) *// result is "00042"*
MyString = DecToStr(42,5,"#") *// result is "###42"*

function HexToStr(pValue **as type** [, pPad **as byte**, pPadChar **as char** = "0"]) **as string**

- *pValue* - The number to convert. The argument can be of type byte, shortint, word, integer, longword or longint.
- *pPad* - An optional parameter which will pack the left hand side of the returned string with (pPad - length of numeric string) zeros.
- *pPadChar* - An optional parameter which changes the default packing character.

This function will convert a number and return a string hexadecimal string representation of the number passed. Optional *pPad* and *pPadChar* arguments can be used to format the returned string value. For example,
MyString = HexToStr(210) *// result is "D2"*
MyString = HexToStr(210,5) *// result is "000D2"*
MyString = HexToStr(210,5,"#") *// result is "###D2"*

function BinToStr(pValue **as type** [, pPad **as byte**, pPadChar **as char** = "0"]) **as string**

- *pValue* - The number to convert. The argument can be of type byte, shortint, word, integer, longword or longint.
- *pPad* - An optional parameter which will pack the left hand side of the

returned string with (pPad - length of numeric string) zeros.

- *pPadChar* - An optional parameter which changes the default packing character.

This function will convert a number and return a string binary representation of the number passed. Optional *pPad* and *pPadChar* arguments can be used to format the returned string value. For example,
MyString = BinToStr(12) *// result is "1100"*
MyString = BinToStr(12,5) *// result is "01100"*
MyString = BinToStr(12,5,"#") *// result is "#1100"*

function FloatToStr (pValue **as float**, pNumDigits **as byte** = 3) **as string**
function FloatToStr (pValue **as float**, pPad, pNumDigits **as byte**, pPadChar **as char** = "0") **as string**

- *pValue* - The floating point number to convert.
- pNumDigits - An optional parameter which specifies the number of digits after the floating point. The default is 3 digits.
- *pPad* - an optional parameter which will pack the left hand side of the returned string with (pPad - length of numeric string) zeros.
- pPadChar - an optional parameter which changes the default packing character.

This function will convert a number and return a string floating point representation of the number passed. Optional *pNumDigits*, *pPad* and *pPadChar* arguments can be used to format the returned string value. For example,
MyString = FloatToStr(42) *// result is "42.000"*
MyString = FloatToStr(42,1) *// result is "42.0"*
MyString = FloatToStr(42,5,1) *// result is "00042.0"*
MyString = FloatToStr(42,5,1,"#") *// result is "###42.0"*

function DecToBCD(pValue **as byte**) **as byte**

- *pValue* - The number to convert.

Returns a packed Binary Coded Decimal (BCD) value

function BCDToDec(pValue **as byte**) **as byte**

- *pValue* - The BCD number to convert.

Unpacks a Binary Coded Decimal (BCD) value.

EEPROM Library

Module and Filename

EE
EEPROM.bas
Interface
Subroutines and Functions

function ReadByte (pAddress **as** TAddress) **as byte**
function ReadBoolean (pAddress **as** TAddress) **as boolean**
function ReadWord (pAddress **as** TAddress) **as word**
function ReadLongWord (pAddress **as** TAddress) **as longword**
function ReadFloat (pAddress **as** TAddress) **as float**
sub WriteByte (pAddress **as** TAddress, pValue **as byte**)
sub WriteBoolean (pAddress **as** TAddress,pValue **as boolean**)
sub WriteWord (pAddress **as** TAddress,pValue **as word**)
sub WriteLongWord (pAddress **as** TAddress, pValue **as longword**)
sub WriteFloat (pAddress **as** TAddress, pValue **as float**)
compound sub Read (pAddress **as** TAddress, **byref** pReadItem)
compound sub Write (pAddress **as** TAddress, pWriteItem)
Variables
Address **as** TAddress

Overview

Microcontroller EEPROM read and write library. Note that TAddress is byte size for devices that have less than 256 bytes of onboard EEPROM. For devices that have more than 256 of onboard EEPROM, TAddress is a word.

Example Code

```
// import modules...
include "USART.bas"
include "EEPROM.bas"
include "Convert.bas"
// working variables...
dim Value as byte
dim Str as string
// write data to EEPROM...
EE.Write(0,"EEPROM TEST, Value = ", 123)
// read data and display...
USART.SetBaudrate(br19200)
EE.Read(0, Str, Value)
USART.Write(Str, DecToStr(Value), 13, 10)
```

Interface

function ReadByte(pAddress **as** TAddress) **as byte**

- *pAddress* - The EEPROM address location.

Read a single byte from microcontroller EEPROM.

function ReadBoolean(pAddress **as** TAddress) **as boolean**

- *pAddress* - The EEPROM address location.

Read a boolean from microcontroller EEPROM.

function ReadWord(pAddress **as** TAddress) **as word**

- *pAddress* - The EEPROM address location.

Read a word or integer from microcontroller EEPROM.

function ReadLongWord(pAddress **as** TAddress) **as longword**

- *pAddress* - The EEPROM address location.

Read a long word or long integer from microcontroller EEPROM.

function ReadFloat(pAddress **as** TAddress) **as float**

- *pAddress* - The EEPROM address location.

Read a floating point number from microcontroller EEPROM.

sub WriteByte(pAddress **as** TAddress, pValue **as byte**)

- *pAddress* - The EEPROM address location.
- *pValue* - Data to write.

Write a single byte to microcontroller EEPROM.

sub WriteBoolean(pAddress **as** TAddress,pValue **as boolean**)

- *pAddress* - The EEPROM address location.
- *pValue* - Data to write.

Write a boolean to microcontroller EEPROM. Note that writing a single boolean value will occupy one byte of EEPROM storage.

sub WriteWord(pAddress **as** TAddress,pValue **as word**)

- *pAddress* - The EEPROM address location.
- *pValue* - Data to write.

Write a word or integer to microcontroller EEPROM.

sub WriteLongWord(pAddress **as** TAddress, pValue **as longword**)

- *pAddress* - The EEPROM address location.
- *pValue* - Data to write.

Write a long word or long integer to microcontroller EEPROM.

sub WriteFloat(pAddress **as** TAddress, pValue **as float**)

- *pAddress* - The EEPROM address location.
- *pValue* - Data to write.

Write a floating point number to microcontroller EEPROM.

compound sub Read(pAddress **as** TAddress, **byref** pReadItem)

- *pAddress* - The EEPROM address location.
- *pReadItem* - Data to read.

Read multiple items from microcontroller EEPROM. Valid argument types include boolean, char, string, byte, shortint, word, integer, longword, longint and floating point.

compound sub Write (pAddress **as** TAddress, pWriteItem)

- *pAddress* - The EEPROM address location.
- *pWriteItem* - Data to write.

Write multiple items to microcontroller EEPROM. Valid argument types include boolean, char, string, byte, shortint, word, integer, longword, longint and floating point.

Address **as** TAddress

Holds the current EEPROM address location.

GLCD Library

Module and Filename

GLCD
GLCD.bas
Interface
Subroutines and Functions

sub Cls ()
sub SetPixel (pX, pY **as byte**)
sub SetImage (pX, pY **as** TXY, **byrefconst** pImage() **as byte**)
sub Line (x1, y1, x2, y2 **as** TXY)
sub Ellipse (pCX, pCY, pXRadius, pYRadius **as** TXY)
sub Circle (pCX, pCY, pRadius **as** TXY)
sub Rectangle (px1, py1, px2, py2 **as** TXY)
sub Square (pX, pY, pSize **as** TXY)
sub MoveTo (pX, pY **as** TXY)
sub LineTo (pX, pY **as** TXY)
compound sub Write (WriteItem)
compound sub WriteAt (pX, pY **as** TXY, WriteItem)
sub WriteStr (pX, pY **as** TXY, pStr **as string**)
sub SetFont (**byrefconst** pFont() **as byte**)
function TextWidth (pStr **as string**) **as** TXY
Variables

Pos as TPosition
Pen **as** TPen
Brush **as** TBrush
Font **as** TFont
TextAlign **as byte**

Compile Options

GLCD_MODEL

Overview
The GLCD library enable you to access drawing primitives, text and image rendering for a particular display type. Using the GLCD library is extremely easy, for example

```
#option GLCD_MODEL = S1D15G00 // GLCD driver
include "GLCD.bas" // GLCD library
include "Graphics.bas" // graphics library
include "Verdana.bas" // font library
// program start...
SetContrast(158)
Cls
Brush.Style = bsSolid
Brush.Color = $00
Pen.Color = $FF
SetFont(VerdanaBold)
WriteAt(2,58,"swordfish compiler")
```

Notice the #option at the top of the program. This tells the GLCD library which driver

to use. In the above example, it's a S1D15G00 used by the Nokia 6100 color display. If no driver option is specified, the GLCD library will use a Samsung KS0108 driver. The driver #option directive must always be declared before the GLCD library is included in your main program. In addition, it is essential that the GLCD library include is the **first include in your program**. Any other libraries, such as Graphics or Fonts, should be declared after the GLCD include, as shown above.
In the example above, it can be seen that you don't need to explicitly include a driver module in your main program. The relationship between GLCD, Graphics and a particular driver implementation is shown in the diagram below.

Different display types have different characteristics. For example, screen resolution, monochrome or color capability. This means that a particular display driver may introduce additional capabilty to the GLCD library, which may not be found on other display types. However, you still access specific driver routines through the GLCD library. For example,

```
#option GLCD_MODEL = S1D15G00 // GLCD driver
include "GLCD.bas" // GLCD library
SetContrast(158)
```

The set contrast subroutine is needed to correctly configure the S1D15G00. However,

```
#option GLCD_MODEL = KS0108 // GLCD driver
include "GLCD.bas" // GLCD library
SetContrast(158)
```

Would generate an error, as the KS0108 device does not support variable contrast settings. Also, some devices allow XORing of graphics and text to the display, whilst others do not. You should therefore study carefully the documentation for a particular display driver, to see what features are available.

Interface

sub Cls()
Clear the GLCD display.
***Cls* isn't physically implemented in the GLCD library, because it is very device dependant. However, it is a mandatory implementation requirement for all drivers and so is documented here.**

sub SetPixel(pX, pY **as byte**)

- *pX* - x position
- *py* - y position

Set a pixel at location *pX*, *pY* using the current Pen. Refer to individual driver documentation for supported Pen attributes.
***SetPixel* isn't physically implemented in the GLCD library, because it is very device dependant. However, it is a mandatory implementation requirement for all drivers and so is documented here.**

sub SetImage(pX, pY **as** TXY, **byrefconst** pImage() **as byte**)

- *pX* - x position
- *pY* - y position
- *pImage* - Image to display

Display an image at location pX, pY. By default, the GLCD will render monochrome images for 1 bit display drivers only. You should refer to individual driver documentation to to determine if this default implementation has been overridden, as some drivers are able to render full color images.
Swordfish compatible images can be created using the IDE Image Converter plugin.

sub Line(x1, y1, x2, y2 **as** TXY)

- *x1* - x start position
- *y1* - y start position
- *x2* - x end position
- *y2* - y end position

Draw a line from *x1*, *y1* to *x2*, *y2* using the current Pen. Refer to individual driver documentation for supported Pen attributes.

sub Ellipse(pCX, pCY, pXRadius, pYRadius **as** TXY)

- *pCX* - x centre
- *pCY* - y centre
- *pXRadius* - x radius
- *pYRadius* - y radius

Draw an ellipse at *pCX*, *pCY* using the current Pen. Refer to individual driver documentation for supported Pen attributes.

sub Circle(pCX, pCY, pRadius **as** TXY)

- *pCX* - x centre
- *pCY* - y centre
- *pRadius* - radius

Draw a circle at *pCX*, *pCY* using the current Pen. Refer to individual driver documentation for supported Pen attributes.

sub Rectangle(px1, py1, px2, py2 **as** TXY)

- *px1* - x start position
- *py1* - y start position
- *px2* - x end position
- *py2* - y end position

Draw a rectangle from *x1*, *y1* (top left) to *x2*, *y2* (bottom right) using the current Pen. Refer to individual driver documentation for supported Pen attributes.

sub Square(pX, pY, pSize **as** TXY)

- *pX* - x start position

- *pY* - y start position
- *pSize* - Size of the square

Draw a square from *pX*, *pY* (top left) of size *pSize* using the current Pen. Refer to individual driver documentation for supported Pen attributes.

sub MoveTo(pX, pY **as** TXY)

- *pX* - x start position
- *pY* - y start position

Move the cursor to position *pX*, *pY*

sub LineTo(pX, pY **as** TXY)

- *pX* - x end position
- *pY* - y end position

Draw a line from the current cursor position to *pX*, *pY* using the current Pen. You can set the cursor position by calling *MoveTo*.

compound sub Write(WriteItem)

- *WriteItem* - one or more char or strings

Write text to the display using the current font. For monochrome displays, set the font style to control how text is rendered to the screen. For color displays, the text is rendered using the current Pen (foreground) and Brush (background). Refer to individual driver documentation for supported Font, Pen and Brush attributes.
To control spacing between characters, set the font letter spacing.

compound sub WriteAt(pX, pY **as** TXY, WriteItem)

- *pX* - x start position
- *pY* - y start position
- *WriteItem* - one or more char or strings

Write text to the display using the current font at position *pX*, *pY*. For monochrome displays, set the font style to control how text is rendered to the screen. For color displays, the text is rendered using the current Pen (foreground) and Brush (background). Refer to individual driver documentation for supported Font, Pen and Brush attributes.
To control spacing between characters, set the font letter spacing.

sub WriteStr(pX, pY **as** TXY, pStr **as string**)

- *pX* - x start position
- *pY* - y start position
- *pStr* - string constant or variable

Write a single text string to the display using the current font at position *pX*, *pY*. For monochrome displays, set the font style to control how text is rendered to the screen. For color displays, the text is rendered using the current Pen (foreground) and Brush (background). Refer to individual driver documentation for supported Font, Pen and Brush attributes.
Unlike WriteAt, the justification of the string can be set using TextAlign.

sub SetFont (**byrefconst** pFont() **as byte**)

- *pFont* - The font table

Use a call to SetFont to load a chosen font before making any calls to the GLCD *Write*, *WriteAt* and *WriteStr* routines. This routine is imported from the Graphics library.
Swordfish compatible fonts can be created using the IDE Font Converter plugin.

function TextWidth (pStr **as string**) **as** TXY

- *pStr* - Constant or variable string.

Returns the width of a string, in pixels. This routine is imported from the Graphics library.

Pos as TPosition
The current cursor position. Imported from the driver module.

Pen **as** TPen

The current pen. Imported from Graphics.

Brush **as** TBrush

The current brush. Imported from Graphics.

Font **as** TFont
The current font. Imported from Graphics.

TextAlign **as byte**
Used to set the justification of text before a call to WriteStr. For example,
TextAlign = taLeft *// left justify*
TextAlign = taCenter *// center jusify*
TextAlign = taRight *// right justify*

#option GLCD_MODEL
Sets the GLCD driver model. Refer to individual driver documentation for supported devices.

I2C Library

Modules and Filenames

I2C, I2C.bas
I2C2, I2C2.bas
Interface

Subroutines and Functions
sub Initialize (pBaudrate **as byte** = I2C_100_KHZ, pSlew **as byte** = I2C_SLEW_OFF)
function IsIdle () **as boolean**
sub WaitForIdle ()
sub Start ()
sub Stop ()
sub Restart ()
sub Acknowledge (pAck **as bit** = I2C_ACKNOWLEDGE)
function ReadByte () **as byte**
sub WriteByte (pData **as byte**)
Variables
NotAcknowledged **as boolean**

Overview
Low level I2C interface library.

Example Code
```
// import libraries...
include "I2C.bas"
include "usart.bas"
// target 24LC128 I2C EEPROM device...
const I2C_EEPROM = $A0
// local variables...
dim
Value as byte,
Address as word
// program start...
Address = 0
I2C.Initialize
// write some data...
I2C.Start
I2C.WriteByte(I2C_EEPROM)
I2C.WriteByte(Address.Byte1)
I2C.WriteByte(Address.Byte0)
I2C.WriteByte("Z")
I2C.Stop
// allow external EEPROM to write data...
delayms(10)
// read the data back...
I2C.Start
I2C.WriteByte(I2C_EEPROM)
I2C.WriteByte(Address.Byte1)
I2C.WriteByte(Address.Byte0)
I2C.Restart
I2C.WriteByte(I2C_EEPROM + 1)
Value = I2C.ReadByte
```

```
I2C.Acknowledge(I2C_NOT_ACKNOWLEDGE)
I2C.Stop
// output the result
USART.SetBaudrate(br19200)
USART.Write("Value = ", Value, 13, 10)
```

Interface

sub Initialize(pBaudrate **as byte** = I2C_100_KHZ, pSlew **as byte** = I2C_SLEW_OFF)

- *pBaudrate* - Bus baudrate. Can be I2C_100_KHZ, I2C_400_KHZ or I2C_1000_KHZ
- *pSlew* - Slew rate. Can be I2C_SLEW_OFF or I2C_SLEW_ON

Initialise the I2C Bus.

function IsIdle() **as boolean**
Returns true if the I2C bus is idle, false otherwise.

sub WaitForIdle()
Blocking call which waits until the I2C bus is in an idle state.

sub Start()
Transmit a start condition on the I2C bus.

sub Stop()
Transmit a stop condition on the I2C bus.

sub Restart()
Transmit a restart condition on the I2C bus.

sub Acknowledge(pAck **as bit** = I2C_ACKNOWLEDGE)

- *pAck* - Bus acknowledge. Can be I2C_ACKNOWLEDGE or I2C_NOT_ACKNOWLEDGE

Transmit an acknowledge on the I2C bus.

function ReadByte() **as byte**
Read a byte from the I2C bus.

sub WriteByte(pData **as byte**)

- *pData* - Data to write

Write a byte to the I2C bus

NotAcknowledged **as boolean**

True if an acknowledge was not received from the slave device, false otherwise.

ISRRX Library

Module and Filename

ISRRX
ISRRX.bas

Interface

Subroutines and Functions

sub Initialize (pOnDataEvent **as** TROMAddress = 0)
sub Reset ()
sub Start ()
sub Stop ()
function DataAvailable () **as boolean**
function Overrun () **as boolean**
function ReadByte () **as byte**
function ReadWord () **as word**
function ReadLongWord () **as longword**
function ReadFloat () **as float**
function ReadStr (**byref** pText **as string**, pTerminator **as char** = **null**) **as byte**

Variables

USARTOverrun **as boolean**
BufferOverrun **as boolean**
DataByte **as byte**
DataChar **as char**
ProcessByte **as boolean**

Compile Options

RX_PRIORITY
RX_BUFFER_SIZE

Overview

Interrupt based USART receive library.

Example Code

```
include "USART.bas"
include "ISRRX.bas"
// RX OnData() event...
sub OnData()
// ignore CR and LF...
if ISRRX.DataByte = 13 or ISRRX.DataByte = 10 then
ISRRX.ProcessByte = false
// replace period with space character...
elseif ISRRX.DataChar = "." then
ISRRX.DataChar = " "
endif
end sub
' program start...
low(PORTD.0)
USART.SetBaudrate(br19200)
ISRRX.Initialize(@OnData)
' loop forever...
while true
delayms (500)
toggle(PORTD.0)
' read data from the buffer and output...
while ISRRX.DataAvailable
USART.Write(ISRRX.ReadByte)
wend
wend
```

Interface

sub Initialize(pOnDataEvent **as** TROMAddress = 0)

- pOnDataEvent - Address of optional subroutine event handler

Initializes the ISRRX module.

sub Reset()
Reset the ISRRX module.

sub Start()
Start buffering data. Data is automatically buffered after a call to *Initialize*. You should therefore only call *Start* after a call to *Stop*.

sub Stop()

Stop buffering data. A call to *Start* will restart data buffering.

function DataAvailable() **as boolean**
Returns true if there is data in the buffer

function Overrun() **as boolean**

Return true if a USART or buffer overrun error has occurred, false otherwise.

function ReadByte() **as byte**

Read a byte or short integer from the buffer. Use DataAvailable to see if the buffer has valid data.

function ReadWord() **as word**

Read a word or integer from the buffer. Use DataAvailable to see if the buffer has valid data.

function ReadLongWord() **as longword**
Read a long word or integer from the buffer. Use DataAvailable to see if the buffer has valid data.

function ReadFloat() **as float**
Read a floating point number from the buffer. Use DataAvailable to see if the buffer has valid data.

function ReadStr(**byref** pText **as string**, pTerminator **as char** = **null**) **as byte**

- pText - String variable to receive data
- pTerminator - The text terminator character

Read a string from the buffer. Use DataAvailable to see if the buffer has valid data.

USARTOverrun **as boolean**
A USART overrun error has occurred. Can be cleared with a call to *Reset*.

BufferOverrun **as boolean**

A buffer error has occurred. Can be cleared with a call to *Reset*.

DataByte **as byte**
If an event handler has been assigned to the interrupt, you can examine the byte received by reading *DataByte*. Note that *DataByte* should only be accessed in an event handler.

DataChar **as char**

If an event handler has been assigned to the interrupt, you can examine the character received by reading *DataChar*. Note that *DataChar* should only be accessed in an event handler.

ProcessByte **as boolean**

If an event handler has been assigned to the interrupt, you can choose to reject the incoming data by setting *ProcessByte* to false. This will prevent the received byte from being buffered. Note that *ProcessByte* should only be accessed in an event handler.

#option RX_PRIORITY

Set the interrupt priority level. Can be ipLow or ipHigh. By default, the option is set to ipHigh.

#option RX_BUFFER_SIZE

Set the buffer size. Can be 1..255 bytes in size. By default, the option is set to 64 bytes.

ISRTimer Library

Module and Filename

ISRTimer
ISRTimer.bas

Interface

Subroutines and Functions

sub Initialize (pCount **as byte** = MaxNumberOfTimers)
sub Start ()
sub Stop ()
sub Adjust (pAdjust **as integer**)

Variables

Items() **as** TTimerItem
ID **as byte**
Block **as boolean**

Compile Options

TIMER_PRIORITY
TIMER_INTERVAL_SIZE
TIMER_AVAILABLE
TIMER_AUTO_RELOAD
TIMER_REFRESH

Overview

Interrupt based timer library.

Example Code

```
// include ISR timer module and create constant ID
// to 2 * 16 bit timers...
include "ISRTimer.bas"
const
Timer1 = 0,
Timer2 = 1
// OnTimer event, flash LED...
sub OnTimer()
toggle(PORTD.0)
end sub
// initialise the timer module...
Timer.Initialize(2)
// initialise each timer - refresh is every 1000Hz (1ms)...
Timer.Items(Timer1).Interval = 100 // 100ms
Timer.Items(Timer1).OnTimer = @OnTimer // timer event handler
Timer.Items(Timer2).Interval = 2000 // 2000ms, no event handler
// enable the timers...
Timer.Items(Timer1).Enabled = true
Timer.Items(Timer2).Enabled = true
// start processing all timers...
Timer.Start
// main program loop...
while true
// this is a polled timer, not event driven - check to see
// if it has timeout...
if not Timer.Items(Timer2).Enabled then
toggle(PORTD.1)
Timer.Items(Timer2).Enabled = true
endif
// background flash LED...
high(PORTD.7)
delayms(500)
low(PORTD.7)
delayms(500)
wend
```

Interface

sub Initialize(pCount **as byte** = MaxNumberOfTimers)

- *pCount* - The number of timer items

Initializes the timer module using *pCount* number of 16 bit timers. By default, the library allocates RAM for four timer items. When initializing, *pCount* must be in the range 1 to MaxNumberOfTimers, which would be four. To allocate RAM for a higher number of timers, see TIMER_AVAILABLE.

sub Start()
Starts interrupt processing for all timer items.

sub Stop()

Stops all interrupt processing of timers items.

sub Adjust(pAdjust **as integer**)

- *pAdjust* - Adjustment value

Adjusts the timer by *pAdjust* instruction cycles. A given crystal is likely to have a tolerance which places it outside of the value stated. This routine allows you to compensate for that and also any latentencies experienced in executing the main Interrupt Service Routine (ISR). A negative value will slow things down, a positive value will speed things up. If you have already called *Start*, it is essential you call Stop before calling this routine. For example,

```
Timer.Stop
Timer.Adjust(-250)
Timer.Start
```

Items() **as** TTimerItem

An array of TTimerItem. The structure is composed of the following,

```
structure TTimerItem
Enabled as boolean
Interval as TInterval
OnTimer as TROMAddress
end structure
```

where

- *Enabled* - Enables or disables the timer item
- *Interval* - The timeout interval
- *OnTimer* - Timer event handler

To correctly initialize a timer item, you should set the enabled flag and also the timeout interval. The *OnTimer* event handler is optional. If it is not used, the timer object needs to be polled to determine if a timeout has occurred. See also TIMER_INTERVAL_SIZE, TIMER_AUTO_RELOAD and TIMER_REFRESH.

ID **as byte**

If an event handler has been assigned to a timer item, you can determine which item fired the event by checking its *ID*. Note that *ID* should only be accessed in an event handler.

Block **as boolean**

Assigning true to *Block* will stop all timer items from being processed in the timer modules interrupt handler. However, unlike *Stop*, the timer modules ISR is still triggered.

#option TIMER_PRIORITY

Set the interrupt priority level. Can be ipLow or ipHigh. By default, the option is set to ipHigh.

#option TIMER_INTERVAL_SIZE

By default, a timer item interval is 16 bits. Using this option enables you to specify an 8 bit (byte) or 16 bit (word) interval size.

#option TIMER_AVAILABLE

Sets the number of available timer items. This is the maximum number of timer items that will be available to a program. You can choose a subset of this number when *Initializing* the timer module. Valid values are 1 to 16. Note that allocating and using a large number of timer items will short interval timeouts is likely to affect system performance.

#option TIMER_AUTO_RELOAD

When a item times out, its interval value is automatically reloaded by default. Setting this option to false will remove this functionality from the timer module. The main program will then need to manually reload the timeout interval when an item times out.

#option TIMER_REFRESH

By default, the timer module counts time in 1 millisecond (ms) intervals (1000 Hz). You can slow the refresh to 10ms (100 Hz) by setting this option to 10. If the refresh interval is set to 10, it is important to note that item intervals should be scaled accordingly. For example,

```
#option TIMER_INTERVAL = 10
include "ISRTimer.bas"
Timer.Initialize(1)
Timer.Items(0).Interval = 100 // // this is 100 * 10, = 1000 ms
```

Keypad Library

Module and Filename

Keypad
Keypad.bas

Interface

Subroutines and Functions
function Value () **as byte**
sub WaitFor ()
Compile Options

KEYPAD_PORT

Overview

Keypad library for 4 x 4 keypad.

Example Code

```
// import libraries...
include "Keypad.bas"
include "USART.bas"
include "Convert.bas"
dim Value as byte
SetBaudrate(br19200)
// loop forever...
while true
Keypad.WaitFor
Value = Keypad.Value
if Value > 0 then
USART.Write("Value = ", DecToStr(Value), 13, 10)
endif
wend
```

Interface

function Value() **as byte**

Returns the button pressed as a number ranging from 1 to 16. A zero value indicates no key was pressed.

sub WaitFor()

A blocking subroutine call that waits for a change in keypad state. The routine will exit when a button has been pressed and also when released. This routine will only work if the keypad has been configured to work on PORTB.

#option KEYPAD_PORT
The keypad port to use. By default, the option is set to PORTB.

LCD Library

Module and Filename

LCD
LCD.bas
Interface

Subroutines and Functions

sub Cls ()
sub Command (pCommand **as byte**)
sub MoveCursor (pLine, pCol **as byte**)
compound sub Write (WriteItem)
compound sub WriteAt (pLine, pCol, WriteItem)

Compile Options

HYPERLINK \l "LCD_DATA"LCD_DATA
HYPERLINK \l "LCD_DATA"LCD_RS
LCD_EN
LCD_COMMAND_US
HYPERLINK \l "LCD_DATA_US"LCD_DATA_US
LCD_INIT_DELAY
LCD_RW

Overview

Supports Hitachi HD44780 LCD controller

Circuit Diagram

Example Code

```
// LCD...
#option LCD_DATA = PORTD.4
#option LCD_RS = PORTE.0
#option LCD_EN = PORTE.1
// import LCD library...
include "LCD.bas"
include "utils.bas"
// refresh speed...
const UpdateMS = 50
// initialise bit patterns...
// programmable characters are available that use codes $00 to $07.
// Create the bit patterns that make up the bars in the LCD's CGRAM.
// The vertical bars are made up of 8 identical bit patterns
const CGRAM(32) as byte = ($00,$00,$00,$00,$00,$00,$00,$00, // base bar
$10,$10,$10,$10,$10,$10,$10,$00, // 8 x %10000 = |
$14,$14,$14,$14,$14,$14,$14,$00, // 8 x %10100 = ||
$15,$15,$15,$15,$15,$15,$15,$00) // 8 x %10101 = |||
// output byte pRepValue times...
noinline sub Rep(pValue, pRepValue as byte)
dim Index as byte
Index = 0
while Index < pRepValue
LCD.Write(pValue)
inc(Index)
wend
end sub
// display the bar...
noinline sub Bargraph(pLine, pBarValue as byte)
const BASE_BAR = 0 // ASCII value of 0 bar (blank)
const FULL_BAR = 3 // ASCII value of ||| bar
const BAR_WIDTH = 16 // Max width in characters of bar
const MAX_BAR_COUNT = BAR_WIDTH * 3 // Max bar counts
dim NumberOfBars as byte
dim Balance as byte
dim BalanceChar as byte
NumberOfBars = pBarValue / 3
Balance = pBarValue mod 3
MoveCursor(pLine,1)
Rep(FULL_BAR,NumberOfBars)
Write(Balance)
Rep(BASE_BAR,BAR_WIDTH - (NumberOfBars + Min(Balance,1)))
end sub
// loop index
dim Index as byte
dim ValueA, ValueB,FadeA, FadeB as byte
// clear screen...
ADCON1 = $07 // PORTE as digital (LCD)
Write(CGRAM)
// display the bar
while true
for Index = 0 to 48
Bargraph(1,Index)
Bargraph(2,48 - Index)
delayms(UpdateMS)
next
for Index = 48 to 0 step -1
Bargraph(1,Index)
Bargraph(2,48 - Index)
delayms(UpdateMS)
next
```

Interface

sub Cls()
Clears the LCD display area

sub Command (pCommand **as byte**)

- *pCommand* - the command to be sent to the LCD. Valid arguments include cmdCGRAM, cmdDDRAM, cmdClear, cmdHome, cmdCursorOff, cmdCursorOn, cmdBlinkOn, cmdBlinkOff, cmdMoveLeft and cmdMoveRight

Issues a command to the LCD display

sub MoveCursor(pLine, pCol **as byte**)

- *pLine* - the LCD line
- *pCol* - the LCD column

Moves the LCD cursor to position co-ordinates Line, Column

compound sub Write (WriteItem)

- *WriteItem* - a string, char, byte or byte array

Writes a string, char, byte or byte array to the LCD display at the current cursor position.

compound sub WriteAt (pLine, pCol, WriteItem)

- *pLine* - the LCD line
- *pCol* - the LCD column
- *WriteItem* - a string, char, byte or byte array

Moves the LCD cursor to position co-ordinates Line, Column and then writes a string, char, byte or byte array to the LCD display.

Compile Options

#option LCD_DATA

The LCD_DATA option sets the port or port pin for the LCD data bus. If a port name is given, without a pin qualifier, then the library defaults to a 8 bit data bus. Specifying a port pin will force the LCD library into 4 bit data mode. Valid pin qualifiers include 0 and 4. If the LCD_DATA option is not used, it defaults to PORTB.4

#option LCD_RS
The LCD_RS option sets the LCD RS pin. If the LCD_RS option is not used, it defaults to PORTB.3

#option LCD_EN
The LCD_EN option sets the LCD EN pin. If the LCD_EN option is not used, it defaults to PORTB.2

#option LCD_COMMAND_US
The LCD_COMMAND_US option sets the delay value after a command write. Values can be ranged between 1 and 65535. If the LCD_COMMAND_US option is not used, it defaults to 2000.

#option LCD_DATA_US
The LCD_DATA_US option sets the delay value after a data write. Values can be ranged between 1 and 255. If the LCD_DATA_US option is not used, it defaults to 50.

#option LCD_INIT_DELAY
The LCD_INIT_DELAY option sets the delay (ms) before the module is initialised. Values can be ranged between 0 and 1000. If LCD_INIT_DELAY option is not used, it defaults to 100.

#option LCD_RW
The LCD_RW is used to defince an optional device busy flag, rather than using fixed delays

Manchester Library

Module and Filename

Manchester
Manchester.bas

Interface

Subroutines and Functions
function Encode (pValue **as byte**) **as word**
function Decode (pEncodedValue **as word**, **byref** pDecodedValue **as byte**) **as boolean**

Overview

Manchester library.

Example Code

```
// import libraries...
include "Manchester.bas"
include "USART.bas"
include "Convert.bas"
dim EncodedValue as word
dim DecodedValue as byte
SetBaudrate(br19200)
EncodedValue = Encode(42)
USART.Write("Encode : ", BinToStr(EncodedValue, 16), 13, 10)
if Decode(EncodedValue,DecodedValue) then
USART.Write("Decode : ", DecToStr(DecodedValue), 13, 10)
else
USART.Write("Error!", 13, 10)
endif
```

Interface

function Encode(pValue **as byte**) **as word**

- *pValue* - Number to encode

Takes a byte value and returns 16 bit (word) Manchester encoded number.

function Decode(pEncodedValue **as word**, **byref** pDecodedValue **as byte**) **as boolean**

- *pEncodedValue* - Number to decode
- *pDecodedValue* - Decoded number

Decodes a Manchester encoded number. Returns true if the number has been decoded successfully, false otherwise.

Math Library

Module and Filename

Math
Math.bas

Interface

Subroutines and Functions

function abs (pValue **as type**) **as byte**
function trunk (pValue **as float**) **as float**
function round (pValue **as float**) **as float**
function ceil (pValue **as float**) **as float**
function floor (pValue **as float**) **as float**
function fmod (x,y **as float**) **as float**
function modf (pValue **as float**, **byref** pIntegral **as float**) **as float**
function sqrt (pValue **as float**) **as float**
function cos (pValue **as float**) **as float**
function sin (pValue **as float**) **as float**
function tan (pValue **as float**) **as float**
function acos (pValue **as float**) **as float**
function asin (pValue **as float**) **as float**
function atan (pValue **as float**) **as float**
function exp (pValue **as float**) **as float**
function log (pValue **as float**) **as float**
function log10 (pValue **as float**) **as float**
function Pow ([pBase **as type**,] pExp **as byte**) **as longword**
function atan2 (y,x **as float**) **as float**
function cosh (x **as float**) **as float**
function sinh (x **as float**) **as float**
function tanh (x **as float**) **as float**
function frexp (x **as float**, **byref** exp **as shortint**) **as float**
function ldexp (value **as float**, exp **as shortint**) **as float**

Compile Options

MATH_ERROR

Overview

Floating point math library.

OneWire (OW) Library

Module and Filename

OW
OW.bas

Interface

Types

TOnSearch

Subroutines and Functions

sub SetPin (**byref** pPin **as bit**)
sub WriteBit (pValue **as bit**)
function ReadBit () **as bit**
sub WriteByte (pValue **as byte**)
function ReadByte () **as byte**
sub WriteArray (**byref** pArray() **as byte**, pMax **as byte** = $FF)
function ReadArray (**byref** pArray() **as byte**, pMax **as byte** = $FF) **as boolean**
function Reset () **as boolean**
sub WaitForHigh ()
function Search (pCommand **as byte**, pOnSearch **as** TOnSearch = 0) **as byte**
function Count () **as byte**

Variables

SearchID
SearchFamily
SearchAbort
SearchIgnore

Compile Options

OW_PIN

Overview

OneWire (OW) library.

Example Code

```
// import modules...
include "ow.bas"
include "convert.bas"
include "usart.bas"
// on find event handler...
event OnFind()
dim Index as byte
USART.Write("FAMILY $", HexToStr(SearchFamily,2))
USART.Write(" ($",HexToStr(SearchID(7),2),")")
USART.Write(" ($")
Index = 6
repeat
USART.Write(HexToStr(SearchID(Index),2))
dec(Index)
until Index = 0
USART.Write(")",13,10)
end event
// working variables...
dim DeviceCount as byte
// program start...
SetBaudrate(br115200)
SetPin(PORTC.0)
DeviceCount = Search(owSearchROM, OnFind)
USART.Write(DecToStr(DeviceCount), " device(s) found", 13, 10)
```

A complete module, showing how a DS18B20 OW device can be implemented using the OW library, can be found here.

Interface

TOnSearch = **event**()

Search event handler type.

sub SetPin(**byref** pPin **as bit**)

- *pPin* - The OW port pin

Used to set the OW port pin at runtime. To fix the OW at compile time, see OW_PIN.

sub WriteBit(pValue **as bit**)

- *pValue* - Bit value to send

Send a bit value to a OW device.

function ReadBit() **as bit**

Receive a bit value from a OW device.

sub WriteByte(pValue **as byte**)

- *pValue* - Byte to send.

Send a byte value to a OW device.

function ReadByte() **as byte**

Receive a byte value from a OW device.

sub WriteArray(**byref** pArray() **as byte**, pMax **as byte** = $FF)

- *pArray* - Array of byte data
- *pMax* - An optional parameter which specifies the number of bytes to send.

Send a byte array to a OW device.

function ReadArray(**byref** pArray() **as byte**, pMax **as byte** = $FF) **as boolean**

- *pArray* - Array of byte to receive incoming data
- *pMax* - An optional parameter which specifies the number of bytes to receive

Receive a byte array from a OW device. The function will return true if the last CRC byte resolves to 0, else it returns false.

function Reset() **as boolean**
Resets the OW data pin. The function returns true if a device is detected, false otherwise.

sub WaitForHigh()

Waits for the OW data pin to go high.

function Search(pCommand **as byte**, pOnSearch **as** TOnSearch = 0) **as byte**

- *pCommand* - The command to send to the OW device. The modules supports the generic commands owReadROM, owSkipROM, owMatchROM and owSearchROM. You should refer to your particular device datasheet to determine what additional commands are supported.
- *pOnSearch* - An optional event handler.

Searches for devices connected to the OW bus, based on the *pCommand* value, normally owSearchROM. The function returns the total number of devices found. The optional event handler *pOnSearch* will be triggered each time a device is found. Setting SearchAbort to true (inside the event handler) will abort the search. Setting SearchIgnore to true (inside the event handler) will prevent the return value (number of devices) from being incremented.

function Count() **as byte**
Returns the number of OW devices connected.

SearchID(8) **as byte**
The OW device ID, which is initialised before a *Search* event handler is triggered.

SearchFamily **as byte**

The OW device family ID, which is initialised before a *Search* event handler is triggered.

SearchAbort **as boolean**

Setting SearchAbort to true, inside a *Search* event handler, will abort the search

SearchIgnore **as boolean**
Setting SearchIgnore to true, inside a *Search* event handler, will will prevent the return value (number of devices) from being incremented.

#option OW_PIN
Sets the OW data pin at compile time. If no value is given, the OW must be initialised at runtime with a call to *SetPin*.

OneWire (OW) Library

Module and Filename

OW
OW.bas

Interface

Types

TOnSearch

Subroutines and Functions

sub SetPin (**byref** pPin **as bit**)
sub WriteBit (pValue **as bit**)
function ReadBit () **as bit**
sub WriteByte (pValue **as byte**)
function ReadByte () **as byte**
sub WriteArray (**byref** pArray() **as byte**, pMax **as byte** = $FF)
function ReadArray (**byref** pArray() **as byte**, pMax **as byte** = $FF) **as boolean**
function Reset () **as boolean**
sub WaitForHigh ()
function Search (pCommand **as byte**, pOnSearch **as** TOnSearch = 0) **as byte**
function Count () **as byte**

Variables

SearchID
SearchFamily
SearchAbort
SearchIgnore

Compile Options

OW_PIN

Overview

OneWire (OW) library.

Example Code

```
// import modules...
include "ow.bas"
include "convert.bas"
include "usart.bas"
// on find event handler...
event OnFind()
dim Index as byte
USART.Write("FAMILY $", HexToStr(SearchFamily,2))
USART.Write(" ($",HexToStr(SearchID(7),2),")")
USART.Write(" ($")
Index = 6
repeat
USART.Write(HexToStr(SearchID(Index),2))
dec(Index)
until Index = 0
USART.Write(")",13,10)
end event
// working variables...
dim DeviceCount as byte
// program start...
SetBaudrate(br115200)
SetPin(PORTC.0)
DeviceCount = Search(owSearchROM, OnFind)
USART.Write(DecToStr(DeviceCount), " device(s) found", 13, 10)
```

A complete module, showing how a DS18B20 OW device can be implemented using the OW library, can be found here.

Interface

TOnSearch = **event**()

Search event handler type.

sub SetPin(**byref** pPin **as bit**)

- *pPin* - The OW port pin

Used to set the OW port pin at runtime. To fix the OW at compile time, see OW_PIN.

sub WriteBit(pValue **as bit**)

- *pValue* - Bit value to send

Send a bit value to a OW device.

function ReadBit() **as bit**

Receive a bit value from a OW device.

sub WriteByte(pValue **as byte**)

- *pValue* - Byte to send.

Send a byte value to a OW device.

function ReadByte() **as byte**

Receive a byte value from a OW device.

sub WriteArray(**byref** pArray() **as byte**, pMax **as byte** = $FF)

- *pArray* - Array of byte data
- *pMax* - An optional parameter which specifies the number of bytes to send.

Send a byte array to a OW device.

function ReadArray(**byref** pArray() **as byte**, pMax **as byte** = $FF) **as boolean**

- *pArray* - Array of byte to receive incoming data
- *pMax* - An optional parameter which specifies the number of bytes to receive

Receive a byte array from a OW device. The function will return true if the last CRC byte resolves to 0, else it returns false.

function Reset() **as boolean**
Resets the OW data pin. The function returns true if a device is detected, false otherwise.

sub WaitForHigh()

Waits for the OW data pin to go high.

function Search(pCommand **as byte**, pOnSearch **as** TOnSearch = 0) **as byte**

- *pCommand* - The command to send to the OW device. The modules supports the generic commands owReadROM, owSkipROM, owMatchROM and owSearchROM. You should refer to your particular device datasheet to determine what additional commands are supported.
- *pOnSearch* - An optional event handler.

Searches for devices connected to the OW bus, based on the *pCommand* value, normally owSearchROM. The function returns the total number of devices found. The optional event handler *pOnSearch* will be triggered each time a device is found. Setting SearchAbort to true (inside the event handler) will abort the search. Setting SearchIgnore to true (inside the event handler) will prevent the return value (number of devices) from being incremented.

function Count() **as byte**
Returns the number of OW devices connected.

SearchID(8) **as byte**
The OW device ID, which is initialised before a *Search* event handler is triggered.

SearchFamily **as byte**

The OW device family ID, which is initialised before a *Search* event handler is triggered.

SearchAbort **as boolean**

Setting SearchAbort to true, inside a *Search* event handler, will abort the search

SearchIgnore **as boolean**
Setting SearchIgnore to true, inside a *Search* event handler, will will prevent the return value (number of devices) from being incremented.

#option OW_PIN
Sets the OW data pin at compile time. If no value is given, the OW must be initialised at runtime with a call to *SetPin*.

Shift Library

Module and Filename

Shift
Shift.bas

Interface

Subroutines and Functions

sub SetInput (**byref** pDataPin **as bit**)
sub SetOutput (**byref** pDataPin **as bit**)
sub SetClock (**byref** pClockPin **as bit**,pIdleHigh **as boolean** = **false**)
sub Out (pMode **as byte**,pData **as** TType, pShift **as byte**)
function In (pMode **as byte**, pShift **as byte**) **as** TType

Compile Options

SHIFT_MAX
SHIFT_CLOCK

Overview

Shift library.

Example Code

```
// Read temperature from DALLAS 1620
// LCD options...
#option LCD_DATA = PORTD.0
#option LCD_RS = PORTE.0
#option LCD_EN = PORTE.1
// import LCD and shift libraries...
include "LCD.bas"
include "shift.bas"
include "convert.bas"
// data, clock and reset pins...
dim DataPin as PORTB.0
dim ClockPin as PORTB.1
dim ResetPin as PORTB.2
// start conversion...
sub StartConversion()
ResetPin = 1
Shift.Out(LSB_FIRST, $EE, 8)
ResetPin = 0
end sub
// read temperature...
function ReadTemperature() as word
ResetPin = 1
Shift.Out(LSB_FIRST, $AA, 8)
Result = Shift.In(LSB_PRE, 9)
ResetPin = 0
end function
// display the temperature on LCD...
sub DisplayTemperature(pValue as word)
LCD.MoveCursor(1,1)
LCD.Write(DecToStr(pValue >> 1), ".", DecToStr(pValue.0 * 5), $DF, "C")
end sub
// LCD configured for FLASH Lab
ADCON1 = $07 // PORTE as digital
Cls
// set clock and reset pin to output...
Shift.SetInput(DataPin)
Shift.SetOutput(DataPin)
Shift.SetClock(ClockPin)
output(ResetPin)
// main program loop...
StartConversion
while true
DisplayTemperature(ReadTemperature)
delayms(500)
wend
```

Interface

sub SetInput(**byref** pDataPin **as bit**)

- *pDataPin* - Port pin

Sets the shift input pin.

sub SetOutput(**byref** pDataPin **as bit**)

- *pDataPin* - Port pin

Sets the shift output pin.

sub SetClock(**byref** pClockPin **as bit**,pIdleHigh **as boolean** = **false**)

- *pClockPin* - Port pin
- *pIdleHigh* - Optional clock idle state

Sets the shift clock pin. By default, the clock will idle low. Setting the optional pIdleHigh flag to true will force the clock to idle high. See also SHIFT_CLOCK.

sub Out(pMode **as byte**,pData **as** TType, pShift **as byte**)

- *pMode* - Output mode. Can be LSB_FIRST or MSB_FIRST.
- *pData* - Data to send. TType will depend on the compiler option SHIFT_MAX.
- *pShift* - Number of bits to shift.

Shift data output routine. The parameter *pMode* indicates:

LSB_FIRST	LSB sent first
MSB_FIRST	MSB sent first

function In(pMode **as byte**, pShift **as byte**) **as** TType

- *pMode* - Input mode. Can be MSB_PRE, MSB_POST, LSB_PRE and LSB_POST.
- *pShift* - Number of bits to receive.

The shift in routine returns data of TType, which will depend on the option SHIFT_MAX. The parameter *pMode* indicates

MSB_PRE	MSB first, sample before clock
MSB_POST	MSB first, sample after clock
LSB_PRE	LSB first, sample before clock
LSB_POST	LSB first, sample after clock

#option SHIFT_MAX

The maximum shift size (in or out) can be 8, 16 or 32 bits. Using a smaller value will reduce the overall code footprint. By default, it is set to 16 bits.

#option SHIFT_CLOCK

The shift clock delay, in microseconds. Can be 1 or 2 us. By default, it is set to 1 us.

Software I2C Library

Modules and Filenames

SI2C
SI2C.bas

Interface

Subroutines and Functions

sub Initialize ()
sub Start ()
sub Stop ()
sub Restart ()
sub Acknowledge (pAck **as bit** = I2C_ACKNOWLEDGE)
function ReadByte () **as byte**
sub WriteByte (pData **as byte**)

Options

I2C_SCL
I2C_SDA

Overview

Low level software I2C interface library.

Example Code

```
// import libraries...
include "SI2C.bas"
include "usart.bas"
// target 24LC128 I2C EEPROM device...
const I2C_EEPROM = $A0
// local variables...
dim
Value as byte,
Address as word
// program start...
Address = 0
SI2C.Initialize
// write some data...
SI2C.Start
SI2C.WriteByte(I2C_EEPROM)
SI2C.WriteByte(Address.Byte1)
SI2C.WriteByte(Address.Byte0)
SI2C.WriteByte("A")
SI2C.Stop
// allow external EEPROM to write data...
delayms(10)
// read the data back...
SI2C.Start
SI2C.WriteByte(I2C_EEPROM)
SI2C.WriteByte(Address.Byte1)
```

```
SI2C.WriteByte(Address.Byte0)
SI2C.Restart
SI2C.WriteByte(I2C_EEPROM + 1)
Value = SI2C.ReadByte
SI2C.Acknowledge(I2C_NOT_ACKNOWLEDGE)
SI2C.Stop
// output the result
USART.SetBaudrate(br19200)
USART.Write("Value = ", Value, 13, 10)
```

Interface

sub Initialize()

Initialise the I2C Bus.

sub Start()
Transmit a start condition on the I2C bus.

sub Stop()
Transmit a stop condition on the I2C bus.

sub Restart()
Transmit a restart condition on the I2C bus.

sub Acknowledge(pAck **as bit** = I2C_ACKNOWLEDGE)

- *pAck* - Bus acknowledge. Can be I2C_ACKNOWLEDGE or I2C_NOT_ACKNOWLEDGE

Transmit an acknowledge on the I2C bus.

function ReadByte() **as byte**
Read a byte from the I2C bus.

sub WriteByte(pData **as byte**)

- *pData* - Data to write

Write a byte to the I2C bus

#option I2C_SCL
Software I2C clock pin. By default, it is set to PORTC.3.

#option I2C_SDA

Software I2C data pin. By default, it is set to PORTC.4.

Software SPI Library

Modules and Filenames

SSPI
SSPI.bas

Interface

Subroutines and Functions
sub Initialize ()
sub SetClock (pIdle **as byte**)
function ReadByte() **as byte**
sub WriteByte(pData **as byte**)
Options

SSPI_SCK
SSPI_SDI
SSPI_SDO

Overview
Low level software SPI library.

Interface
sub Initialize ()

Initializes the software SPI bus for master mode.

sub SetClock(pIdle **as byte**)

- *pIdle* - Clock idle mode. Can be spiIdleHigh, spiIdleLow

Set the SPI idle state and data transmission clock edge.

function ReadByte() **as byte**

Read a byte from the SPI bus.

sub WriteByte(pData **as byte**)

- *pData* - Data to write.

Write a byte to the SPI bus.

#option SSPI_SCK
Software SPI clock pin. Default is PORTC.3.

#option SSPI_SDI
Software SPI data in pin. Default is PORTC.4

#option SSPI_SDO
Software SPI data out pin. Default is PORTC.5

Software UART Library

Modules and Filenames

UART
SUART.bas

Interface

Subroutines and Functions

sub SetTX (**byref** pPin **as bit**)
sub SetRX (**byref** pPin **as bit**)
sub SetBaudrate (pBaudrate **as word**)
sub SetMode (pMode **as byte**)
function ReadByte () **as byte**
function ReadBoolean () **as boolean**
function ReadWord () **as word**
function ReadLongWord () **as longword**
function ReadFloat () **as float**
sub WriteByte (pValue **as byte**)
sub WriteBoolean (pValue **as boolean**)
sub WriteWord (pValue **as word**)
sub WriteLongWord (pValue **as longword**)
sub WriteFloat (pValue **as float**)
compound sub Read (**byref** pReadItem)
compound sub Write (pWriteItem)

Variables

ReadTerminator **as char**
Pacing **as word**

Overview

Software UART library.

Interface

sub SetTX(**byref** pPin **as bit**)

- *pPin* - Port pin.

Set the software UART transmit pin.

sub SetRX(**byref** pPin **as bit**)

- *pPin* - Port pin.

Set the software UART receive pin.

sub SetBaudrate(pBaudrate **as word**)

- *pBaudrate* - UART baudrate. Can be sbr300, sbr600, sbr1200, sbr2400, sbr4800, sbr9600, sbr19200, sbr38400 or sbr57600.

Set the software UART baudrate. Note that with higher baudrates, your

microcontroller needs to be clocked at quite a high speed. Generally, sbr300 to sbr9600 should give good results. For high baudrates, you should really look at using the microcontrollers onboard USART.

sub SetMode(pMode **as byte**)

- *pMode* - UART Mode. Can be umTrue, umInverted, umOpen, umOpenTrue, umOpenInverted.

Set the UART mode. By default, mode is set to umInverted.

function ReadByte() **as byte**
Read a single byte.

function ReadBoolean() **as boolean**
Read a boolean.

function ReadWord() **as word**
Read a word or integer.

function ReadLongWord() **as longword**
Read a long word or long integer.

function ReadFloat() **as float**
Read a floating point number.

sub WriteByte(pValue **as byte**)

- *pValue* - Data to write.

Write a single byte.

sub WriteBoolean(pValue **as boolean**)

- *pValue* - Data to write.

Write a boolean.

sub WriteWord(pValue **as word**)

- *pValue* - Data to write.

Write a word or integer.

sub WriteLongWord(pValue **as longword**)

- *pValue* - Data to write.

Write a long word or long integer.

sub WriteFloat(pValue **as float**)

- *pValue* - Data to write.

Write a floating point number.

compound sub Read(**byref** pReadItem)

- *pReadItem* - Data to read.

Read multiple data items. Valid argument types include boolean, char, string, byte, shortint, word, integer, longword, longint and floating point. See also ReadTerminator.

compound sub Write (pWriteItem)

- *pWriteItem* - Data to write.

Write multiple items. Valid argument types include boolean, char, string, byte, shortint, word, integer, longword, longint and floating point.

ReadTerminator **as char**
Set the Read terminator character. By default, it is set to NULL (0).

Pacing **as word**
Sets the character transmission delay in microseconds (us). By default, it is set to 0.

SPI Library

Modules and Filenames

SPI, SPI2
SPI.bas, SPI2.bas

Interface

Subroutines and Functions

sub SetAsMaster (pMode **as byte** = spiOscDiv4)
sub SetAsSlave (pMode **as byte** = spiSlaveSSDisabled)
sub SetSample (pSample **as byte**)
sub SetClock (pIdle **as byte**, pEdge **as byte**)
function ReadByte() **as byte**
sub WriteByte(pData **as byte**)

Variables

Enabled **as boolean**
Overflow **as boolean**
WriteCollision **as boolean**

Overview

Low level SPI library.

Interface

sub SetAsMaster(pMode **as byte** = spiOscDiv4)

- *pMode* - SPI master mode. Can be spiOscDiv4, spiOscDiv16,spiOscDiv64 or spiOscTimer2.

Initializes the SPI bus for master mode.

sub SetAsSlave(pMode **as byte** = spiSlaveSSDisabled)

- *pMode* - SPI slave. Can be spiSlaveSSEnabled or spiSlaveSSDisabled.

Initializes the SPI bus for slave mode.

sub SetSample(pSample **as byte**)

- *pSample* - Data sample. Can be spiSampleEnd or spiSampleMiddle.

Sets when the SPI input data in sampled. For slave mode, pSample should always be spiSampleMiddle.

sub SetClock(pIdle **as byte**, pEdge **as byte**)

- *pIdle* - Clock idle mode. Can be spiIdleHigh, spiIdleLow
- *pEdge* - Transmission clock edge. Can be spiRisingEdge or spiFallingEdge.

Set the SPI idle state and data transmission clock edge.

function ReadByte() **as byte**

Read a byte from the SPI bus.

sub WriteByte(pData **as byte**)

- *pData* - Data to write.

Write a byte to the SPI bus.

Enabled **as boolean**
Setting to true will enable the SPI bus, setting to false will disable it. Boolean equivalent of SSPEN.

Overflow **as boolean**
Checks the SPI overflow status bit. Boolean equivalent of SSPOV.

WriteCollision **as boolean**
Checks the SPI write collision status bit. Boolean equivalent of WCOL.

String Library

Modules and Filenames

Str
String.bas

Interface

Subroutines and Functions
function Length (pStr **as string**) **as byte**
function Copy (pStr **as string**, pStart, pLength **as byte**) **as string**
function Uppercase (pStr **as string**) **as string**
function Lowercase (pStr **as string**) **as string**
function Position (pSubStr,pStr **as string**) **as integer**
function Mid (pStr **as string**,pStart **as byte**,pLength **as byte** = $FF) **as string**
function Left (pStr **as string**,pLength **as byte**) **as string**
function Right (pStr **as string**,pLength **as byte**) **as string**
function Delete (pSub, pStr **as string**) **as string**
sub Insert (pStr **as string**,**byref** pDest **as string**,pIndex **as byte**)
function TrimLeft (pStr **as string**) **as string**
function TrimRight (pStr **as string**) **as string**
function Trim (pStr **as string**) **as string**
function Compare (pStrA, pStrB **as string**) **as shortint**

Overview
String library

Interface

function Length(pStr **as string**) **as byte**

- *pStr* - Input string.

Returns the length of a string, excluding the null terminator.

function Copy(pStr **as string**, pStart, pLength **as byte**) **as string**

- *pStr* - Input string.
- *pStart* - Starting index.
- *pLength* - Number of characters to copy.

Returns a substring of *pStr*, starting at *pStart* for *pLength* characters.

function Uppercase(pStr **as string**) **as string**

- *pStr* - Input string.

Returns a copy of *pStr* in uppercase.

function Lowercase(pStr **as string**) **as string**

- *pStr* - Input string.

Returns a copy of *pStr* in lowercase.

function Position(pSubStr,pStr **as string**) **as integer**

- *pSubStr* - Sub string to find.
- *pStr* - Input string.

Returns the index value of the first character in a specified substring that occurs in a given string. If no match is found, the function returns - 1.

function Mid(pStr **as string**,pStart **as byte**,pLength **as byte** = $FF) **as string**

- *pStr* - Input string.
- *pStart* - Starting index.
- *pLength* - Optional number of characters to copy.

Returns a substring of *pStr*, starting at *pStart* for *pLength* characters.

function Left(pStr **as string**,pLength **as byte**) **as string**

- *pStr* - Input string.
- *pLength* - Number of characters to copy.

Returns a string containing a specified number of characters from the left side of a string.

function Right(pStr **as string**,pLength **as byte**) **as string**

- *pStr* - Input string.
- *pLength* - Number of characters to copy.

Returns a string containing a specified number of characters from the right side of a string.

function Delete(pSub, pStr **as string**) **as string**

- *pSub* - Sub string.
- *pStr* - Input string.

Deletes and returns a copy of a string with the first *pSub* string removed.

sub Insert(pStr **as string**,**byref** pDest **as string**,pIndex **as byte**)

- *pStr* - Input string.
- *pDest* - Output string.
- *pIndex* - Insertion index.

Inserts a substring into *pDest* at location *pIndex*.

function TrimLeft(pStr **as string**) **as string**

- *pStr* - Input string.

Returns a copy of a string with all spaces to the left of *pStr* removed.

function TrimRight(pStr **as string**) **as string**

- *pStr* - Input string.

Returns a copy of a string with all spaces to the right of *pStr* removed.

function Trim(pStr **as string**) **as string**

- *pStr* - Input string.

Returns a copy of a string with all spaces to the left and right of *pStr* removed.

function Compare(pStrA, pStrB **as string**) **as shortint**

- *pStrA* - Input string A.
- *pStrB* - Input string B.

Compares two strings. Returns a negative number if string A is less than string B. Returns zero if string A is equal to string B. Returns a positive integer if string A is greater than stringB.

USART Library

Modules and Filenames

USART, USART2
USART.bas, USART2.bas

Interface

Subroutines and Functions

sub SetBaudrate (pSPBRG **as byte** = br19200)
sub ClearOverrun ()
function ReadByte () **as byte**
function ReadBoolean () **as boolean**
function ReadWord () **as word**
function ReadLongWord () **as longword**
function ReadFloat () **as float**
sub WriteByte (pValue **as byte**)
sub WriteBoolean (pValue **as boolean**)
sub WriteWord (pValue **as word**)
sub WriteLongWord (pValue **as longword**)
sub WriteFloat (pValue **as float**)
compound sub Read (**byref** pReadItem)
compound sub Write (pWriteItem)
function WaitFor (pValue **as byte**) **as boolean**
function WaitForTimeout (pValue **as byte**, pTimeout **as word**) **as boolean**
sub WaitForCount (**byref** pArray() **as byte**, pCount **as word**)
sub WaitForStr (pStr **as string**)
sub WaitForStrCount (**byref** pStr **as string**, pCount **as word**)
function WaitForStrTimeout (pStr **as string**, pTimeout **as word**) **as boolean**
function DataAvailableTimeout (pTimeout **as word**) **as boolean**
sub Rep (pValue, pAmount **as byte**)
sub Skip (pAmount **as byte**)

Variables

ReadTerminator **as char**
DataAvailable **as boolean**
ReadyToSend **as boolean**
ContinousReceive **as boolean**
Overrun **as boolean**
FrameError **as boolean**
RCIEnable **as boolean**
TXIEnable **as boolean**
RCIPHigh **as boolean**
TXIPHigh **as boolean**

Compile Options

USART_BRGH
USART_BRGH16

Overview
Hardware USART library.

Interface

sub SetBaudrate(pBaudrate **as word**)

- *pBaudrate* - USART baudrate. Can be br300, br600, br1200, br2400, br4800, br9600, br19200, br38400, br57600 or br115200.

Set the USART baudrate.

sub ClearOverrun()
Clear overrun. You can check if a USART overrun has occurred using Overrun boolean flag.

function ReadByte() **as byte**
Read a single byte. This is a blocking call, which means it will not return until data is received. To see if data is available, use the DataAvailable boolean flag.

function ReadBoolean() **as boolean**
Read a boolean. This is a blocking call, which means it will not return until data is received. To see if data is available, use the DataAvailable boolean flag.

function ReadWord() **as word**
Read a word or integer. This is a blocking call, which means it will not return until data is received. To see if data is available, use the DataAvailable boolean flag.

function ReadLongWord() **as longword**
Read a long word or long integer. This is a blocking call, which means it will not return until data is received. To see if data is available, use the DataAvailable boolean flag.

function ReadFloat() **as float**
Read a floating point number. This is a blocking call, which means it will not return until data is received. To see if data is available, use the DataAvailable boolean flag.

sub WriteByte(pValue **as byte**)

- *pValue* - Data to write.

Write a single byte.

sub WriteBoolean(pValue **as boolean**)

- *pValue* - Data to write.

Write a boolean.

sub WriteWord(pValue **as word**)

- *pValue* - Data to write.

Write a word or integer.

sub WriteLongWord(pValue **as longword**)

- *pValue* - Data to write.

Write a long word or long integer.

sub WriteFloat(pValue **as float**)

- *pValue* - Data to write.

Write a floating point number.

compound sub Read(**byref** pReadItem)

- *pReadItem* - Data to read.

Read multiple data items. Valid argument types include boolean, char, string, byte, shortint, word, integer, longword, longint and floating point. See also ReadTerminator.

compound sub Write (pWriteItem)

- *pWriteItem* - Data to write.

Write multiple items. Valid argument types include boolean, char, string, byte, shortint, word, integer, longword, longint and floating point.

function WaitFor(pValue **as byte**) **as boolean**

- *pValue* - Data to wait for.

A blocking call which will wait for data to be received. It returns true if *pValue* matches the data byte received, false otherwise.

function WaitForTimeout(pValue **as byte**, pTimeout **as word**) **as boolean**

- *pValue* - Data to wait for.
- *pTimeout* - Timeout value in milliseconds.

A blocking call which will wait for data to be received. It returns true if *pValue* matches the data byte received, false otherwise. The function will also return false if the timeout interval has been exceeded.

sub WaitForCount(**byref** pArray() **as byte**, pCount **as word**)

- *pArray* - Array of byte to receive data.
- *pCount* - Number of bytes to receive.

A blocking call which will wait for *pCount* data byte to be received.

sub WaitForStr(pStr **as string**)

- *pStr* - String to wait for.

A blocking call which will wait for *pStr* to be received. The sub will not return until the input string matches the string data received, See also *ReadTerminator*.

sub WaitForStrCount(**byref** pStr **as string**, pCount **as word**)

- *pStr* - String variable to receive data.
- *pCount* - Number of characters to receive.

A blocking call which will wait for *pCount* data characters to be received.

function WaitForStrTimeout(pStr **as string**, pTimeout **as word**) **as boolean**

- *pStr* - String to wait for.
- *pTimeout* - Timeout value in milliseconds.

A blocking call which will wait for *pStr* to be received. The function will return false if the timeout interval has been exceeded.

function DataAvailableTimeout(pTimeout **as word**) **as boolean**

- *pTimeout* - Timeout value in milliseconds.

A blocking call which will wait for a specified timeout value before returning. Returns true is data received, false is it timesout.

sub Rep(pValue, pAmount **as byte**)

- *pValue* - Data to transmit.
- *pAmount* - Number of times to repeat transmission.

Sends *pValue* data *pAmount* times.

sub Skip(pAmount **as byte**)

- *pAmount* - Number of times to skip incoming data.

A blocking call which will wait until *pAmount* data byte have been received.

ReadTerminator **as char**
Set the Read terminator character. By default, it is set to NULL (0).

DataAvailable **as byte**
Indicates if the USART has received data. Boolean equivalent to RCIF. See the microchip datasheet for more information.

ReadyToSend **as byte**
Indicates if the USART is able to send data. Boolean equivalent to TXIF. See the microchip datasheet for more information.

ContinousReceive **as byte**
Indicates if the USART is set for continuous receive. Boolean equivalent to CREN. See

the microchip datasheet for more information.

Overrun **as byte**
Indicates if a buffer overrun error has occurred. Use ClearOverrun to correctly clear this flag. Boolean equivalent to OERR. See the microchip datasheet for more information.

FrameError **as byte**
Indicates if a frame error has occurred. Boolean equivalent to FERR. See the microchip datasheet for more information.

RCIEnable **as byte**
Enables or disables RX interrupt enable. Boolean equivalent to RCIE. See the microchip datasheet for more information.

TXIEnable **as byte**
Enables or disables TX interrupt enable. Boolean equivalent to TXIE. See the microchip datasheet for more information.

RCIPHigh **as byte**
Setting this flag to true will set the RX interrupt priority to high. Set to false for RX priority low. Boolean equivalent to RCIP. See the microchip datasheet for more information.

TXIPHigh **as byte**
Setting this flag to true will set the TX interrupt priority to high. Set to false for TX priority low. Boolean equivalent to TXIP. See the microchip datasheet for more information.

#option USART_BRGH
Enables the High Speed Baud Rate select bit. Enabling BRGH can reduce the baudrate error for serial communications, even for low baudrates. However, if a slow baudrate is selected when using a high clock frequency you may need to disable this option. You should refer to the MCU datasheet for more information regarding BRGH. For USART2, the option is USART2_BRGH. By default, this option is set to true.

#option USART_BRGH16
Links in code to support 16 bit baud rate generator. You should read the device datasheet to ensure that BRGH is supported. For USART2, the option is USART2_BRGH16. By default, this option is set to false.

USB HID Library

Please note that this library is not available in the Special Edition (SE) version of the Swordfish compiler.

Module and Filename

HID
USBHID.bas

Interface

Subroutines and Functions

function Attached() **as boolean**
sub Service()
sub EnableISR()
sub DisableISR()
function DataAvailable() **as boolean**
sub ReadReport()
sub WriteReport()
sub ReadArray(**byref** pBuffer() **as byte**, pCount **as word** = 0)
sub WriteArray(**byref** pBuffer() **as byte**, pCount **as word** = 0)

Constants

BufferRAM
TXReportRAM
RXReportRAM

Variables

Buffer(256) **as byte**
TXReport(HIDReportBytesIn) **as byte**
RXReport(HIDReportBytesOut) **as byte**

Compile Options

USB_DESCRIPTOR
USB_SERVICE
USB_SERVICE_PRIORITY

Overview

The Human Interface Device (HID) module enables you to send structured data packets to and from the host computer. HID has the virtue of being very user friendly and plug and play. No device drivers are required on the PC side. When you plug in your USB device, the OS will detect and configure the HID device automatically. However, it does require a little more work from the programmer in terms of providing an application for the PC that will interface with the HID device. If you want to make the programming task a little easier on the PC, you might want to look at using the compiler CDC module instead.

Alternatively, you might want to try using the Swordfish EasyHID Plugin. The EasyHID Wizard can be used to automatically generate two template programs. The first program (the host software) is used on your PC. The second program (the device software) is used on your microcontroller. If the plugin wasn't supplied with your compiler installation, you can download from here

- Download EasyHID

Additional Information

- USB Connection Schematic
- Sample Program
- USB Descriptors and Options
- Vendor and Product ID (VID and PID)

Example Code

```
// device and clock...
device = 18F4550
clock = 48
// 20Mhz crystal, 48Mhz internal (FS USB)
config
PLLDIV = 5,
CPUDIV = OSC1_PLL2,
USBDIV = 2,
FOSC = HSPLL_HS,
VREGEN = ON
// import modules...
include "usbhid.bas"
// TX report...
structure TTXReport
Time as word
Message as string
end structure
dim TXReport as TTXReport absolute TXReportRAM
// RX report...
structure TRXReport
LED0 as bit
LED1 as bit
end structure
dim RXReport as TRXReport absolute RXReportRAM
// alias port pins to LEDs...
dim
LED0 as PORTB.0,
LED1 as PORTB.1
// initialise...
TXReport.Time = 0
low(LED0)
low(LED1)
// connect to USB...
repeat
until Attached
// main program loop...
while true
// if we have data, set port values, update message
// and then reset time counter...
if DataAvailable then
ReadReport
LED0 = RXReport.LED0
```

```
LED1 = RXReport.LED1
TXReport.Message = "PORT CHANGED"
WriteReport
TXReport.Time = 0
delayms(100)
// no data, set waiting message...
else
TXReport.Message = "WAITING..."
WriteReport
inc(TXReport.Time)
endif
wend
```

You can download a test HID executable which will interface with the above program from here.

Interface

function Attached() **as boolean**
Returns true if the USB bus is in a configured state, false otherwise.

sub Service()

USB is a polled protocol which needs to be serviced every 1 millisecond or so. This is to ensure the connection between the microcontroller and PC is maintained and any transactions are handled. Unless you disable this option using USB_SERVICE = false, you should never have to call Service directly().

Polling service manually just means inserting Service() in your code at regular intervals. For example,
// wait for connection...
repeat
Service
until Attached

sub EnableISR()
sub DisableISR()

If the USB_SERVICE is enabled (which it is by default) then the module will service the USB connection using interrupts. However, you may have a critical section of code that should not be disturbed. For example, a software based serial read or write may will have exact timing requirements. You don't want an interrupt firing during this time, else these timing will be disturbed and the routines may fail. You can temporarily disable interrupt servicing by calling DisableISR(). When you have finished, enable the interrupt by calling EnableISR().
Remember - if you have disabled the polling interrupt and your routines are likely to exceed 1 millisecond in duration, you need to manually service the USB connection.

function DataAvailable() **as boolean**
Indicates if any data has been received.

sub ReadReport()

Read and load the RXReport buffer with data. The RXReport buffer is dimensioned to the size of HIDReportBytesOut. You can read RXReport like any other array. However, ReadReport() is far more useful when you want to use more structured data. The first thing you need to do is declare a structure which represents the format of the data to be received. For example,

```
structure TRXReport
Message as string
DataCount as byte
Data(10) as byte
end structure
The first field is a string, followed by a DataCount and DataArray. We
now declare a variable of type TRXReport, like this
dim ReportIn as TRXReport absolute RXReportRAM
Notice the absolute keyword followed by RXReportRAM. This will force the
compiler to overlay the structure at the reports RX data buffer
location. Now we can access the data like any other variable. For
example,
if DataAvailable then
ReadReport
LCD.Write(1,1,ReportIn.Message)
Index = 0
while Index < ReportIn.DataCount
LCD.Write(2,1,DecToStr(ReportIn.Data(Index)),"  ")
inc(Index)
delayms(500)
wend
Endif
```

Please note that this routine is only available for devices that support more than 256 of dual port USB RAM. For example, the 18F2455, 18F2550, 18F4455 and 18F4550

sub WriteReport()

Write the contents of the TXReport buffer to the USB bus. The TXReport buffer is dimensioned to the size of HIDReportBytesIn. You can write to TXReport like any other array. However, WriteReport() is far more useful when you want to use more structured data. The first thing you need to do is declare a structure which represents the format of the data to be transmitted. For example,

```
structure TTXReport
Message as string
Sample as word
end structure
The first field is a string, followed by a word sized sample. We now
declare a variable of type TTXReport, like this
dim ReportOut as TTXReport absolute TXReportRAM
Notice the absolute keyword followed by TXReportRAM. This will force the
compiler to overlay the structure at the reports TX data buffer
location. Now we can access the data like any other variable. For
example,
```

```
ReportOut.Message = "Hello"
ReportOut.Sample = $1234
WriteReport
```

Please note that this routine is only available for devices that support more than 256 of dual port USB RAM. For example, the 18F2455, 18F2550, 18F4455 and 18F4550

sub ReadArray(**byref** pBuffer() **as byte**, pCount **as word** = 0)
Read an array of bytes - pCount is an optional number of bytes to read. If pCount is not given, the size of the array is used.

sub WriteArray(**byref** pBuffer() **as byte**, pCount **as word** = 0)
Write an array of bytes - pCount is an optional number of byte to write. If pCount is not given, the array size is used. Bytes are transferred in HIDReportBytesIn sized packets. If you pass an array which is only dimensioned 4 bytes, for example, **dim** Array(4) **as byte,** then a HIDReportBytesIn sized data packet is still transferred. This isn't a problem, as the host application should know what to do with the data anyway.

BufferRAM
A constant which gives the RAM location of Buffer.

TXReportRAM
A constant which gives the RAM location of TXReport. See WriteReport for more information.

RXReportRAM

A constant which gives the RAM location of RXReport. See ReadReport for more information.

Buffer (256) **as byte**

Some USB devices have large areas of USB dual port RAM (typically 18F2455, 18F2550, 18F4455 and 18F4550). This is a general purpose buffer which can be used for receiving or transmitting very large data blocks. Please note that this buffer is not available on devices that have limited dual port RAM. For example, the 18F2450 and 18F4450

TXReport (HIDReportBytesIn) **as byte**

This smaller array overlays Buffer and is dimensioned HIDReportBytesIn bytes. See WriteReport for more information. Please note that this buffer is not available on devices that have limited dual port RAM. For example, the 18F2450 and 18F4450.

RXReport (HIDReportBytesOut) **as byte**
This smaller array overlays Buffer and is dimensioned HIDReportBytesOut. See ReadReport for more information. Please note that this buffer is not available on devices that have limited dual port RAM. For example, the 18F2450 and 18F4450.

#option USB_DESCRIPTOR
Assign the name of the descriptor file used by the module. More information on descriptors can be found here.

#option USB_SERVICE

Enables or disables USB interrupt servicing. By default, this option is enabled. For more information, see Service().

#option USB_SERVICE_PRIORITY

Set the interrupt priority level. Can be ipLow or ipHigh. By default, the option is set to ipHigh.

USB CDC Library

Please note that this library is not available in the Special Edition (SE) version of the Swordfish compiler.

Module and Filename

CDC
USBCDC.bas

Interface

Subroutines and Functions

function Attached() **as boolean**
sub Service()
sub EnableISR()
sub DisableISR()
sub ClearOverrun ()
function DataAvailable() **as boolean**
function ReadByte () **as byte**
function ReadBoolean () **as boolean**
function ReadWord () **as word**
function ReadLongWord () **as longword**
function ReadFloat () **as float**
sub ReadArray**(byref** pArray() **as byte**, pCount **as word)**
sub WriteByte (pValue **as byte**)
sub WriteBoolean (pValue **as boolean**)
sub WriteWord (pValue **as word**)
sub WriteLongWord (pValue **as longword**)
sub WriteFloat (pValue **as float**)
sub WriteArray(**byref** pArray() **as byte**, pCount **as word**)
compound sub Read (**byref** pReadItem)
compound sub Write (pWriteItem)
function WaitFor (pValue **as byte**) **as boolean**
function WaitForTimeout (pValue **as byte**, pTimeout **as word**) **as boolean**
sub WaitForCount (**byref** pArray() **as byte**, pCount **as word**)
sub WaitForStr (pStr **as string**)
sub WaitForStrCount (**byref** pStr **as string**, pCount **as word**)
function WaitForStrTimeout (pStr **as string**, pTimeout **as word**) **as boolean**
function DataAvailableTimeout (pTimeout **as word**) **as boolean**
sub Rep (pValue, pAmount **as byte**)
sub Skip (pAmount **as byte**)

Variables

ReadTerminator **as char**
DTR **as bit**
RTS **as bit**
Overrun **as boolean**
Buffer(512) **as byte**
TXBuffer(256) **as byte**
RXBuffer(256) **as byte**

Events

OnControl **as** TOnControl

Compile Options

USB_DESCRIPTOR
USB_SERVICE
USB_SERVICE_PRIORITY

Overview

The CDC module provides enables you to send RS232 data via a USB connection. The module does not require a specialist driver on the PC, as it is supplied with the OS. This is perhaps one of the easiest methods to port legacy RS232 applications to USB

because (a) the firmware mimics a RS232 connection and (b) the PC creates a Virtual COM Port (VCP) which is seen by applications as just another standard COM connection. For example, COM7 etc.
This first task is to have a suitable PIC® microcontroller, which you can connect as shown in this schematic. Next, write your firmware and program onto the microcontroller. A simple program is shown below. When the device is first connected to the computer, windows will start the install USB driver wizard. All you need to do is point it towards a simple *.INI file. An example file is located in the compiler ..\Library\USB subfolder (MecaniqueCDC.ini) or you can take a look at it here. This is a one off operation, after which you can communicate with you microcontroller using a standard terminal program, the Swordfish Serial Communicator or any other application that uses standard RS232.

Additional Information

- USB Connection Schematic
- Sample Program
- CDC Driver Configuration File
- USB Descriptors and Options
- Vendor and Product ID (VID and PID)

Example Code

```
// device and clock...
device = 18F4550
clock = 48
// 20Mhz crystal, 48Mhz internal (FS USB)
config
PLLDIV = 5,
CPUDIV = OSC1_PLL2,
USBDIV = 2,
FOSC = HSPLL_HS,
VREGEN = ON
// import modules...
include "usbcdc.bas"
// wait for connection...
repeat
until Attached
// main program loop - this just simply reads a byte from a
// terminal window (for example, SerialCommunicator) and then
// echos it back...
while true
if DataAvailable then
WriteByte(ReadByte)
endif
Wend
```

Interface

function Attached() **as boolean**
Returns true if the USB bus is in a configured state, false otherwise.

sub Service()

USB is a polled protocol which needs to be serviced every 1 millisecond or so. This is to ensure the connection between the microcontroller and PC is maintained and any transactions are handled. Unless you disable this option using USB_SERVICE = false, you should never have to call Service directly(). Polling service manually just means inserting Service() in your code at regular intervals. For example,
// wait for connection...
repeat
Service
until Attached

sub EnableISR()
sub DisableISR()

If the USB_SERVICE is enabled (which it is by default) then the module will service the USB connection using interrupts. However, you may have a critical section of code that should not be disturbed. For example, a software based serial read or write may will have exact timing requirements. You don't want an interrupt firing during this time, else these timing will be disturbed and the routines may fail. You can temporarily disable interrupt servicing by calling DisableISR(). When you have finished, enable the interrupt by calling EnableISR().
Remember - if you have disabled the polling interrupt and your routines are likely to exceed 1 millisecond in duration, you need to manually service the USB connection.

sub ClearOverrun()
Clear overrun. If the RX buffer overruns, the overrun flag will be set and needs to be cleared with a call to ClearOverrun().

function DataAvailable() **as boolean**
Indicates if any data has been received.

function ReadByte() **as byte**
Read a single byte. This is a blocking call, which means it will not return until data is received. To see if data is available, use DataAvailable.

function ReadBoolean() **as boolean**
Read a boolean. This is a blocking call, which means it will not return until data is received. To see if data is available, use DataAvailable.

function ReadWord() **as word**
Read a word or integer. This is a blocking call, which means it will not return until data is received. To see if data is available, use DataAvailable.

function ReadLongWord() **as longword**
Read a long word or long integer. This is a blocking call, which means it will not return until data is received. To see if data is available, use DataAvailable.

function ReadFloat() **as float**
Read a floating point number. This is a blocking call, which means it will not return until data is received. To see if data is available, use DataAvailable.

sub ReadArray**(byref** pArray() **as byte**, pCount **as word** = 0**)**

Read an array of bytes. This is a blocking call, which means it will not return until pCount data bytes are received. If a value of pCount is not given, then pCount will be set to the number of elements contained in the array.

sub WriteByte(pValue **as byte**)

- *pValue* - Data to write.

Write a single byte.

sub WriteBoolean(pValue **as boolean**)

- *pValue* - Data to write.

Write a boolean.

sub WriteWord(pValue **as word**)

- *pValue* - Data to write.

Write a word or integer.

sub WriteLongWord(pValue **as longword**)

- *pValue* - Data to write.

Write a long word or long integer.

sub WriteFloat(pValue **as float**)

- *pValue* - Data to write.

Write a floating point number.

sub WriteArray(**byref** pArray() **as byte**, pCount **as word** = 0)

Write an array of bytes. The parameter pCount can be used to limit the number of array elements transmitted. If a value of pCount is not given, then pCount will be set to the number of elements contained in the array.

compound sub Read(**byref** pReadItem)

- *pReadItem* - Data to read.

Read multiple data items. Valid argument types include boolean, char, string, byte, shortint, word, integer, longword, longint and floating point. See also ReadTerminator.

compound sub Write (pWriteItem)

- *pWriteItem* - Data to write.

Write multiple items. Valid argument types include boolean, char, string, byte, shortint, word, integer, longword, longint and floating point.

function WaitFor(pValue **as byte**) **as boolean**

- *pValue* - Data to wait for.

A blocking call which will wait for data to be received. It returns true if *pValue* matches the data byte received, false otherwise.

function WaitForTimeout(pValue **as byte**, pTimeout **as word**) **as boolean**

- *pValue* - Data to wait for.
- *pTimeout* - Timeout value in milliseconds.

A blocking call which will wait for data to be received. It returns true if *pValue* matches the data byte received, false otherwise. The function will also return false if the timeout interval has been exceeded.

sub WaitForCount(**byref** pArray() **as byte**, pCount **as word**)

- *pArray* - Array of byte to receive data.
- *pCount* - Number of bytes to receive.

A blocking call which will wait for *pCount* data byte to be received.

sub WaitForStr(pStr **as string**)

- *pStr* - String to wait for.

A blocking call which will wait for *pStr* to be received. The sub will not return until the input string matches the string data received, See also *ReadTerminator*.

sub WaitForStrCount(**byref** pStr **as string**, pCount **as word**)

- *pStr* - String variable to receive data.
- *pCount* - Number of characters to receive.

A blocking call which will wait for *pCount* data characters to be received.

function WaitForStrTimeout(pStr **as string**, pTimeout **as word**) **as boolean**

- *pStr* - String to wait for.
- *pTimeout* - Timeout value in milliseconds.

A blocking call which will wait for *pStr* to be received. The function will return false if the timeout interval has been exceeded.

function DataAvailableTimeout(pTimeout **as word**) **as boolean**

- *pTimeout* - Timeout value in milliseconds.

A blocking call which will wait for a specified timeout value before returning. Returns true is data received, false is it timesout.

sub Rep(pValue, pAmount **as byte**)

- *pValue* - Data to transmit.
- *pAmount* - Number of times to repeat transmission.

Sends *pValue* data *pAmount* times.

sub Skip(pAmount **as byte**)

- *pAmount* - Number of times to skip incoming data.

A blocking call which will wait until *pAmount* data byte have been received.

ReadTerminator **as char**
Set the Read terminator character. By default, it is set to NULL (0).

DTR **as bit**

Holds the state of DTR.

RTS **as bit**
Holds the state of RTS. Please note that it appears that the default windows driver (usbser.sys) does not implement a RTS correctly from host to device - this is based on observations made and sources from the internet. However, RTS is mapped to the correct bit, should another driver become available.

Overrun **as byte**
Indicates if a buffer overrun error has occurred. Use ClearOverrun to correctly clear this flag.

Buffer(512) **as byte**

Some USB devices have large areas of USB dual port RAM (typically 18F2455, 18F2550, 18F4455 and 18F4550). This is a general purpose buffer which can be used for receiving or transmitting very large data blocks. Please note that this buffer is not available on devices that have limited dual port RAM. For example, the 18F2450 and 18F4450

TXBuffer(256) **as byte**

This smaller array overlays Buffer and can be used for transmitting large data blocks.

RXBuffer(256) **as byte**
This smaller array overlays Buffer and can be used for receiving large data blocks. It is located in RAM above the TXBuffer.

OnControl **as** TOnControl
You can assign a control event that can be fired when DTR or RTS changes state. For example,

```
// this event will fire if the DTR line
// from the PC is set or cleared...
event OnControl()
output(PORTD.0)
PORTD.0 = DTR
end event
// assign event handler...
CDC.OnControl = OnControl
```

#option USB_DESCRIPTOR
Assign the name of the descriptor file used by the module. More information on descriptors can be found here.

#option USB_SERVICE

Enables or disables USB interrupt servicing. By default, this option is enabled. For more information, see Service().

#option USB_SERVICE_PRIORITY

Set the interrupt priority level. Can be ipLow or ipHigh. By default, the option is set to ipHigh.

Utils Library

Modules and Filenames

Utils
Utils.bas

Interface

Subroutines and Functions

function Reverse (pValue **as TType**, pAmount **as byte**) **as TType**
function Digit (pValue **as TType**, pIndex **as byte**) **as byte**
function Min (pValueA, pValueB **as TType**) **as TType**
function Max (pValueA, pValueB **as TType**) **as TType**
sub Swap (**byref** pValueA, pValueB **as TType**)
function HighNibble (pValue **as byte**) **as byte**
function LowNibble (pValue **as byte**) **as byte**
sub SetAllDigital ()

Overview

Utilities library.

Interface

function Reverse(pValue **as TType**, pAmount **as byte**) **as TType**

- *pValue* - Variable to reverse.
- *pAmount* - Number of bits.

Reverse the bits of *pValue* by *pAmount*. TType can be byte, word or longword.

function Digit(pValue **as TType**, pIndex **as byte**) **as byte**

- *pValue* - Input value.
- *pIndex* - Digit index.

Return the value of a decimal digit. For example
Digit(123,3)
will return the number 1. TType can be byte, word or longword.

function Min(pValueA, pValueB **as TType**) **as TType**

- *pValueA* - First value.
- pValueB - Second value.

Returns the minimum of two numbers. TType can be byte, shortint, word, integer, longword, longint or float.

function Max(pValueA, pValueB **as TType**) **as TType**

- *pValueA* - First value.
- pValueB - Second value.

Returns the maximum of two numbers. TType can be byte, shortint, word, integer, longword, longint or float.

sub Swap(**byref** pValueA, pValueB **as TType**)

- *pValueA* - First value.
- pValueB - Second value.

Swaps two value. TType can be byte, shortint, word, integer, longword, longint, float or string.

function HighNibble(pValue **as byte**) **as byte**

- *pValue* - Input value.

Returns the high nibble of a byte.

function LowNibble(pValue **as byte**) **as byte**

- *pValue* - Input value.

Returns the low nibble of a byte.

sub SetAllDigital()
Set all microcontroller analog pins to digital.