Utilizing USBmicro's U4x1 USB I/O Board With RobotBASIC

Document Version 1.01, Applicable To RobotBASIC V4.0.0

RobotBASIC provides many ways you can communicate from a PC to a digital electronic circuit. One very effective, convenient, safe and versatile method is through the U4x1 USB I/O family of devices from USBmicro. There are two versions, the U401 and the U421.

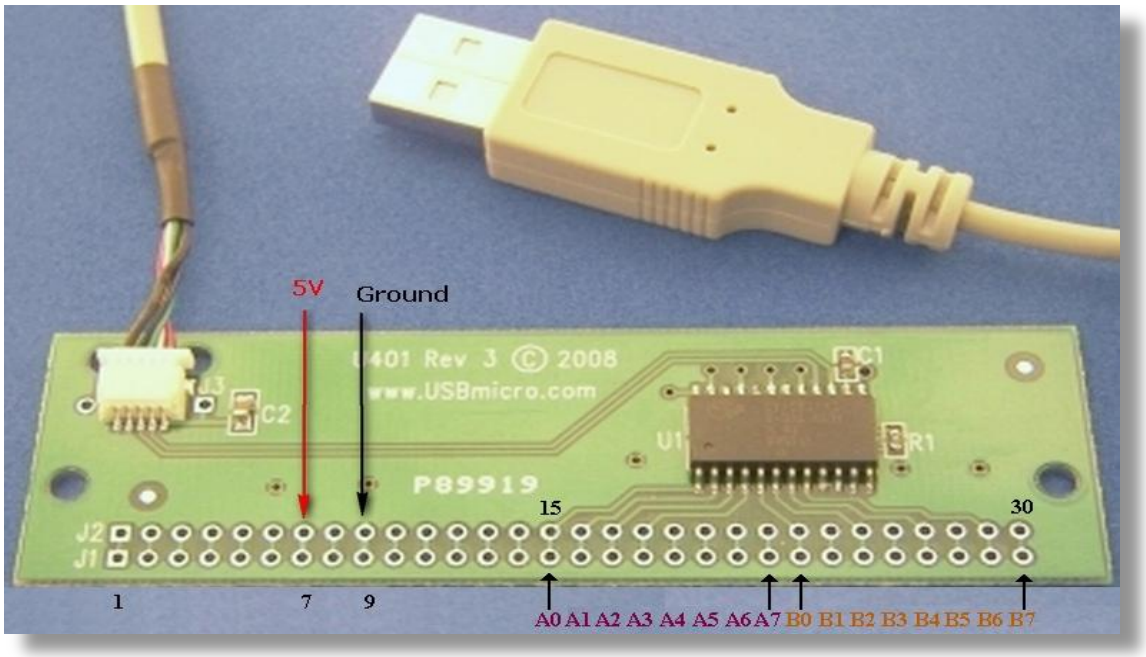


Figure 1: USBmicro's U401. A USB digital I/O Interface.

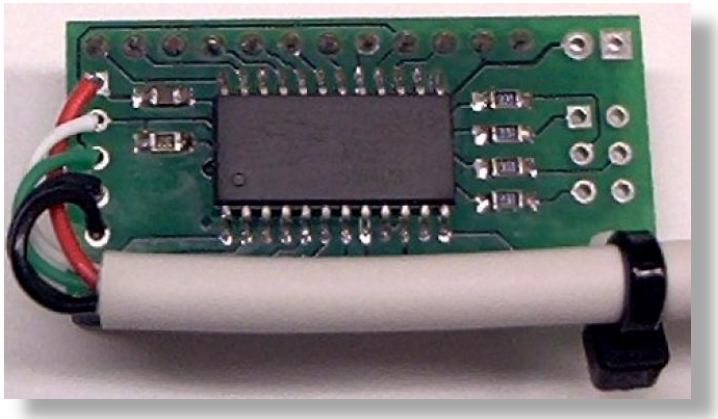


Figure 2: USBmicro's U421 USB digital I/O Interface

Note: ***The two devices are only different in the arrangement of the I/O pins, otherwise they are exactly the same. For simplicity this article will only deal with the U401 device.***

The U4x1 has 16 digital I/O pins that can sink a maximum of 50 mA, and source a maximum 5mA. The device is powered from the USB port so it can be very convenient for doing low powered I/O experiments and prototyping. If you need more current than can be provided by the U4x1 you have to use it with an appropriate power isolation circuit. You must also consider that a typical PC's USB port or a powered USB hub can source a maximum of 500 mA. The U4x1 device requests only 100 mA, making it work well with laptops. The device itself draws under 20 mA.

If you are familiar with using the parallel port on a PC to do digital I/O then you will find the U4x1 to be infinitely more versatile and convenient. The U4x1 is a lot more than just 16 plain digital I/O pins - you can use these pins in a straightforward manner as 2×8 bit ports with individual pins being configurable as either Input or Output. But you can also utilize the device to carry out SPI and 1-Wire serial communications with devices that support these protocols. Additionally, the U4x1 can be used to control an LCD and a Stepper motor (***control only - not a driver- you need to provide current driving separately***), or you can configure it to provide a parallel strobe signal.

For the SPI system you can configure the U4x1 to act as an SPI slave or SPI master. For the 1-Wire it acts as a Master. With these protocols you can easily interact with a plethora of devices like ADC, DAC, thermometers, compasses, GPSs, and much more.

RobotBASIC provides a suite of functions that provide access to all the utilities supported by the U401. All you need to do is have a U4x1 connected into one (or more) USB port on your computer. You can either employ the USB power that the U4x1 makes available through two of its pins (5V and Ground), or you can add supplementary power. In either case the U4x1 will enable you to do digital control of devices and to control electronics projects.

This document will show how to use the functions in RobotBASIC along with one U4x1 device. You can use multiple devices and you can use a U401 or U421. However, for the purposes of demonstrating all the functionalities we shall use only one U401.

Note: ***This document is not a replacement for the documentation of the U4x1 available from USBmicro . To make proper use of the U4x1 devices and to get deeper understanding of the operations of the functions in RobotBASIC and of the device's limitations and capabilities, you must read the USBmicro documentation and information. This document pertains to the usage of the functions within RobotBASIC that reflect the LOW LEVEL ones within the U4x1's ROM. To use them correctly you need to understand how the U4x1 devices perform them. To do this you MUST read the most up to date documentation. There you will find programming information that is kept up to date as well as further examples. For additional information on how to use the hardware - including demonstration programs written in RobotBASIC - read through the blog at www.circuitgizmos.com.***

Note: ***You may find Section 3 in the RobotBASIC_Networking.PDF document of great help in addition to the information in this document.***

Table Of Contents

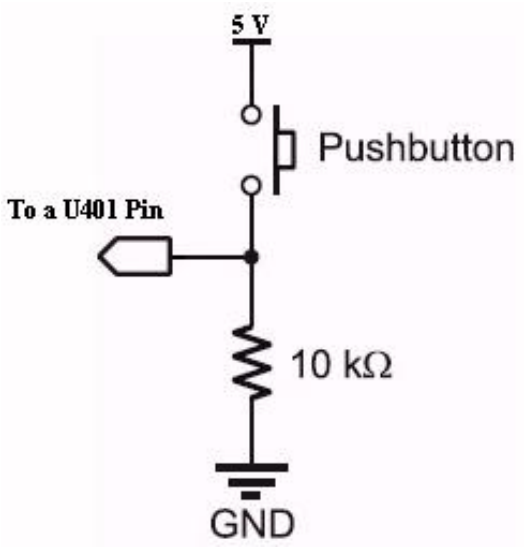
1- Building Circuits For Experimentation

In order to demonstrate the actions of the U401 and to exercise all the given demonstration programs you will need to wire up a few simple circuits. You may use a Bread Board if your U401 has the layout shown in Figure 3. This arrangement can be very easily inserted into a bread board for experimentation.

Figure 3: U401 with Pins installed.

Note: ***The schematics shown below will utilize the power provided by the USB port through the U401 pins (Pin 7 is+5V and Pin 9 is Ground). These schematics are only for the purposes of experimentation and are not suitable for other purposes.***

WARNING!!! ***Take care not to exceed the current limitations of the U4x1 or the PC's USB port. Wrongly wired components will cause damage to the USB hub and the U4x1***.



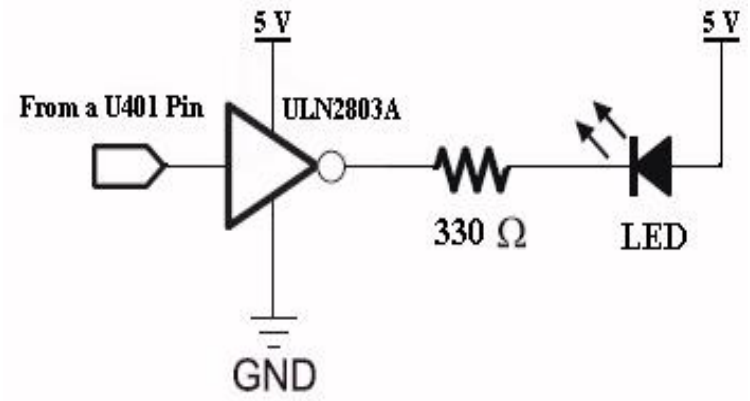


Figure 4: A Pushbutton and an LED arrangement. The Ground has to be common with the ground of the U401 (Pin 9, see Figure 1). The +5V can be either from a separate power supply or from the U401 (Pin 7, see Figure 1).

Since the U401 cannot source more than 5 mA, you must use a driver to supply current for an LED. This is achieved through a ULN2803A driver chip. This chip has 8 Inverting buffers that are Darlington transistor pairs and can provide the current required to light the LEDs. Since the buffer is an inverter the LED is wired to emit light when the output of the driver is LOW which will be when the Output pin from the U401 is high. Thus the LED will be effectively Active-High from the point of view of the U401.

The pushbutton is wired to have a pull down resistor which will cause the U401 Input pin to be low unless the Pushbutton is pushed down which will cause a high to be seen by the input pin. Thus the Pushbutton is Active-High from the point of view of the U401.

Other experiments will use an LCD and a 1-Wire thermometer chip (DS1822). We shall show the connections required in the appropriate sections later.

Whenever the demo programs below call for an LED you should use the arrangements shown in Figure 4. However, for the pushbutton we shall use a different arrangement (active low) that will be explained later.

2- Information About The USBmicro System

The way RobotBASIC is able to interface with the hardware of the U4x1 is through a specialized Dll (USBm.dll). This Dll provides the interface between the hardware, Windows and RobotBASIC.

2.1- Verifying that the Dll is installed and running

To verify that the Dll is accessible to RobotBASIC, look for the Dll by calling the function **usbm_DllSpecs()**. The function returns a string that contains 4 sections of information about the Dll. The sections are separated with the character | which enables the extraction of the different section with the Extract() function. Here is an example program that will verify that the Dll is installed by printing the information in each section of the string returned by usbm_DllSpecs():

```
Data Sections; "About:","Copyright:","Date:","Version"
s = usbm_DllSpecs()
xytext 10,10,"About    :","",15,fs_Bold \ xystring -1,-1,Extract(s,"|",1)
xytext 10,30,"Copyright:","",15,fs_Bold \ xystring -1,-1,Extract(s,"|",2)
xytext 10,50,"Version #:","",15,fs_Bold
xystring -1,-1,Extract(s,"|",4)," (",Extract(s,"|",3),")"
```

If the Dll is not somewhere where the program can find it, the returned string will be an empty string (""). You can use this fact to take actions in your program that warn the user. If the Dll is findable none of the functions in RobotBASIC will return valid data.

2.2- Verifying that there are U4x1 devices and how many

The next action you need to take before you start using the USBmicro functions is to verify that there are U4x1 devices actually connected to the PC. You do this with the **usbm_FindDevices()** function which returns true if there are devices and false if there are not. **usbm_FindDevices()** sets up the whole U4x1 Dll system for access and should be the first Dll function that you call before accessing any devices themselves.

You can find out how many U4x1 devices there are attached to the PC by using the function **usbm_NumberOfDevices()**, which returns the total number of devices connected to the PC. In all the

programs that we will develop later we will ignore other devices if there is more than one. In other words, these examples use only one U4x1 device, but since each device has a unique serial number, each device can be accessed/treated distinctly. You can have up to 50 U4x1 devices on a single PC.

Device numbering is 0 indexed. So the first device is device number 0, the second device is number 1 and so forth. Here is a program that will find out if there are devices and if there are it will find out how many.

```
if USBm_DllSpecs() != ""
    if USBm_FindDevices()
        print "There are ",USBm_NumberOfDevices()," Devices"
    else
        print "There are no Devices found"
    endif
else
    print "The USBm.Dll is not installed"
endif
```

2.3- Obtaining device information

The function **usbm_DeviceSpecs(ne_DeviceNumber)** returns a string that has information about a particular device. Choose a device to use (you can use any or all) and verify that the device is the correct device you are expecting by reading the information about it and examining this information to verify that it is the device you need. Before you use a device you should always verify that it is still valid (that it has not been unplugged) and is functioning correctly with the function **usbm_DeviceValid(ne_DeviceNumber)**. This function will return true if the device is still functioning correctly and false otherwise.

Here is a program that will iterate through all the devices plugged to the PC and will return information about them:

```
data sections;"Device ID: ","Prduct ID: ","Vendor ID: ","Mnfcturer: "
data sections;"Product  : ","Serial No: ","Firmware : "
If USBm_DllSpecs() != ""
   f = USBm_FindDevices()
   n = USBm_NumberOfDevices()
   if n > 0
      for i=0 to n-1
         s = usbm_DeviceSpecs(i)
         if usbm_DeviceValid(i)
            print "Device :",i
            print sRepeat("-",10)
            for j=1 to 7
               print sections[j-1],extract(s,"|",j)
            next
            print
         else
            print "Device ",i," is invalid"
         endif
```

```
      next
   else
      print "There are no Devices installed"
   endif
else
  print "The USBm.Dll is not installed"
Endif
```

Notice that the Product name is the 5th field in the specification string returned by the function **usbm_DeviceSpecs()**. So if you want to verify that the device is the U401 for instance then you need to extract the 5th section of the string and verify that it is "U401".

3- Utilizing The U4x1 Devices

To summarize the actions you need to perform before you can actually use one (or more) of the devices (as described in the previous section):

1- Verify the USBm.Dll can be found by RobotBASIC or your RobotBASIC program.
2- Find installed U4x1 devices.
3- Find out the number of installed devices.
4- Iterate through them and verify which one is the device you want to use.
5- Before using the device, always verify that it is still valid (that it has not been unplugged).

Now that you have decided on a device and verified that it is valid, you can start performing I/O with the device. You may use more than one U4x1 device simultaneously. ***However in this document we shall assume that there is only one U401 connected to the PC and that it is device number 0 (1st device)***.

3.1- Using the U401 for digital I/O

The U401 has 16 I/O pins - two 8 bit Ports (A and B). See Figure 1. The pins A0 to A7 (physical pins 15 to 22) are Port A and pins B0 to B7 (physical pins 23 to 30) are Port B. You can designate each I/O pin as an Input or as an Output. Initially when the device is plugged into the PC's USB port it will be initialized with all I/O pins as INPUT. You can also make all I/O pins as input at any time using the function **usbm_InitPorts(ne_DeviceNumber)** which resets the device and reinitialize it to all I/O pins as inputs.

3.1.1- Assigning which pins are Input and which are Output

To use the device you must decide what I/O pins are going to be set as Inputs and which ones you want to be designated as Outputs and then set the device to that format. This is achieved with the functions: **usbm_DirectionA(ne_DeviceNumber,ne_PinsDirection,ne_PinsFormat)** and **usbm_DirectionB(ne_DeviceNumber,ne_PinsDirection,ne_PinsFormat)**

The parameter ***ne_PinsDirection*** should reflect the desired format of each pin. If the bit in the byte

corresponding to the pin's position is 0 then the pin is an input pin, and if the bit is 1 then the pin is an output pin. ***The parameter ne_PinsFormat is used to do some advanced options (see later), but for now it should match ne_PinsDirection.***

For example, if you want to set Port A on the first device (device number 0) to have pin A0 and pin A4 as inputs and the rest as outputs then use (pin numbering in the byte is from right to left so the LSBit is bit number 0 and the MSBit is bit number 7):

```
        n = usbm_DirectionA(0,0%11101110, 0%11101110) //clearest method
or      n = usbm_DirectionA(0,0xEE, 0xEE)
or      n = usbm_DirectionA(0,238, 238)
```

Notice that you can use binary literals (e.g. 0%11001010) or hexadecimal literals (e.g. 0xCA) or normal decimal literals (e.g. 202). However, in this case it is most convenient to use the binary format. With the binary format you can ensure that every pin has a corresponding bit either as 1 to designate it as an output pin or a 0 to designate it as an input pin.

In the example above we used 0%11101110. So counting from right to left pin A0 and A4 will be input pins. Pins A1 to 3 and A5 to 7 will be output pins.

Note: LSBit signifies the Least Significant Bit and MSBit means Most Significant bit. In binary values we start counting from right to left where the first bit from the right is the LSBit which is bit number 0. The MSBit is the left most bit where in an 8 bit byte it is byte 7, and in a 16 bit number it is byte number 15 and so forth.

3.1.2- Writing to Output pins

The following functions are used to write a 1 or 0 to each Output pin either individually or as a group of 8 bits (Port):

usbm_WriteA(ne_DeviceNumber,ne_ByteValue) writes to all 8 pins A0 to A7.
usbm_WriteB(ne_DeviceNumber,ne_ByteValue) writes to all 8 pins B0 to B7.
usbm_SetBit(ne_DeviceNumber,ne_PinNumber) sets (1) an individual pin.
usbm_ResetBit(ne_DeviceNumber,ne_PinNumber) resets (0) and individual pin.
usbm_WriteABit(ne_DeviceNumber,ne_AndingMask, ne_OringMask) see below.
usbm_WriteBBit(ne_DeviceNumber,ne_AndingMask, ne_OringMask) see below.

Writing a value to an Output pin will affect the state of that pin. However if the pin is an input pin then there will be no effect on the pin since it is an input and thus its state cannot be changed.

In our previous example we defined A0 and A4 as inputs and the rest as outputs. So let's say we want to put a 1 on A2 and a 0 on A6 we would say:

```
n = usbm_SetBit(0,2)
n = usbm_ResetBit(0,6)
```

The above is the best way to change the state of a pin if you are going to do one at a time. Notice that

in this situation we cannot write to the entire 8 bits in one go since that would affect the other output pins. But we could read the current status of the port and then change the bits we want to change then write back the resulting value to the port. Like this:

```
n = usbm_ReadA(0)  //read the current status of the port all 8 bits
n = SetBit(n,2) \ n= ClrBit(n,6)  //set (1) A2 and reset (0) A6.
                                  //This way we do not change others
n = usbm_WriteA(0,n) //write the value back to the port all 8 bits but only
                     //changed bits will have any effect since we did not change
                     //the other bits. Also input pins are not affected anyway.
```

Note: ***The above is relying on some RobotBASIC functions to manipulate individual bits in a number. RB has many bit-wise operators and bit manipulation functions that can set, reset or read the value of bits in a number. Here is a list:***

Bitwise Operators:

~ @ | & >> << bRotL bRotR

Functions:

MakeBit(ne_Number,ne_BitPosition,on|off)
SetBit(ne_Number,ne_BitPosition)
ClrBit(ne_Number,ne_BitPosition)
GetBit(ne_Number,ne_BitPosition)
BitSwap(ne_Number{,ne_NumberOfBits})

You can use the above functions to manipulate individual bits in a number and change their values or obtain their values and so on.

A third way for doing the same action we did in the previous examples is to use the functions **usbm_WriteABit()** and **usbm_WriteBBit()**. They are in a way similar to the second method we used in the example above, but without the explicit need to read the port first, it is read implicitly by the function. The way usbm_WriteABit() (same for B) performs its task is like this:

a- It reads the current states of the A port.
b- It then ANDs this value with the ***ne_ANDingMask***.
c- It then ORs the result of step b with the ***ne_ORingMask***.
d- It then writes the result of step c to the A port.

This is all performed by the U401 in hardware and there is no need to use RB functions to manipulate the port value and then write it back. Therefore, this method will execute a lot faster.

So in our example we wanted to set A2 and reset A6. So we need the ANDing mask to be %10111011 and we need the ORing mask to be 0%00000100. The logic is a bit complicated but here is a simple rule:

Put a 0 in the bit corresponding to the pin you want to change in the ANDing mask and a 1 for any pin you do not want to change. In the ORing mask put a 0 for any pin you do not want to

change. But for the pins you want change put the value as desired (0 or 1).

So according to the above rule, since we want to change the value of A2 and A6 then the ANDing mask has to have bits 0,1,3,4,5,7 as 1 and bits 2 and 6 as 0. So the ANDing mask has to be 0%10111011. Also since we don't want to change the pins 0,1,3,4,5 and 7, the bits have to be 0 in the ORing mask. But for bit 2 we want a 1 since we want to set A2. For bit 6 we want a 0 since we want to reset A6. So the ORing mask has to be 0%00000100. So to do the example above:

```
n = usbm_WriteABit(0, 0%10111011, 0%00000100)
```

You see it is a lot faster and easier except for figuring out what the masks have to be, but, once you know how to work that out this command is very convenient. However, any of the above three methods is good and you chose the method most suitable to the situation.

3.1.3 - Reading from Input pins

You cannot read the state of an individual pin by itself. You can read the entire port it is in and then use the **&** operator to ***mask*** the other bits. The functions to use are:

usbm_ReadA(ne_DeviceNumber)
usbm_ReadB(ne_DeviceNumber)

So for example to read the status of the A4 pin you would need to read the entire A port and then AND it with the value 0%00010000 to mask all other bits except for the bit number 4 (5th bit). So you would say: `n = usbm_ReadA(0) & 0%00010000`. So now the variable *n* would either be a 1 or 0 depending on the state of the A4 pin.

Note: There is a special case format that the U4x1 supports for configuring input pins. If you do not want to go to the trouble to set up a Pull-down or Pull-Up circuit for the pin and leave it floating then you can set the U4x1 to create an internal Pull-Up resistor. This means that if the pin is not connected to anything then it will be as if it is connected to 5V (i.e. 1). In this case the pushbutton will then pull the pin down when pushed see Figure 5. This will make the pushbutton an Active-Low button.

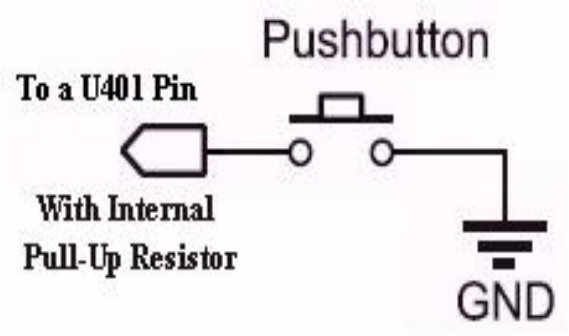


Figure 5: Active-Low Pushbutton for pins with Internal Pull-up resistors.

To set a pin as an input and also with an internal pull-up resistor you need to use the functions usbm_DierectionA() or usbm_DirectionB(). The bit in the ***ne_PinsDirection*** corresponding to the pin has to be 0 and the bit in the ***ne_PinsFormat*** corresponding to the pin has to be 1. But also right after using the function to set the direction you must write a 1 to the corresponding pin using any of the functions usbm_WriteA() or usbm_WriteB() or usbm_SetPin().

For example to set all the pins in port B as inputs with a pull-up resistor we would do:

```
n = usbm_DirectionB(0,0,255)
n = usbm_WriteB(255)
```

Note: ***For the circuits we will build in the demo programs developed in the rest of this document we shall use the above setup. That is the Pushbuttons will be Active_Low and the U401 will be set to create an internal pull-up resistor for the input pins.***

3.1.4- Demo programs using the U401 for digital I/O

We are going to write 3 programs. In the first one we will connect some Pushbuttons to some pins of the U401. The program will continuously read these pins and display their status on the PC screen.

In the second program there will be a few check boxes on the screen. The user can check or uncheck any of the boxes which will continuously be reflected on output pins of the U401 which will cause LEDs connected to them to turn on or off.

In the third program we will combine the above two programs. We will read pushbuttons on some pins and show their states on the screen but at the same time we will turn on/off a corresponding LED connected to output pins. This way the PC acts as a relay between the pushbuttons and the LEDs.

While these programs are simple, they illustrate all the I/O operations you are likely to need for using the U4x1 as a digital I/O device.

Program 1 (Input)

In this program we will use port pins B0 to B7 as inputs (ignoring the A port) connected to pushbuttons. If you do not want to connect all 8 pins to pushbuttons then just leave them unconnected since we are going to use the internal pull up resistor configuration. But this means that all unconnected pins will be read as 1. See Figure 1 to verify which pins are B0 to B7. They are pins 23 to 30 counting from left to right with the setup shown in Figure 1.

To connect a pushbutton to the U401 use the setup in Figure 5 as described in Section 3.1.3, since we are going to configure the U401 to have input pins with internal pull up resistors.

When the program runs, any pins with no pushbutton will always show a 1 since they have an internal pull up resistor to +5V. The pins connected to the pushbutton should show a 0 when the button is pressed and a 1 otherwise. Here is the program:

```
if usbm_DllSpecs() != ""
   if usbm_FindDevices()
      //---there is a device and we will use device 0
      n=usbm_DirectionB(0,0,0xFF)  //set port B0 to B7 as inputs
      n = usbm_WriteB(0,0xFF)      //with internal pull up resistors
      xytext 20,10,"Port B Status:","",25,fs_Bold|fs_Underlined
      while true
          xytext 30,60,bin(usbm_ReadB(0),8),"",30,fs_Bold
```

```
      wend
   else
      print "There are no Devices"
   endif
else
   print "The USBmicro DLL is not installed"
endif
```

Program 1: Port B Input.

Program 2 (Output)

In this program we will use port pins A0 to A7 (ignoring the B port) as output pins connected to LEDs. If you do not want to connect all 8 pins to LEDs then just leave them unconnected. See Figure 1 to verify which pins are A0 to A7. They are pins 15 to 22 counting from left to right with the setup shown in Figure 1.

To connect an LED to the U401 use the setup in Figure 4 as described in Section 1.

The pins connected to an LED should turn on when the corresponding check box is checked and off when unchecked. Here is the program:

```
if usbm_DllSpecs() != ""
   if usbm_FindDevices()
      //---there is a device and we will use device 0
      n=usbm_DirectionA(0,0xFF,0xFF)  //set port A0 to A7 as outputs
      xyText 10,10,"Set Port A Status:","",20,fs_Bold|fs_Underlined
      for i=0 to 7
         addcheckbox ""+i,30+20*(7-i),60," "
      next
      while true
          for i=0 to 7
             if getcheckbox(""+i)
                n = usbm_SetBit(0,i)
             else
                n= usbm_ResetBit(0,i)
             endif
          next
          delay 100
      wend
   else
      print "There are no Devices"
   endif
else
   print "The USBmicro DLL is not installed"
endif
```

Program 2: Port A Output.

Program 3 (Input/Output)

In this program we will use port pins A0 to A7 as output pins connected to LEDs and port pins B0 to B7 as inputs connected to pushbuttons. If you do not want to connect all pins to LEDs and push buttons, make sure that for each button on a pin in B there is an LED on A. See Figure 1 to verify which pins are which.

To connect an LED use the setup in Figure 4 as described in Section 1. To connect pushbuttons use the setup in Figure 5 in Section 3.1.3. The pins in the A port connected to an LED should turn off when the corresponding pushbutton connected to port B is pushed down and they should be on otherwise. Here is the program:

```
if usbm_DllSpecs() != ""
   if usbm_FindDevices()
      //---there is a device and we will use device 0
      n=usbm_DirectionA(0,0xFF,0xFF) //set port A0 to A7 as outputs
      n=usbm_DirectionB(0,0,0xFF)    //set port B0 to B7 as inputs
      n=usbm_WriteB(0,0xFF)          //with internal pull-up resistors
      xyText 10,10,"Port B => A Status:","",20,fs_Bold|fs_Underlined
      for i=0 to 7
         addcheckbox ""+i,30+20*(7-i),60," "
      next
      while true
          n = usbm_ReadB(0)
          for i=0 to 7
             if GetBit(n,i)
                m = usbm_SetBit(0,i)
                SetCheckBox ""+i
             else
                m = usbm_ResetBit(0,i)
                SetCheckBox ""+i,false
             endif
          next
          delay 100
      wend
   else
      print "There are no Devices"
   endif
else
   print "The USBmicro DLL is not installed"
endif
```

Program 3: Port B Input Piped To Port A Output.

3.2- Using the U4x1 to control an LCD

One really useful device often used in standalone devices is the Liquid Crystal Display (LCD). An LCD is useful for displaying information and status about the actions of the device as well as a method for interacting with the user of the device during user operations.

With a device that is connected to a PC you would more likely use the PC's screen as the display device rather than an LCD. Nonetheless, there are situations where you may have an embedded PC without a display (PC displays are bulky, power hungry and expensive), or you may have the PC remote from the device that it would be inconvenient for the operator of the device to be operating the device while observing the PC's display. In these situations an LCD would be quite a convenient display device.

There are many different LCDs and some can even be controlled using an RS232 link. The U4x1 devices support a specific family of LCDs that follow the HITACHI HD44780 standard for controlling an LCD. Also see this PDF document for more information.

From reading the above two sources you will most likely have concluded that this is way too complicated. Well, the U4x1 makes it utterly simple to use any LCD that can support the above standard.

This family of LCDs can be controlled with 12 I/O pins from the U4x1 to control and communicate with the LCD to display text and change its setup and so on.

The LCD has 4 control lines and 8 data lines. The control lines are used to configure the LCD and the data lines are a byte of either a command or the ASCII code of a character to be displayed, depending on the state of the control lines.

So to control the LCD and clear it or send a char to be displayed and so and so forth, a U4x1 needs to connect to the 4 control lines and the 8 data lines. To use a U4x1 to display data on the LCD and to control it use the following functions:

```
usbm_InitLCD(ne_DeviceNumber,ne_Sel, ne_Port)      //designate the pins on the U4x1 to
                                                   //be used for controlling the LCD and
                                                   //send data to it.
usbm_LCDCmd(ne_DeviceNumber,ne_CommandByte)        //send a command to the LCD like
                                                   //clear
usbm_LCDData(ne_DeviceNumber,ne_DataByte)          //send a character to display or a
                                                   //parameter for a command.
```

The function usbm_InitLCD() is used to tell a U4x1 which I/O pins to use as data lines and which to use as the 4 control lines. The HD44780 LCDs use 4 control lines (RW, RS, E and Reset). The ***ne_Sel*** parameter defines which I/O pin to use as the one connected to the RW in the high nibble of the ***ne_Sel*** byte and to define the RS line in the lower nibble of the byte (see Table 1 below).

The nibble value can be from 0 to 15 (0x0 to 0xF). 0 means pin A0, 1 means A1 and so on where 8 means B0 and 9 means B1 and so forth up to 15 being B7. So for example if we want to use B1 to be the RS line and B2 to be the RW line then the high nibble of ***ne_Sel*** has to be 10 (0xA) and the lower nibble has to be 9 (0x9). So the value of ***ne_Sel*** has to be 0xA9 (169).

For the ***ne_Port*** parameter the high nibble is used to determine which port A or B to use as the 8 data lines. This will set the entire 8 lines of the data port. So if the high nibble of ***ne_Port*** is 0 then port A will be used to be the 8 data lines. If it is 1 then port B will be used.

The lower nibble of ***ne_Port*** is used to define which I/O pin to use to connect to the E line of the LCD. Again the value ranges from 0 to 15 where, as before, A0 = 0 and A1 = 1 ….B0 = 8 …B7 = 15. So for example if we want to use Port A as the data port and Pin B5 as the E line then the high nibble has to be 0 and the low nibble has to be 13 (0xD). So the ***ne_Port*** parameter would be set to 0x0D.

Let's do another example. We want Port B to be the data lines. We want A1 to be the RW line, A2 as the RS line and A3 as the E line. So

Parameter	High nibble	Low nibble	Value
ne_Sel	1 i.e. pin A1 = RW	2 i.e. pin A2 = RS	0x12 = 18 decimal
ne_Port	1 i.e. Port B = Data Port	3 i.e. pin A3 = E	0x13 = 19 decimal

Table 1: usbm_InitLCD() parameters calculations.

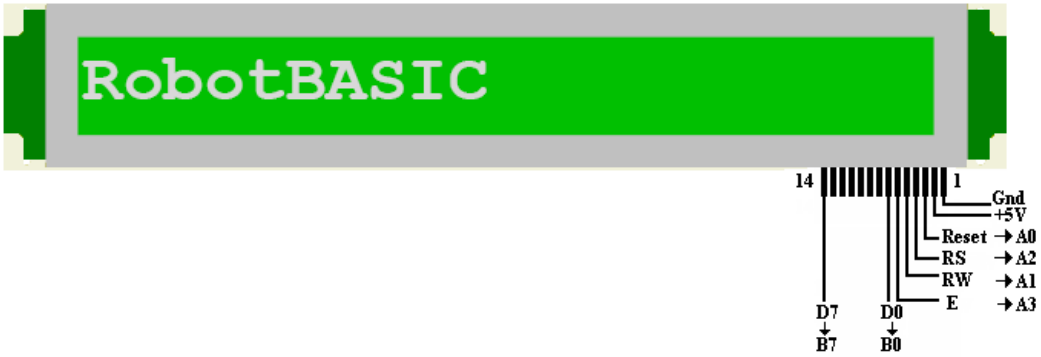
As far as the Reset line you can use any pin (for example we shall use A0). This pin is then driven as an individual I/O pin to be High then Low then High again to reset the LCD and have it ready for further commands to initialize it and set it up for the correct format. This will be illustrated with a full example later. Not all character LCD displays have a reset line. Some have, instead, a contrast input and the reset function is not present.

So now we use the code line `n = usbm_InitLCD(0, 0x12, 0x13)` to achieve the above setup. Later we say `n = usbm_LCDData( 0, Ascii("A"))` to send the character 'A' to be displayed. Also to clear the LCd for example, we would send the control byte 0x1, `n = usbm_LCDCmd(0,0x1)`.

3.2.1 A simple program to control an LCD with a U401

As a concrete example let's develop a few programs to use the setup in the table above. Initially we shall develop a simple program to establish the principles then we are going to develop a more sophisticated one.

The demos will assume that you are using the one line by 24 characters LCD (WD-C2401P-1GNN) which uses 8 data lines in addition to the RESET, RW, RS and E control lines. Other LCD displays have very similar setup.



Connect the Ground pin (physical pin 1 on the LCD) to the same ground pin as the U401 (physical pin 9). You can use the Power pin from the U401 (physical pin 7) or a separate +5V power supply with its Ground also connected to the Ground of the U401.

Then Connect the RS, RW and E lines (physical pins 4, 5, 6) of the LCD to the I/O pins A1, A2 and A3 (see the table above and also Figure 1) on the U401. Also connect the RESET line (physical pin 3) of the LCD to the A0 pin on the U401. Additionally, connect the data lines D0 to D7 on the LCD (physical pins 7 to 14) to the pins B0 to B7 on the U401.

With all the connections above completed we can now write a program as per the specifications above to interact with the LCD and display data on it.

Note: ***Many different LCDs follow the very same setup and can be controlled by the U401. However, you need to know the control codes and physical pins arrangements for the particular LCD to be able to use the program below with it. Read the LCD's documentation to ascertain the necessary information.***

```
01 MainProgram:
02  DeviceNum = 0
03  if usbm_DllSpecs() != ""
04    if usbm_FindDevices()
05      //---there is a device and we will use device 0
06      n = usbm_DirectionA(DeviceNum,0x0F,0x0F)  //A0 to A3 output
07      n = usbm_DirectionB(DeviceNum,0xFF,0xFF) //Port B output
08      GoSub Init_LCD
09      S = "Hello there"
10      while true
11         GoSub Write_LCD
12         Input "Enter a text:",S
13      wend
14    else
15      print "There are no Devices"
16    endif
17  else
18    print "The USBmicro DLL is not installed"
19  endif
20 end
```

```
//=================================================
Init_LCD:
  if !usbm_DeviceValid(DeviceNum) then return
  n = usbm_InitLCD(DeviceNum, 0x12, 0x13)   //---RW=A1,RS=A2,E=A3,Data=Port B
  n = usbm_SetBit(DeviceNum,0)  \ delay 20  //---RES connected to A0
  n = usbm_ResetBit(DeviceNum,0)\ delay 100 //High-Low-High to reset with 100 ms pulse
  n = usbm_SetBit(DeviceNum, 0) \ delay 100 //give the LCD time to settle
  n = usbm_LCDCmd(DeviceNum, 0x1C)  //---command sequence to setup the LCD
  n = usbm_LCDCmd(DeviceNum, 0x14)  //---this is obtainable from the device
  n = usbm_LCDCmd(DeviceNum, 0x28)  //---specs sheet
  n = usbm_LCDCmd(DeviceNum, 0x4F)  //
  n = usbm_LCDCmd(DeviceNum, 0xE0)  //
  n = usbm_LCDCmd(DeviceNum, 0x1)   //--clear the LCD display
Return
//=========================================================
Write_LCD: //S is the string to write
  if !usbm_DeviceValid(DeviceNum) then return
  n = usbm_LCDCmd(DeviceNum, 0x1)  //code 0x1 clears the LCD
  if Length(S) == 0 then Return
  for i = 1 To Length(S)
    n=usbm_LCDData(DeviceNum,GetStrByte(S,i)) //send ascii code of each character
  next
Return
//=========================================================
```

Program 4: A Simple LCD program.

The main program should be familiar by now. It ensures that there is access to the DLL and that there are devices (***usbm_FindDevices() should always be called before doing anything with the USBmicro functions***). As before we assume that the U401 is device zero. However this time we define a variable to hold the number of the device we wish to use. This way the subroutines can easily be made to work with any device number by just changing this one variable.

Line 06 in the main program designates pins A0 to A3 as output pins. These pins will be used, as discussed earlier, to control the RW, RS, E and RESET lines of the LCD. Line 07 designates Port B as all output pins so as to act as the 8 data lines to connect to the LCD. The main program then calls the ***Init_LCD*** subroutine.

The ***InitLCD*** subroutine calls the usbm_InitLCD() function discussed earlier. Notice that it uses the parameters calculated in Table 1. Also notice how the subroutine sets the A0 pin high then low then waits for 100 ms then high again. This is so as to reset the LCD and make it ready to receive commands. The routine then sends a few control codes to the LCD so as to configure it. These codes are from the specifications of the LCD device and can be different for you device but, then again they may work with no change. The final code sent is to clear the LCD screen.

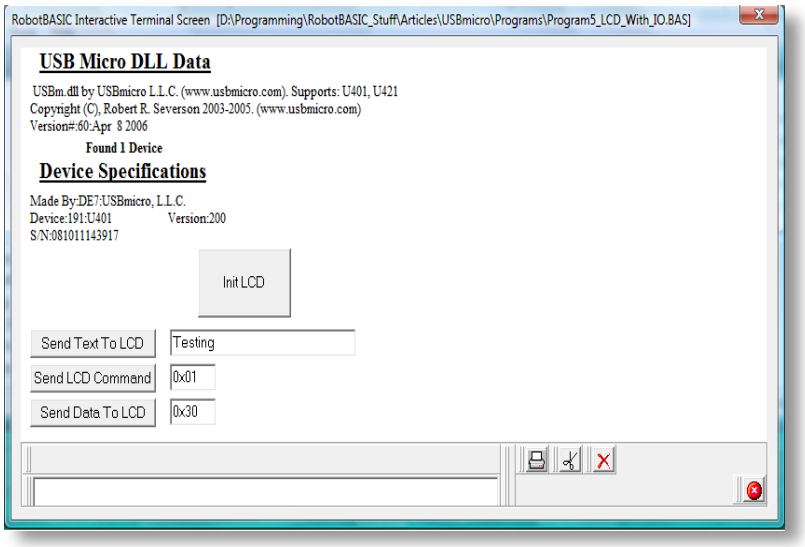
The main program sets the variable *S* with an initial display text and then line 11 calls ***Write_LCD*** subroutine to display the text and the loop then keeps asking the user for a new text to display.

The ***Write_LCD*** subroutine expects *S* to have the text to be displayed on the LCD. Notice how it iterates through the string obtaining one character at a time and then sends it to the LCD as data. Notice the function GetStrByte() used to extract the character. This function does not extract the character as a string rather it returns its ASCII code. So the function returns a number that represents the byte value of the character at the specified position within the string. We need this because when we send the character to the LCD it expects the ASCII code of the character not the actual text of the character. We could have achieved the same thing with `Ascii(SubString(S,i,1))`. But GetStrByte() is a lot more convenient and versatile. Read about this function and other similar functions in the RobotBASIC Help File and also in the document RobotBASIC_Networking.PDF.

We shall use these two subroutines again later. Study them carefully to see how they apply the functions and the principles discussed in Section 3.2.

3.2.2 A better program

This program will use the same two subroutines we established in the previous section as well as new ones to build a better user interface. The program will provide buttons for the user to initialize the LCD, to send a command byte to the LCD and to send a data byte to the LCD. Additionally we shall allow for sending a whole line of text as specified by the user. To allow for this, the program will provide a few edit boxes and a few buttons as the GUI interface with the user.



Another improvement in this program is that we will use an LED and a real physical Pushbutton along with the LCD. The LED will serve as a means for indicating that the system is working by blinking the LED continuously. Also the pushbutton will serve as a way for the user to indicate to the RB program that an action is required. The action will be simple in this example, but it serves to illustrate how such an interaction can be performed. Pushing the button will cause the RB program to send codes to clear the LCD screen and then it will send the characters to display a number that will be incremented every time the user presses the pushbutton.

Note: ***This program illustrates how you can have PC side user interfacing as well device side user interfacing all the while maintaining communications between the device and the PC for control.***

To summarize; the program will:

1- Display information about the device and DLL in a nice way on the PC screen.
2- Display edit-boxes and buttons on the PC screen for the user to enter the data to be sent to the LCD and to actuate the desired actions.

3- The program will continuously blink an LED on the same circuit as the LCD. We will use pin A6 from the U401 to do that.
4- There will be a pushbutton on the LCD circuit connected to pin A5 on the U401 that will be monitored and once pushed it will cause a number to be displayed on the LCD that reflects the number of times the button has been pushed since the start of the program.

This program demonstrates how to combine all that we have learnt so far. Therefore study it carefully and make sure that you understand all its actions.

```
MainProgram:
  GoSub Initialization
  GoSub MonitorInputs
End
//================================================
Initialization:
  DeviceNum = 0 \ LED_Pin = 5 \ LED_On = 1 \ PB_Pin = 6 \ PB_Count = 0
  LCDText = "LCDText" \ LCDData = "LCDData" \ LCDCmd = "LCDCmd"
  AddEdit LCDText,160,430,200,0,"Testing","Enter text to display on the LCD"
  AddEdit LCDCmd,160,460,50,0,"0x01","Enter the LCD command in hex or decimal"
  AddEdit LCDData,160,490,50,0,"0x30","Enter the LCD data in hex or decimal"
  IntegerEdit LCDCmd \ IntegerEdit LCDData
  InstBtn = "Instructions" \ InitLCDBtn = "Init LCD" \ SndTxtBtn = "Send Text To LCD"
  SndCmdBtn = "Send LCD Command" \ SndDataBtn = "Send Data To LCD"
  data Buttons;InitLCDBtn   ,190,360,100,60,"Initialize the LCD"
  data Buttons;SndTxtBtn    ,10,430,135,0,"Send the Text String to the LCD"
  data Buttons;SndCmdBtn    ,10,460,135,0,"Send the Command byte to the LCD"
  data Buttons;SndDataBtn   ,10,490,135,0,"Send The Data byte to the LCD"
  for n=0 to MaxDim(Buttons,1)-1 step 6
    AddButton Buttons[n],Buttons[n+1],Buttons[n+2],Buttons[n+3],Buttons[n+4],Buttons[n+5]
  next
  Gosub Display_DllSpecs
  GoSub Display_DeviceSpecs
  GoSub Init_LCD
Return
//================================================
MonitorInputs:
  while true
01    n = usbm_WriteABit(DeviceNum,255-2^LED_Pin,2^LED_Pin*LED_On)
02    LED_On = !LED_On
03    GoSub Check_PushButton
    GetButton btn
    if btn == InitLCDBtn
      GoSub Init_LCD
    elseif btn == SndTxtBtn
      S = GetEdit(LCDText)
      n = usbm_LCDCmd(DeviceNum, 0x1)  //clear LCD
      GoSub Write_LCD
    elseif btn == SndCmdBtn
      n = ToNumber(GetEdit(LCDCmd),0)
      n = usbm_LCDCmd(DeviceNum,n )
    elseif btn == SndDataBtn
      n = ToNumber(GetEdit(LCDData),0)
      n = usbm_LCDData(DeviceNum,n )
    endif
  wend
Return
//================================================
```

```
Display_DllSpecs:
  xyText 20,0,"USB Micro DLL Data","Times New Roman",15,fs_Bold|fs_Underlined
  m = usbm_DLLSpecs()
  xyText 10,30,Extract(m,"|",1),"Times New Roman",10
  xyText 10,45,Extract(m,"|",2),"Times New Roman",10
  xyText 10,60,"Version#:"+Extract(m,"|",4)+":"+Extract(m,"|",3),"Times New Roman",10
Return
//================================================
Display_DeviceSpecs:
  RectangleWH 0,75,230,120,white,white
  n = usbm_finddevices()
  n = usbm_numberofdevices()
  m = ""
  if n != 1 then m = "s"
  xyText 10,79, spaces(20)+"Found "+n+" Device"+m,"Times New Roman",10,fs_Bold
  xyText 20,95,"Device Specifications","Times New Roman",15,fs_Bold|fs_Underlined
  if usbm_DeviceValid(DeviceNum)
    n = usbm_initports(DeviceNum)
    n = usbm_DirectionA(DeviceNum,255-2^PB_Pin,255) //all outputs but PB+Pin is set as input
    n = usbm_SetBit(DeviceNum,PB_Pin)               //with internal pull-up resistor
    n = usbm_DirectionB(DeviceNum,255,255)
    m = usbm_DeviceSpecs(DeviceNum)
    VID = "Made By:"+hex(tonumber(extract(m,"|",3)))+":"
    MFR = VID+extract(m,"|",4)
    PID = "Device:"+hex(tonumber(extract(m,"|",2)))+":"
    PRD = PID+extract(m,"|",5)
    DID = "Version:"+hex(tonumber(extract(m,"|",1)))
    SER = "S/N:"+extract(m,"|",6)
  else
    MFR = "Made By:" \ PRD = "Device:"\ DID = "" \ SER = "S/N:"
  endif
  xyText 10,125,MFR,"Times New Roman",10
  xyText 10,140,PRD+spaces(20)+DID,"Times New Roman",10
  xyText 10,155,SER,"Times New Roman",10
Return
//================================================
Check_PushButton:
  if !usbm_DeviceValid(DeviceNum) then return
  n = usbm_ReadA(DeviceNum) & (2^PB_Pin)
  if !n //---active low Push button so we act on 0
    PB_Count = PB_Count+1 \ S = ""+PB_Count
    n = usbm_LCDCmd(DeviceNum, 0x1)  //clear LCD
    GoSub Write_LCD
  endif
Return
//=======================================================
Init_LCD:
  if !usbm_DeviceValid(DeviceNum) then return
  n = usbm_InitLCD(DeviceNum, 0x12, 0x13)   //---RW=A1,RS=A2,E=A3,Data=Port B
  n = usbm_SetBit(DeviceNum,0)  \ delay 20  //---RES connected to A0
  n = usbm_ResetBit(DeviceNum,0)\ delay 100 //High-Low-High to reset with 100 ms pulse
  n = usbm_SetBit(DeviceNum, 0) \ delay 100 //give the LCD time to settle
  n = usbm_LCDCmd(DeviceNum, 0x1C)  //---command sequence to setup the LCD
  n = usbm_LCDCmd(DeviceNum, 0x14)  //---this is obtainable from the device
  n = usbm_LCDCmd(DeviceNum, 0x28)  //---specs sheet
  n = usbm_LCDCmd(DeviceNum, 0x4F)  //
  n = usbm_LCDCmd(DeviceNum, 0xE0)  //
  n = usbm_LCDCmd(DeviceNum, 0x1)   //--clear the LCD display
Return
//=======================================================
```

```
Write_LCD: //S is the string to write
  if !usbm_DeviceValid(DeviceNum) then return
  n = usbm_LCDCmd(DeviceNum, 0x1)  //code 0x1 clears the LCD
  if Length(S) == 0 then Return
  for i = 1 To Length(S)
    n=usbm_LCDData(DeviceNum,GetStrByte(S,i)) //send the ascii code of each character
  next
Return
//=======================================================
```

Program5: A Better LCD program with an LED and Pushbutton.

The ***Init_LCD*** and ***Write_LCD*** routines are exactly as before. The ***Display_DllSepcs*** and ***Display_DeviceSpecs*** are similar to the program fragments we developed in Sections 2.1 and 2.3. They go through the specifications string to extract the various sections and then display them in a nice format.

The ***Initialization*** routine makes sure all the edit boxes and buttons are displayed and also calls the subroutines to initialize everything. Notice how the *LCDcmd* and *LCDData* edit boxes are forced to be only integer inputs and how they are also initialized with HEX values to indicate that hex values can be entered instead of decimal values.

The two routines of interest are ***Check_PushButton*** and ***MonitorInputs***. ***Check_PushButton*** reads the U401 Port A but then ANDs the result with the necessary mask to isolate the PB_Pin (A6 in this case). See how this is achieved using bitwise ANDing with the MASK created from the pin number. If the button is pushed we increment the variable *PB_Count* and then we make it into a text so as to send it to the LCD for display by using the ***Write_LCD*** routine.

The ***MonitorInputs*** routine is where all the action occurs. Here, whenever the user clicks a button on the screen a corresponding action is taken. Either to send the text in the text box, or to send a command as specified by the command edit box or to send a single character as specified by its ASCII code in the edit box. Another screen button allows for re-initialization of the LCD.

Another two important actions performed by the routine are the three lines right after the while-statement. The first one sets the U401 pin specified by *LED_Pin* (A5 in this case) to either on or off according the current value of *LED_On* which is inverted every time through the while-loop. The third line calls the ***Check_PushButton*** routine discussed above in order to monitor the state of the pushbutton in the LCD circuit.

3.2.3 Another improvement

Program 5 is very nice and gives the user good control over the LCD and also illustrates how to blink an LED and respond to a real pushbutton all at the same time. However, there are two shortcomings.

First, notice how the LED blinks so fast that it almost does not appear to be blinking. Second, notice what happens when you send text to the LCD. The LED either stops blinking in the off or on state for

a little while. Also notice that when you push the button it is hard to get the count to increase only one at a time, it increments too fast.

We can solve both these problems using a timer. With the timer we can check the button only every few milliseconds to ensure no rapid response for an effective control over the count. Also with the timer we can control how fast the LED blinks. To do this we need to add a subroutine and remove three lines and add a line.

1- In the ***MonitorInputs*** routine delete the three lines right after the "while true" line (lines 01-03).
2- At the end of the ***Initialization*** routine add the following line:

```
addtimer "t1",150 \ onTimer tHandler
```

3- Add the following subroutine

```
tHandler:
  lt = LastTimer()
  n = usbm_WriteABit(DeviceNum,255-2^LED_Pin,2^LED_Pin*LED_On)
  LED_On = !LED_On
  GoSub Check_PushButton
  onTimer tHandler
Return
```

With these changes we have added a timer that causes an interrupt every 150 ms. In the interrupt handler we do the actions of Blinking the LED and responding to the pushbutton.

Notice how these simple changes now make the LED blink at an observable rate, and how it never pauses due to sending text or due to responding to the pushbutton. Also notice how now pushing the button is more controllable for incrementing the number displayed on the LCD.

Study this program and the changes we made to it in detail. We shall use this program again after we learn how to read a thermometer chip using the 1-wire serial communications protocol in the next section.

Note: ***There is a slight problem with the final program that results from the above changes. This problem occurs in a very specific situation that is extremely unlikely to take place. For the sake of keeping the program uncomplicated no attempt has been made here to allow for the very rare possibility of the error happening. It is left as an exercise for you to try and figure out what the problem is, when does it occur, and what to do about it.***

3.3- Using the U4x1 to control 1-Wire devices

1-Wire is an intricate standard protocol for communicating a Master to multiple Slave devices using, well, one wire. In fact, you also need a ground wire too. The communication is a half-duplex bidirectional serial protocol that can be used with numerous devices like clocks, thermometers, relays, medical equipment, security equipment and many more.

The protocol is quite complicated involving timed pulses and so on and so forth. However, thanks to the U4x1 you do not need to be concerned with all the details. All the timing and bit-banging and so on is performed by the U4x1. All you need is to tell it which pin to use as the 1-wire line and then just send bytes and read bytes from any or all the devices on the line. The U4x1 takes care of the bit shifting in and out and of the timing and signal levels.

Nonetheless, if you would like to gain more insight into what the 1-Wire protocol is all about, see this most excellent 17 part video. Also visit the following web site to see the kind of devices available and what the industry is doing with this communications protocol:

http://www.maxim-ic.com/products/1-wire/
http://www.maxim-ic.com/appnotes.cfm/an_pk/1796
http://www.maxim-ic.com/design_guides/en/1_WIRE_PRODUCTS_4.pdf

The pin selected as a 1-wire bus is automatically configured with an internal pull-up resistor of approximately 14 KΩ. During idle bus times it is this resistor that pulls the line high. When the U4x1 transmits a low signal on the bus, it pulls the line low with an open collector. This internal 14 KΩ pull up resistor will suffice for a short bus distance, but you should consider supplementing it with a 10 KΩ resistor external to the U4x1 device. The 10 KΩ resistor would be connected between the 1-wire data line and Vcc (+5V).

RobotBASIC provides 5 functions for using the 1-wire capabilities of the U4x1 devices. That is all you need to be concerned with to achieve communications with a 1 wire device. No timing or pulsing details are of interest. The U4x1 takes care of all that. The functions are:

usbm_Reset1Wire(ne_DeviceNumber,ne_Specs)
usbm_Write1Wire(ne_DeviceNumber,ne_Data)
usbm_Read1Wire(ne_DeviceNumber)
usbm_Write1WireBit(ne_DeviceNumber,ne_BitValue)
usbm_Read1WireBit(ne_DeviceNumber)

Multiple 1-wire busses can exist simultaneously on the U4x1. It is the usbm_Reset1Wire() function that sets the port configuration for a specific pin, as well as designates it as the pin to use for any subsequent read/write operations. The U4x1 devices support 1-wire communication with any 1-wire device. When you select an I/O pin of the U4x1 to use as the connection to a 1-wire device, you change that pin from just being a digital I/O line to a 1-wire bus.

The usbm_Reset1Wire() function configures the line with a 14 KΩ pull up resistor (see above), and issues a reset pulse on that line. The function returns a value that indicates if any device on the bus has indicated a presence pulse. If a device is detected, the returned value is 0. If no device is detected it returns 1. ***Notice this is not the way you might expect. If a device is present the function returns false and if no device is present it returns true.***

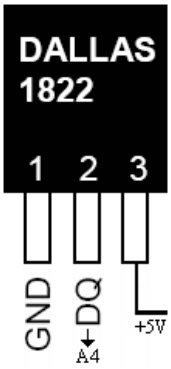
Once you specify which U4x1 pin to use as the 1-wire bus using the usbm_Reset1Wire() function, the usbm_Read1Wire() and usbm_Write1Wire() functions will operate on that pin from that point

onwards. What this means is that you can use all 16 I/O pins on the U4x1 as 16 separate 1-wire busses. Issuing the usbm_Reset1Wire() function is the way to get attention of the 1-wire devices on that bus, prior to using the usbm_Read1Wire() and usbm_Write1Wire() commands to communicate with the 1-wire device. You can use usbm_Reset1Wire() to select and communicate with one line of the U4x1, then use it again to communicate with a different line at a later stage in the program.

Each 1-Wire bus can also have multiple 1-Wire devices on it. You can address them individually by using their ROM serial numbers. The device's documentation contains the details that you need to communicate with it.

The parameter ***ne_Specs*** in the usbm_Reset1Wire() function specifies the I/O pin to be used as the bus. It has to be a number from 0 to 15. 0 being pin A0, 1 being A1 and so on to 7 being A7 and 8 being B0, 9 being B1 and so on until 15 being B7.

3.3.1 A 1-Wire thermometer



All the above is best illustrated with a concrete example. We shall use the DS1822 Digital Thermometer as the demonstration device. This device gives a temperature reading either in Celsius or Fahrenheit. The link above will give you all the data sheet information you will need.

In our application we shall provide power to the DS1822 from the Power pin of the U401 (pin 7). So you must connect the 5V and Gnd pins of the DS1822 to the U401 5V and Ground pins (pins 7 and 9 respectively). The DQ pin should be connected to the A4 pin of the U401.

```
MainProgram:
 DeviceNum = 0    //use device 0
 Thermo_Pin = 4   //use pin A4 for input from the DS1820
 if usbm_DllSpecs() != ""
   if usbm_FindDevices()
     //---there is a device and we will use device DeviceNum
     xyText 10,10,"Temperature =","",15,fs_Bold
     while true
       GoSub Read_ThermoData
       xyText 160,10,Format(Tc,"##0.00°C = ")+Format(Tf,"0.00°F")+spaces(10),"",15
     wend
   else
     print "There are no Devices"
   endif
 else
   print "The USBmicro DLL is not installed"
 endif
end
//================================================
01 Read_ThermoData:
```

```
02   if !usbm_DeviceValid(DeviceNum) then Tc = -40 \ Tf = -40 \return
03   n = usbm_reset1wire(DeviceNum,Thermo_Pin)
04   n =usbm_write1wire(DeviceNum,0xCC) \ n =usbm_write1wire(DeviceNum,0x44)
05   n =usbm_reset1wire(DeviceNum,Thermo_Pin)
06   n =usbm_write1wire(DeviceNum,0xCC) \ n =usbm_write1wire(DeviceNum,0xBE)
07   n =usbm_read1wire(DeviceNum) \ m = usbm_read1wire(DeviceNum)
08   x = 0 \ if(m & 0x80) then x = 0xFFFFF000
09   Tc = (x |(((m & 0x0F)<< 8)+n))*0.0625
10   Tf = Tc*1.8+32
11 Return
//=========================================================
```

Program 6: A Thermometer program using the DS1822 1-Wire digital thermometer.

The main program should be familiar. After initializing some variables and making sure that there is a DLL and then making sure that there is a U4x1 it starts a loop of calling the ***Read_ThermoData*** subroutine and then displaying the temperature data.

With this program we use the A4 pin to be the 1-wire bus pin from the DS1822. This pin is configured automatically as an output pin when writing and as an input pin when reading from the DS1822. This is all performed automatically by the U4x1.

The details of what bytes to send to the DS1822 to get it to perform its actions and how many bytes of data we need to read from it to obtain its data is something you should read from the device's specifications sheet.

The DS1822 gives a temperature reading that is 12 bits of resolution (really 11 with the 12^{th} bit being a sign bit). The format of the reading is in fact a 16 bit 2's compliment format of the temperature. If the temperature is negative all the top 5 MSbits will have a 1 and the lower 11 LSbits will be 2's compliment. If the temperature is positive then the 5MSbits will be 0 and the 11 LSbits will contain the temperature binary value. Each bit represents 0.0625°C so we have to multiply the actual decimal number with 0.0625 to get the true temperature value.

The work of reading the temperature from the DS1822 is performed in the ***Read_ThermoData*** subroutine. It is performed in two stages. We reset the 1-wire pin (line 03) so as to send a pulse to the device and then we send it a two byte command (line 04) to get it to read the temperature and store it in its memory. We then reset the pin again (line 05) so as to start the DS1822 and then we send it a two byte command (line 06) to get it to send the data. We then read the result (line 07) as two bytes with the LSByte being stored in *n* and the next byte stored in *m*.

Then we need to convert the two bytes that are now stored in two integers to be one 32 bit number that would be a 32 bit 2's compliment representation of the temperature. This is the function of lines 08 and 09.

On line 10 we also store the Fahrenheit temperature by converting the Celsius value. The end result is that the two variable *Tc* and *Tf* will contain the Celsius and Fahrenheit temperature reading.

If we know that the temperature is not ever going to go negative we could have performed a simpler operation to get the temperatures reading. Delete line 08 and change 09 to:

```
Tc = (((m&7)<<8)+n) * 0.0625
```

3.3.2 A 1-Wire thermometer with LCD display

Let's now write a program that will:

1- Read the temperature
2- Display it on the PC screen and LCD as Celsius.
3- Keep toggling an LED
4- A pushbutton is used to toggle between Fahrenheit and Celsius.

The program will not have a fancy GUI so as to keep it as simple as possible, only illustrating how we can combine all that we have learnt up to this point.

```
MainProgram:
   GoSub Initialization
   If AllOk
     xyText 10,20,"Temperature =","Time new roman",15,fs_Bold
     while true
        if !Fahrenheit
           S = Format(Tc,"##0.00°C")
        else
           S = Format(Tf,"##0.00°F")
        endif
        xyText 170,20,S+spaces(10),"Times new roman",15,fs_Bold
        S = "Temperature = "+S+spaces(10)
        GoSub Write_LCD
     wend
   endif
end
//================================================
Initialization:
  DeviceNum = 0      //use device 0
  Thermo_Pin = 4     //use pin A4 for input from the DS1820
  LED_Pin = 5 \ LED_On = 1 \  PB_Pin = 6
  Fahrenheit = false \ AllOk = false
  Tc = 25 \ Tf = 77
  if usbm_DllSpecs() == "" then  print "The USBmicro DLL is not installed" \ return
  if !usbm_FindDevices() then print "There are no Devices" \ return
  if usbm_DeviceValid(DeviceNum)
    n= usbm_initports(DeviceNum)
    n= usbm_DirectionA(DeviceNum,255-2^PB_Pin,255) //all output but PB_Pin is set as
    n= usbm_SetBit(DeviceNum,PB_Pin)           //input with internal pull-up resistor
    n= usbm_DirectionB(DeviceNum,255,255)
    GoSub Init_LCD
    addtimer "t1",150 \ onTimer tHandler
    AllOk = true
  else
    print "Device is invalidated"
  endif
```

```
Return
//================================================
tHandler:
  lt = LastTimer()
  n = usbm_WriteABit(DeviceNum,255-2^LED_Pin,2^LED_Pin*LED_On)
  LED_On = !LED_On
  GoSub Check_PushButton
  GoSub Read_ThermoData
  onTimer tHandler
Return
//================================================
Check_PushButton:
  if !usbm_DeviceValid(DeviceNum) then return
  n = usbm_ReadA(DeviceNum) & (2^PB_Pin)
  if !n then Fahrenheit = !Fahrenheit  //active low Push button so we act on 0
Return
//=========================================================
Read_ThermoData:
  if !usbm_DeviceValid(DeviceNum) then Tc = -40 \ Tf = -40 \return
  n = usbm_reset1wire(DeviceNum,Thermo_Pin)
  n = usbm_write1wire(DeviceNum,0xCC) \ n = usbm_write1wire(DeviceNum,0x44)
  n = usbm_reset1wire(DeviceNum,Thermo_Pin)
  n = usbm_write1wire(DeviceNum,0xCC) \ n = usbm_write1wire(DeviceNum,0xBE)
  n = usbm_read1wire(DeviceNum) \ m = usbm_read1wire(DeviceNum)
  x = 0 \ if(m & 0x80) then x = 0xFFFFF000
  Tc = (x | ( ( (m & 0x0F)<< 8 )+ n) )*0.0625
  Tf = Tc*1.8+32
Return
//=========================================================
Init_LCD:
  if !usbm_DeviceValid(DeviceNum) then return
  n = usbm_InitLCD(DeviceNum, 0x12, 0x13)   //---RW=A1,RS=A2,E=A3,Data=Port B
  n = usbm_SetBit(DeviceNum,0)  \ delay 20  //---RES connected to A0
  n = usbm_ResetBit(DeviceNum,0)\ delay 100 //High-Low-High to reset with 100 ms pulse
  n = usbm_SetBit(DeviceNum, 0) \ delay 100 //give the LCD time to settle
  n = usbm_LCDCmd(DeviceNum, 0x1C)  //---command sequence to setup the LCD
  n = usbm_LCDCmd(DeviceNum, 0x14)  //---this is obtainable from the device
  n = usbm_LCDCmd(DeviceNum, 0x28)  //---specs sheet
  n = usbm_LCDCmd(DeviceNum, 0x4F)  //
  n = usbm_LCDCmd(DeviceNum, 0xE0)  //
  n = usbm_LCDCmd(DeviceNum, 0x1)   //--clear the LCD display
Return
//=========================================================
Write_LCD: //S is the string to write
  if !usbm_DeviceValid(DeviceNum) then return
  n = usbm_LCDCmd( DeviceNum, 0x2) //code 0x2 to Go to 1st char position
  if Length(S) == 0 then Return
  for i = 1 To Length(S)
    n=usbm_LCDData(DeviceNum,GetStrByte(S,i))//send the ascii code of each character
  next
Return
//=========================================================
```

Program 7: 1-Wire Thermometer and LED and Pushbutton and LCD.

You should be familiar with all the program's aspects. The only changes are in the ***Write_LCD*** and ***Check_PushButton*** subroutines. In ***Write_LCD*** we use the command code 0x2 in place of 0x1 to return the cursor to the beginning of the line instead of clearing it. This helps to avoid flicker.

The ***Check_PushButton*** subroutines checks to see if the pushbutton is down and if it is it toggles a variable Fahrenheit which is used to indicate whether the display should be Fahrenheit or Celsius. Notice how when you push the button the display is changed on both the PC screen and LCD.

The rest of the program is the same as the other programs you have already seen.

3.4- Using the U4x1 to control SPI devices

The Serial Peripheral Interface (SPI) bus is a full duplex bidirectional ***synchronous*** serial communications standard protocol. Devices communicate in master/slave mode where the master device initiates the data exchange. Multiple slave devices can be connected to one Master using the same 3 lines with an additional individual slave select (often called chip select) line from the Master to each slave. In addition there is, of course, the ground line.

A master controls a slave using the following lines:

1- Ground
2- Clock (called SCLK or SCK or CLK and is output from Master).
3- Data Out (called MOSI output from the Master connects to the SIMO input of the Slave, which may also be labeled SDI, DI, SI).
4- Data In (called MISO input to the Master connects to the SOMI output of the Slave which may also be labeled SDO, DO, SO).
5- Slave Select [1 line per slave] (called SS output from the master connects to the Slave's Chip Select and may be labeled nCS, CS, nSS, STE and is often active low).

The reason it is called synchronous is because bits are transferred in synch with a clock signal. Also if multiple slaves are to be able to connect to the same CLK, MOSI, and MISO lines then all of them will have to have HiZ lines when they are not the selected device (usually when CS is high).

If you would like to find out more about this protocol see the following links:

http://www.intersil.com/data/an/AN1340.pdf
http://www.mct.net/faq/spi.html
http://www.embedded.com/columns/beginerscorner/9900483
http://ww1.microchip.com/downloads/en/devicedoc/spi.pdf
http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

The protocol can be quite complex in its sequence of actions and since it is full duplex the Master has to be able to listen as well as send simultaneously. Fortunately all this is taken care of by the U4x1. All we need is to use the following functions:

usbm_InitSPI(ne_DeviceNumber,ne_Specs)
usbm_SPIMaster(ne_DeviceNumber,se_DataBytes)
sbm_SPISlaveRead(ne_DeviceNumber)

usbm_SPISlaveWrite(ne_DeviceNumber,se_DataBytes)

The U4x1 can be either a slave or a master. However, most devices normally act as slaves and therefore the U4x1 will more often be used as a Master.

To use the U4x1 you need to decide if you want to use it as a slave or as a master. You then configure the U4x1 for that mode using the function usbm_InitSPI() where the parameter ***ne_Specs*** contains the necessary codes for the various configurations (we will see this later).

If you are going to use the U4x1 as a master you use the usbm_SPIMaster() function to send commands to slave devices and at the same time to receive data from them (we will show how later). You also need to make sure that the CS line is driven to the require state to activate the required slave (usually low).

If you are going to use the U4x1 as a slave then you will have to monitor a CS line and when it is driven to the right sate (usually low) you use usbm_SPISlaveRead() to read any input from the Master and then act upon it then send out any responses with the usbm_SlaveWrite() function,

In this document we will show how to use the U4x1 as a master. For details on using it as a slave refer to the USBmicro documentation.

3.4.1 Initializing the U4x1 SPI system

The function usbm_InitSPI() is used to configure the U4x1 to be a slave or a Master. It also configures the frequency of the clock signal as well as whether data is read on a rising edge or falling edge and what is the idling state of the clock.

ne_Specs is an integer that holds a byte value. The 6 LSBits of the byte are set according to the information below. The 2 MSbits will always be 00.

Bits 0 and 1 configure the clocking speed as follows
- 00 = 2 Megabits per second
- 01 = 1 Megabits per second
- 10 = 500 Kilobits per second
- 11 = 62500 Bits per second.

Bits 2 and 3 set the clock phasing
- 00 = idle low, data transfer on falling edge (SPI mode 1)
- 01 = idle low, data transfer on rising edge (SPI mode 0)
- 10 = idle high, data transfer on rising edge (SPI mode 3)
- 11 = idle high, data transfer on falling edge (SPI mode 2)

Bits 4 and 5 set the SPI system as slave, master or off
- 00 = SPI system is turned off

01 = Master
10 = Slave
11 = this bit combination should not be used

To set the U4x1 to act as an SPI Master with a clock frequency of 500 Kilobits per second and to have data being transferred on a rising edge with the clock idling high we would say

```
n = usbm_InitSPI(DeviceNum,0x00011010)
```

So, which I/O pins will be used? Remember, that to act as a master we need 4 I/O lines. MOSI (data output from the master to the slave), MISO (data input to the master from the slave), SCK (clock line output from the master to the slave) and SS (slave select line to activate/deactivate the slave [output from the master]).

When you use the usbm_InitSPI() function it also configures the following pins:
A5 as an output (MOSI)
A6 as an input (MISO)
A7 as an output (SCK)

You will also need to manually configure a 4th pin as an output pin to act as the SS. You can choose any remaining I/O pin for that. However it is advisable to choose A4; we would do that with a separate call to the function usbm_DirectionA().

When you turn off the SPI system with a call to usbm_InitSPI() with bits 5 and 4 of the parameter ***ne_Specs*** as 00, the U4x1 will ***not*** reconfigure the pins. They will remain as output (A5,A7 and A4 if you selected that as SS) and as input (A6). So if you desire to have them in a different mode then you need to use usbm_DirectionA() to do so.

Now we are ready to use the U4x1 device to do SPI Master control over an SPI slave chip. But before we do that we need to read the slave device's specifications to figure out what clocking speed and phase it requires.

Another concern is Bit-Ordering. The U4x1 sends the bits with the LSBit First order. So a byte 0%00110101 will be sent as 1 then 0 then 1 then 0 then 1 and 1 then 0 and 0. That is bit 0 is sent first then bit 1 and so on. So if a device expects the other way then you cannot use the device. Additionally the U4x1 sends a byte then waits for a response and then sends the next byte and then waits for a response. If the device requires multiple bytes before responding or it requires nibbles or a non byte number of bits then you cannot use the U4x1 to control these devices.

3.4.2 The U4x1 as an SPI Master

To use the U4x1 as an SPI master after having initialized it, you use the function usbm_SPIMaster(). The parameter s***e_DataBytes*** must be a string buffer. To learn more about string buffers read Section 3 in the document RobotBASIC_Networking.PDF. This document gives advanced information on how

to create and use a string buffer. However, for the purposes of the SPI system this string buffer will not be any longer than 6 bytes and thus we can use it in a simple way as we will show below.

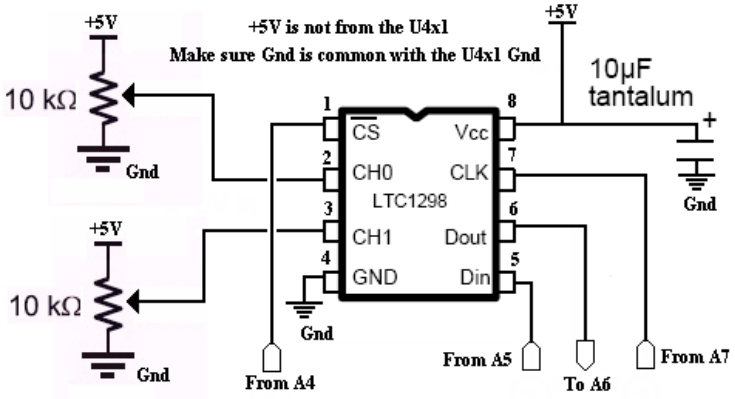
To create ***se_DataBytes*** just use the ToByte() finction to create a byte from an integer and add them together. So if for instance you want the buffer to have three bytes with the values 20, 56, 233 then you would do `sBuff = toByte(20)+toByte(56)+toByte(233)`. Then you can send this buffer with the usbm_SPIMaster() function by saying `rBuff = usbm_SPIMaster(DeviceNum, sBuff).`

The usbm_SPIMaster() also returns a string buffer which contains the responses from the slave to the sent bytes. In the code line above *rBuff* would end up containing the response bytes. However, these response bytes will be preceded with a byte. So the returned bytes will always start at the second byte onwards. This is because the first byte in the return buffer is used by the U4x1 to put its own information. ***Just remember that the values in the retuned buffer always start on the second byte of the buffer.***

We can get the values of these bytes using the GetStrByte() function. So for example if we know that the string buffer *rBuff* has 3 bytes of data (1st byte from the U4x1 and 2 from the slave), and we want the 2nd byte we would say `Val=GetStrByte(rBuff,2)`. *Val* will end up containing an integer with a value equal to the value of the second byte in the string buffer. ***Remember that the 1st byte is <u>not</u> data from the slave***.

3.4.3 An Analog To Digital Converter (LTC1298 ADC) application

Figure 6: Connecting a U4x1 and an LTC1298



As a tangible example we shall use the Analog To Digital Converter (ADC) chip LTC1298. This is a 12-bit 2 channel ADC which is also an SPI slave. Read the specifications in the above link and study the program below in light of these specifications and the information in the previous sections.

After checking for a device the program sets up the SPI system according to what is needed by the LTC1298 and then turns it off by pulling SS (/CS) high. The loop then continuously reads Channel 0 then Channel 1 then converts the results to voltages by multiplying the results by the resolution. It then displays the values.

```
MainProgram:
  GoSub Initialization
  while true
     GoSub Read_ADC
     xyText 200,100,Format(V1,"0.00V"),"",15
     xyText 200,120,Format(V2,"0.00V"),"",15
```

```
  wend
end
//================================================
Initialization:
  DeviceNum = 0    //use device 0
  SS_Pin = 4   //use pin A4 as Slave Select
  if usbm_DllSpecs()=="" then Print "The USBmicro DLL is not installed" \ End
  if !usbm_FindDevices() then print "There are no Devices" \ End
  if !usbm_DeviceValid(DeviceNum) then print "Device is invalidated" \ End
  n = usbm_InitPorts(DeviceNum)
  n=usbm_DirectionA(DeviceNum,2^SS_Pin,2^SS_Pin)  //set SS_Pin to output
  n = usbm_InitSPI(DeviceNum,0%00011011) // 01=master,10=SPI mode3,11=62500 bits/sec
  n = usbm_SetBit(DeviceNum,SS_Pin)  //disable the slave SS=high initially
  setcolor white,black
  xyText 100,70," LTC1298 ","",15,fs_Bold
  xyText 50,100,"Channel[0]:","",15,fs_Bold
  xyText 50,120,"Channel[1]:","",15,fs_Bold
  setcolor black,white
  onAbort AbortHandler
Return
//================================================
AbortHandler:
  n = usbm_InitSPI(DeviceNum,0x00)  //trun SPI off
Terminate
//================================================
Read_ADC:
   V1 = 0 \ V2 = 0
   if !usbm_DeviceValid(DeviceNum) then return
   //-----read channel 0
   n = usbm_ResetBit(DeviceNum,SS_Pin)
   m = usbm_SpiMaster(DeviceNum,toByte(1)+toByte(0x80)) //codes for channel 0 and MSB first
   n = usbm_SetBit(DeviceNum,SS_Pin)
   v = ((getstrbyte(m,2)&0x0F)<<8)+getstrbyte(m,3)
   V1 = 5.0*v/0xFFF
   //-----read channel 1
   n = usbm_ResetBit(DeviceNum,SS_Pin)
   m = usbm_SpiMaster(DeviceNum,toByte(1)+toByte(0xC0)) //codes for channel 1 and MSB first
   n = usbm_SetBit(DeviceNum,SS_Pin)
   v = ((getstrbyte(m,2)&0x0F) << 8)+getstrbyte(m,3)
   V2 = 5.0*v/0xFFF
Return
//================================================
```

Program 8: An SPI ADC.

The only new thing in this program is the OnAbort statement in the ***Initialization*** subroutine. This makes it so that when the user of the program stops it by closing the Terminal screen the program will go to the ***AbortHandler*** subroutine. This subroutine ensures that the SPI system is turned off and then ends the program. This is necessary so that you won't have to unplug the U401 before you want to use it next time in another program.

In the ***Read_ADC*** routine you notice that we send two bytes to the LTC1298. These bytes are to tell it what mode and which channel to use to do the voltage conversion. You can get information about what these bytes should be from the LTC1298 spec sheet.

Between reading each channel the SS (/CS) pin is set high to disable the LTC1298 but this is also a necessary action according to the LTC1298 spec sheet. Since it is what makes the LTC1298 become ready for the next command after it sends the results of its current command.

Also notice that the LTC1298 returns its reading in two bytes in the buffer which is the return value from the usbm_SPIMaster() function (***starting with 2nd byte onwards***). So we need to obtain these two bytes and create the actual ADC reading from them. Notice that the MSByte is the first byte. This is how the LTC298 was told to send the data, we also need to AND it with 0xF because only the lower 4 bits hold a value the top nibble does not have significance. Remember the LTC1298 is a 12-Bit ADC, so the 3rd byte in the buffer is used to be the lower 8 bits of the reading and the lower 4 bits of 2nd byte in the buffer are used to create the next 4 bits of the reading.

Finally we multiply the reading by 5.0/0xFFF. Why 0xFFF and 5.0? The supply voltage to the LTC1298 is also its reference voltage and since its resolution is 12 bits (0xFFF) then the resolution of the reading is 5.0/0xFFF Volts and we use that as a multiplier to convert the reading to a voltage value.

3.5- Using the U4x1 to control stepper motors

One of the many exciting features of the U4x1 is its ability to control 2 stepper motors, independently, simultaneously, and in various stepping modes.

The U4x1 is told the stepping rate, the stepping mode, the direction and number of steps. It then carries out the necessary control and all signaling to control the stepper motor as desired, without any further monitoring from RobotBASIC.

You can have up to two motors turning at different speeds in different directions and in different stepping modes. If the stepping is continuous the U4x1 will carry out all the controls without the necessity for any further monitoring from RobotBASIC. Additionally, you can change the direction, stop or change the speed of the motors at any time.

Note: The U4x1 provides the 4 bits (for each motor so 8 bits altogether) that are the stepping signal needed to activate a ***DRIVER*** (e.g. ULN2803A) that actually provides the voltage and current that will excite the coils of the stepper motor. ***You must not connect the I/O pins of the U4x1 directly to the motor.*** This will overload the U4x1 and the PC's USB port. **You must use a different power supply and a driver** (the Gnd is of course common). ***Even better would be to use a power isolation circuit with something like the 4N33 to totally isolate the U4x1 and PC from the motors which is usually necessary to avoid the EMF feedback and noise usually related to Electromagnetic devices like motors and solenoids and so forth.***

For the purposes of the programs below you do not need a motor or any connections. However, you may wish to connect the I/O lines to LEDs so that you can observe the signaling. The programs will also display the bit values on the screen, so you do not even need the LEDs.

3.5.1 Controlling a stepper motor

The function **usbm_Stepper(ne_DeviceNumber,se_DataSpecs)** is what tells the U4x1 to start the control action. One motor will be controlled on A0 to A3 and the other motor on A4 to A7. So the lower nibble of Port A is used to provide the 4 bits signal to control the first motor and the upper nibble of Port A is used to control the other motor.

Note: This function does not set Port A to output. You have to do it using usbm_DirectionA(). The U4x1 can control the direction of the turn of the motor which for reference we shall call right or left, however, the actual direction depends on the order of the line connections and orientation of the motor.

The parameter ***se_dataSpecs*** is a 7 byte string (buffer) that contains the control information, as follows:

Byte Number (0 is first byte and so on)	Significance
0	The motor to be affected. 1 (controlled by A0-3) or 2 (controlled by A4-7)
1	On =1 turns the motor on. Off = 0 turns the motor off
2	Direction 0 = right, 1 = left
3	Stepping type 0=half step, 1= Full Step or Wave stepping
4	Initial signal state has to be either 3 (0%11) for Half or Full step, or 1 for Wave stepping. **Note:** Initialization can only take place if byte 1 =0.
5	Delay between steps ranges from 0 to 255 and is in 128µsecs intervals. So a value of 100 is a delay between steps of 12.8 millisecs. The turn rate will then depend on the number of steps per cycle and this number.
6	The number of steps to turn. 0 means no turning, 255 means continuous turning. Any other number will be the number of steps performed.

Table 2: Stepper motor control bytes.

Form Table 2, we can see that to make the U4x1 start the first motor with the half stepping mode and at 1 step every 25.6 milliseconds to the right (relative) then we need to do the following:

```
S = toByte(1)+toByte(0)+toByte(0)+toByte(0)+toByte(3)+toByte(200)+toByte(0)
n = usbm_Stepper(DeviceNum,S)  //this will initialize the motor and notice
                               //the motor is off since byte 1 = 0
S = BufferWriteB(S,1,on) //make byte 1 = 1 i.e. turn motor on
S = BuffWrite(S,6,255)   //make byte 6 = 255 i.e. continuous turning
n = usbm_Stepper(DeviceNum,S)
```

To stop the motor later on:

```
S = BuffWrite(S,6,0)   //make byte 6=0 i.e. no more stepping
n = usbm_Stepper(DeviceNum,S)
```

3.5.2 A simple program

To try this program ***you do not need a motor.*** You can, instead, connect A0-A3 to LEDs like in previous experiments. This will enable you to observe the signal. Also, the program will read the status of the A0-3 pins and display the value on the PC screen so in reality you do not even need the LEDs; but having the LEDs is good. ***Notice that even though a pin is an output pin you can read its status***.

```
MainProgram:
  GoSub Initialization
  if AllOk
     while true
       n=usbm_ReadA(DeviceNum)
       xytext 10,10,bin((n >>((Channel-1)*4))&0x0F,4),"",20,fs_Bold
       waitkey k
       GoSub Stepper
     wend
  endif
End
//============================================================
Initialization:
   DeviceNum = 0 \ AllOk = false
   if usbm_DllSpecs() == "" then print "The USBm.Dll is not installed" \return
   if !usbm_FindDevices() then print "There are no U4x1 devices" \return
   Channel     = 1
   Enable      = 0
   Direction   = 0  //0=right  1=left
   movetype = 0 //---0 = half, 1 = full, 2 = wave
   if movetype = 0 then Type = 0 \ Initial = 3
   if movetype = 1 then Type = 1 \ Initial = 3
   if movetype = 2 then Type = 1 \ Initial = 1
   Rate        = 255
   Steps       = 1
   S = toByte(Channel)+toByte(Enable)+toByte(Direction)
   S = S+toByte(Type)+toByte(Initial)+toByte(Rate)+toByte(0)
   if usbm_DeviceValid(DeviceNum)
      n = usbm_DirectionA(DeviceNum,0xF,0xF)  //SetPort A0-A3 to output
      n = usbm_Stepper(DeviceNum,S)           //Initialize motor
      S = BuffWriteB(S,1,1)                   //
      n = usbm_Stepper(DeviceNum,S)           //turn it on but no steps
      AllOk = true
   endif
Return
//============================================================
Stepper:
  S = BuffWriteB(S,6,Steps)
```

```
  n = usbm_Stepper(DeviceNum,S)  //do the steps
Return
//============================================================
```

Program 9: Controlling A Stepper Motor.

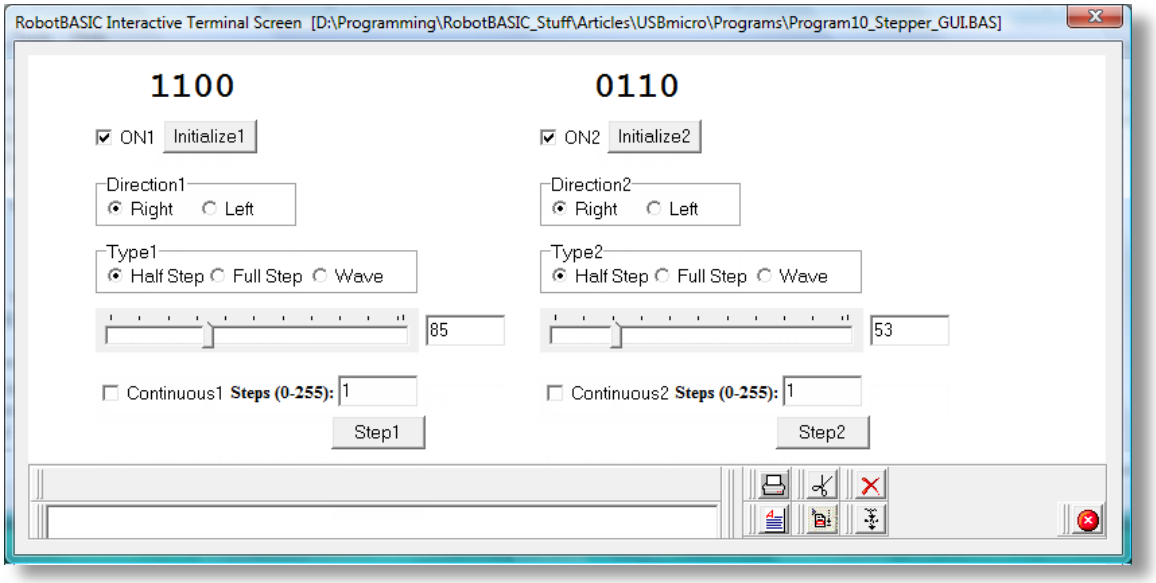
This program is kept simple to illustrate the actions. You can change any of the parameter in the ***Initialization*** routine to get different effects. However, see the next program for a better user interface.

3.5.3 A better program with GUI

This program is complicated in so far as the user interface is concerned and in how it formulates the ***se_DataSpecs*** parameter according to the user input. It also employs the ***Event Driven*** model as well as ***Timers***. Additionally, the program provides control over both motors so that you can see how they can be controlled totally independently and simultaneously and the program can just set them and go on with doing other tasks.

No explanation of the program will be given. It should not be too hard to read. For information on the GUI aspects and event driven aspects refer to the RobotBASIC help file.

Many places of the program could have been made more efficient, and the screen could have been made fancier, but that would have complicated it further, so whenever possible the simpler options were chosen for the sake of clarity.




```
ManinProgram:
  GoSub Initialization
  if AllOk
     while true
       vn=usbm_ReadA(DeviceNum)
       xytext 90,60,bin(vn&0x0F,4),"",20,fs_Bold
```

```
       xytext 420,60,bin(vn >> 4,4),"",20,fs_Bold
     wend
  endif
End
//===============================================================
Initialization:
   DeviceNum = 0 \ AllOk = false
   if usbm_DllSpecs() == "" then print "The USBm.Dll is not installed" \return
   if !usbm_FindDevices() then print "There are no U4x1 devices" \return
   Dim S[3]
   S[1] = toByte(1)+toByte(0)+toByte(0)+toByte(0)+toByte(3)+toByte(255)+toByte(0)
   S[2] = BuffWrite(S[1],0,2)
   if usbm_DeviceValid(DeviceNum)
      n = usbm_DirectionA(DeviceNum,255,255)  //SetPort A to output
      AllOk = true
      for i=1 to 2
         n    = usbm_Stepper(DeviceNum,S[i])           //Initialize motor
         S[i] = BuffWriteB(S[i],1,1)                   //
         n    = usbm_Stepper(DeviceNum,S[i])           //turn it on but no steps
         AddCheckbox "ON"+i,50+330*(i-1),105,"",1
         Addbutton   "Initialize"+i,100+330*(i-1),100,70
         Addbutton   "Step"+i,225+330*(i-1),320,70
         AddRBGroup  "Direction"+i,50+330*(i-1),140,150,40,2,"Right"+crlf()+"Left"
         AddRBGroup  "Type"+i,50+330*(i-1),190,240,40,3,"Half Step"+crlf()+"Full Step"+crlf()+"Wave"
         SetRBGroup  "Direction"+i,1 \ SetRBGroup "Type"+i,1
         AddSlider   "Speed"+i,50+330*(i-1),240,240,0,255
         AddEdit     "Speed"+i,295+330*(i-1),245,60,0,0 \ ReadOnlyEdit "Speed"+i
         AddCheckBox "Continuous"+i,55+330*(i-1),295,"",0
         AddEdit     "Steps"+i,230+330*(i-1),290,60,0,"1" \ IntegerEdit "Steps"+i
         xyText      150+330*(i-1),295,"Steps (0-255):","Times new roman",10,fs_Bold
      next
      AddTimer "t1" \ SetTimer "t1",off \ AddTimer "t2" \ SetTimer "t2",off
      onCheckBox cbHandler \ onSlider slHandler
      OnEdit edHandler \ onRBGroup rbgHandler
      onButton bHandler \ onTimer tHandler
   endif
Return
//===============================================================
Stepper:
  n = usbm_Stepper(DeviceNum,S)  //do the steps
Return
//===============================================================
tHandler:
   lt = LastTimer()
   ltn = Right(lt,1)
   RenameButton "Stop"+ltn,"Step"+ltn
   SetTimer lt,off
   onTimer tHandler
Return
//===============================================================
bHandler:
   lb = LastButton()
   lbn = tonumber(Right(lb,1))
   if left(lb,10) == "Initialize"
      S[lbn] = BuffWriteB(S[lbn],1,0) \ S[lbn] = BuffWriteB(S[lbn],6,0)
      n = usbm_Stepper(DeviceNum,S[lbn])           //Initialize motor
      S[lbn] = BuffWriteB(S[lbn],1,1)              //
      n = usbm_Stepper(DeviceNum,S[lbn])           //turn it on but no steps
   elseif left(lb,4) == "Step"
      Steps = ToNumber(GetEdit("Steps"+lbn),0)
```

```
      Rate  = 255-GetSliderPos("Speed"+lbn)
      S[lbn] = BuffWriteB(S[lbn],6,Steps)
      S = S[lbn] \ GoSub Stepper
      RenameButton "Step"+lbn,"Stop"+lbn
      SetTimerPeriod "t"+lbn,128*Rate*Steps/1000
      SetTimer "t"+lbn,on
   elseif left(lb,4) == "Stop"
      SS = S[lbn]
      S[lbn] = BuffWriteB(S[lbn],1,0) \ S = S[lbn] \ GoSub Stepper
      S[lbn] = SS \  SetTimer "t"+lbn,off
      RenameButton "Stop"+lbn,"Step"+lbn
   endif
   onButton bHandler
Return
//================================================================
rbgHandler:
   lrbg = LastRBGroup()
   lrbn = tonumber(Right(lrbg,1))
   if left(lrbg,9) == "Direction"
      Direction = GetRBGroup(lrbg)-1
      S[lrbn] = BuffWriteB(S[lrbn],2,Direction)
      S = S[lrbn] \ GoSub Stepper
   elseif left(lrbg,4) == "Type"
      movetype = GetRBGroup(lrbg)-1
      if movetype = 0 then Type = 0 \ Initial = 3
      if movetype = 1 then Type = 1 \ Initial = 3
      if movetype = 2 then Type = 1 \ Initial = 1
      SetCheckBox "Continuous"+lrbn,0  //if continuous turn it off
      S[lrbn] = BuffWriteB(S[lrbn],1,0) \ S[lrbn] = BuffWriteB(S[lrbn],3,Type)
      S[lrbn] = BuffWriteB(S[lrbn],4,Initial) \ S[lrbn] = BuffWriteB(S[lrbn],6,0)
      n = usbm_Stepper(DeviceNum,S[lrbn])           //turn motor off and Initialize it
      S[lrbn] = BuffWriteB(S[lrbn],1,1)                 //
      n = usbm_Stepper(DeviceNum,S[lrbn])           //turn back on
   endif
   onRBGroup rbgHandler
Return
//================================================================
edHandler:
   led = LastEdit()
   if left(led,5) == "Steps"
      Steps = Limit(ToNumber(GetEdit(led),0),0,254)
      SetEdit led,Steps \ edn =EditChanged(led)
   endif
   onEdit edHandler
Return
//================================================================
cbHandler:
   lcb = LastCheckBox()
   lcbn = tonumber(right(lcb,1))
   if left(lcb,10) == "Continuous"
      cbn = GetCheckBox(lcb)
      EnableEdit "Steps"+lcbn,!cbn \ EnableButton "Step"+lcbn,!cbn
      S[lcbn] = BuffWriteB(S[lcbn],6,255 * cbn)
      S = S[lcbn] \ GoSub Stepper
   elseif left(lcb,2) == "ON"
      S[lcbn] = BuffWriteB(S[lcbn],1,GetCheckBox(lcb))
      S = S[lcbn] \ GoSub Stepper
   endif
   onCheckBox cbHandler
Return
```

```
//=============================================================
slHandler:
   lsl = LastSlider()
   Rate = 255-GetSliderPos(lsl)
   SetEdit lsl,GetSliderPos(lsl)
   lsn = tonumber(Right(lsl,1))
   S[lsn] = BuffWriteB(S[lsn],5,Rate)
   if GetCheckBox("Continuous"+lsn) then S = S[lsn] \ GoSub Stepper
   onSlider slHandler
Return
//=============================================================
```

Program10: Stepper Motor Control With A GUI.

4- An Internet Project

One of the uses of the U4x1 can be as a data collection and instrument control device. RobotBASIC has the ability to communicate PCs over a network, either the LAN or even across the internet. Read the document RobotBASIC_Networking.PDF for more details on how to use the networking facilities in RobotBASIC.

As an illustration for the principles of using the networking abilities of RobotBASIC combined with the interfacing and instrumentation ability of the U4x1 we shall develop a small and simple project. The project will demonstrate the principles but will not be complex or rigorous.

We shall combine the 1-Wire thermometer with the stepper motor to create a small project to implement a bidirectional data exchange system.

There will be two PCs, we will call them Reader and Controller. Reader is connected to the U4x1 which is connected to a stepper motor and a Ds1822. 'Reader' requests temperature readings from the U4x1 and then displays them on its screen and also sends them to Controller. 'Controller' will also have a slider that allows a user to select a speed for the stepper motor between -255 and 255. Negative numbers will make the motor go to the left and positive numbers to the right. This speed and direction selection will then be transmitted to 'Reader' which will bring about the speed change using the U4x1 and will also display it on its screen. 'Reader' will also read the stepping signal from the lower nibble of port A and will display it on the screen as well as send to the Controller, which will also display it on its screen.

The above actions are not very complex, but they do illustrate sending and receiving on both sides. They also illustrate user interfacing and instrumentation.

For example instead of having the user select the speed for the stepper motor you could implement some control algorithm (e.g. PID) to decide what the speed has to be in response to the difference between the reported temperature and a desired set level. This will effectively make the system a feedback mechanism but with the link between the actuators and sensors and the controller being over the internet or LAN instead of direct connections.

As far as connecting the U4x1 with the stepper, we will again use LEDs which are not really needed either -their states will be read from the port and displayed on the screen. A0 to A3 will be used, which is motor 1. Also B7 will be use to be the 1-wire line to connect to the DS1822.

4.1- The Reader Program

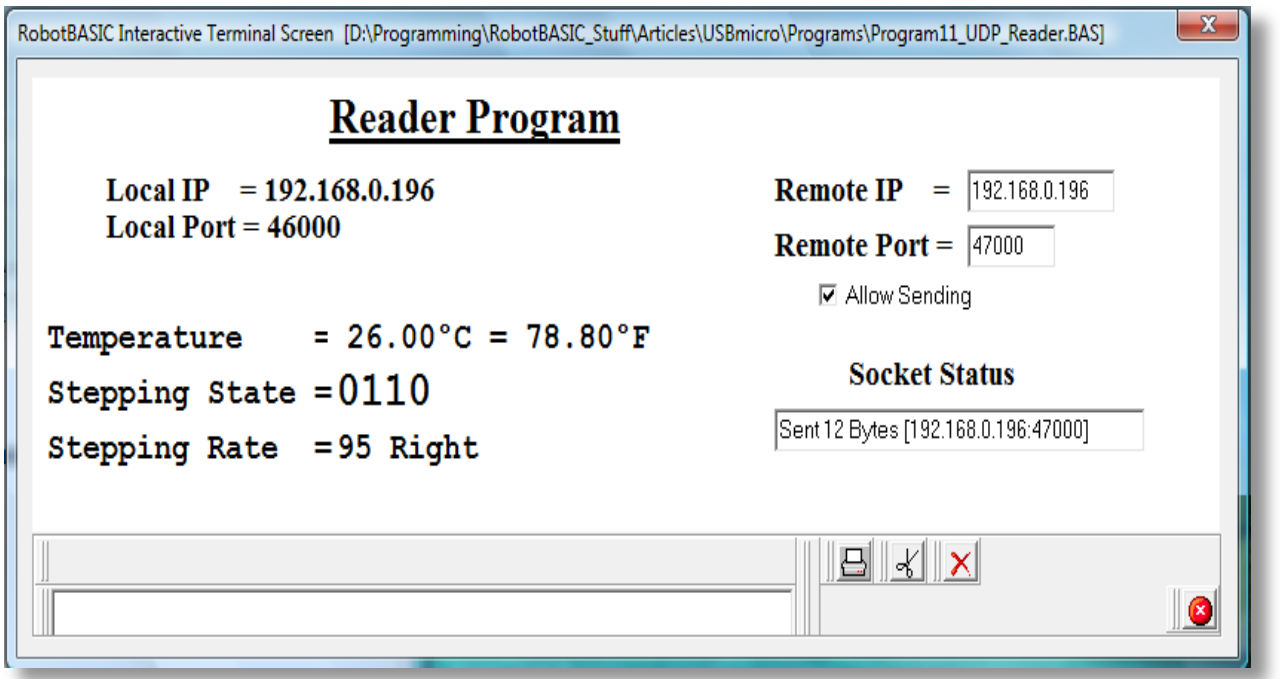


Figure 7: UDP Reader Program

Figure 7 is a screen shot of the Reader program. Notice the Remote IP and Port fields. Before you can allow sending data from the program to the controller program ensure that these two fields are correctly filled. Also make sure that the Controller program is running. Then check the check box.

Warning!!! ***You must not allow sending from the program to a non-existing Port on the same IP as the Reader program. So before you allow sending you must ensure the other side is running. Likewise when you want to close the programs make sure that both have stopped sending by un-checking the check box.***

The remote IP and port fields of the Reader program should correspond to the Local IP and Port of the Controller Program if you are running both programs within the LAN. If you are running them across the Internet then the remote IP should reflect the IP of the router of the LAN where the Controller is running. See Appendix B of the RobotBASIC_Networking.PDF document.

Notice that the Temperature is as read from the DS1820 through the U401 attached to the Reader program's PC. Furthermore, the Stepping State is also read from lower nibble of Port A as it is being set by the U401 to do the stepping.

The Stepping Rate is as is ***received*** over the UDP from the Controller program. This received value is displayed and also is used to set the stepper motor's rate.

The Temperature and Stepping state are also sent (if sending is enabled) to the Controller program over the UDP.

```
MainProgram:
  GoSub Initialization
  if AllOk
     while true
       GoSub Read_ThermoData
       xyText 200,130,Format(Tc,"##0.00°C = ")+Format(Tf,"0.00°F")+spaces(10),"",15,fs Bold
       n=usbm_ReadA(DeviceNum)
       xytext 205,155,bin(n&0x0F,4),"",20,fs_Bold
       x = ""+(255-Rate)
       if Direction == 0 && Rate < 255 then x = x+" Right"
       if Direction == 1 && Rate < 255 then x = x+" Left"
       SetEDit "Status",udp_Status("u1")
       xytext 205,190,JustifyL(x," ",12),"",15,fs_Bold
       if GetCheckBox("Allow Sending")
          US = 0 \ BuffPrintB US,Tc,n
          x = udp_Send("u1",US,GetEDit("rmtIP"),ToNumber(GetEdit("rmtPort"),0))
       endif
     wend
  endif
end
//================================================
Initialization:
  AllOk = false
  DeviceNum = 0    //use device 0
  Thermo_Pin = 15   //use pin B7 for input from the DS1820
  Direction = 0 \ Rate = 255 \ Steps = 255
  lclPort = 46000
  if usbm_DllSpecs() == "" then print "The USBmicro DLL is not installed" \return
  if !usbm_FindDevices() then print "There are no Devices" \ return
  S = toByte(1)+toByte(0)+toByte(Direction)+toByte(0)+toByte(3)+toByte(Rate)+toByte(0)
  if !usbm_DeviceValid(DeviceNum) then Return
  n = usbm_DirectionA(DeviceNum,0xF,0xF)  //SetPort A0-A3 as output
  n = usbm_Stepper(DeviceNum,S)           //Initialize motor
  S = BuffWriteB(S,1,1)                   //
  n = usbm_Stepper(DeviceNum,S)           //turn it on but no stepping
  AllOk = true
  x = udp_start("u1",lclPort)
  xyText 50,50,"Local IP    = "+TCP_LocalIP(),"Times new roman",14,fs_Bold
  xyText 50,70,"Local Port = "+lclPort,"Times new roman",14,fs_Bold
  xyText 500,50,"Remote IP    = ","Times new roman",14,fs_Bold
  xyText 500,80,"Remote Port = ","Times new roman",14,fs_Bold
  xyText 550,150,"Socket Status","Times new roman",14,fs_Bold
  AddEdit "rmtIP",630,50,100,0,tcp_LocalIP()
  AddEdit "rmtPort",630,80,60,0,47000 \ IntegerEdit "rmtPort"
  AddEdit "Status",500,180,250
  AddCheckBox "Allow Sending",530,110
  xyText 200,5,"Reader Program","Times new roman",20,fs_Bold|fs_Underlined
  xyText 10,130,"Temperature    =","",15,fs_Bold
  xyText 10,160,"Stepping State =","",15,fs_Bold
  xyText 10,190,"Stepping Rate  =","",15,fs_Bold
  onUDP udpHandler
Return
//================================================
```

```
Read_ThermoData:
  if !usbm_DeviceValid(DeviceNum) then Tc = -40 \ Tf = -40 \return
  n = usbm_reset1wire(DeviceNum,Thermo_Pin)
  n =usbm_write1wire(DeviceNum,0xCC) \ n =usbm_write1wire(DeviceNum,0x44)
  n =usbm_reset1wire(DeviceNum,Thermo_Pin)
  n =usbm_write1wire(DeviceNum,0xCC) \ n =usbm_write1wire(DeviceNum,0xBE)
  n =usbm_read1wire(DeviceNum) \ m = usbm_read1wire(DeviceNum)
  x = 0 \ if(m & 0x80) then x = 0xFFFFF000
  Tc = (x |(((m & 0x0F)<< 8)+n))*0.0625
  Tf = Tc*1.8+32
Return
//============================================================
Stepper:
  S = BuffWriteB(S,5,Rate) \ S = BuffWriteB(S,2,Direction)
  S = BuffWrite(S,6,Steps)
  n = usbm_Stepper(DeviceNum,S)  //do the steps
Return
//============================================================
udpHandler:
   if udp_BuffCount("u1") >= 4
      m = BuffReadI(udp_Read("u1"),0)
      Steps = 255 \ Direction = 0
      if m < 0 then Direction = 1
      if !m then Steps = 0
      Rate = 255-abs(m)
      GoSub Stepper
   endif
   onUDP udpHandler
Return
//============================================================
```

Program 11: UDP Reader.

4.2- The Controller Program

Figure 8 below is a screen shot of the Controller program. Notice the Remote IP and Port fields. Before you can allow sending data from the program to the Reader program ensure that these two fields are correctly filled. Also make sure that the Reader program is running. Then check the check box.

Warning!!! ***You must not allow sending from the program to a non-existing Port on the same IP as the Controller program. So before you allow sending you must ensure the other side is running. Likewise when you want to close the programs make sure that both have stopped sending by un-checking the check box.***

The remote IP and port fields of the Controller program should correspond to the Local IP and Port of the Reader program if you are running both programs within the LAN. If you are running them across the Internet then the remote IP should reflect the IP of the router of the LAN where the Reader is running. See Appendix B of the RobotBASIC_Networking.PDF document.

Notice that the Temperature is as received from the Reader program over the UDP link. Additionally, the Stepping State is also received from the Reader program over the UDP.

The Motor Speed is as acquired from a user through the Slider control. This value is displayed and also is sent over the UDP to the Reader program.

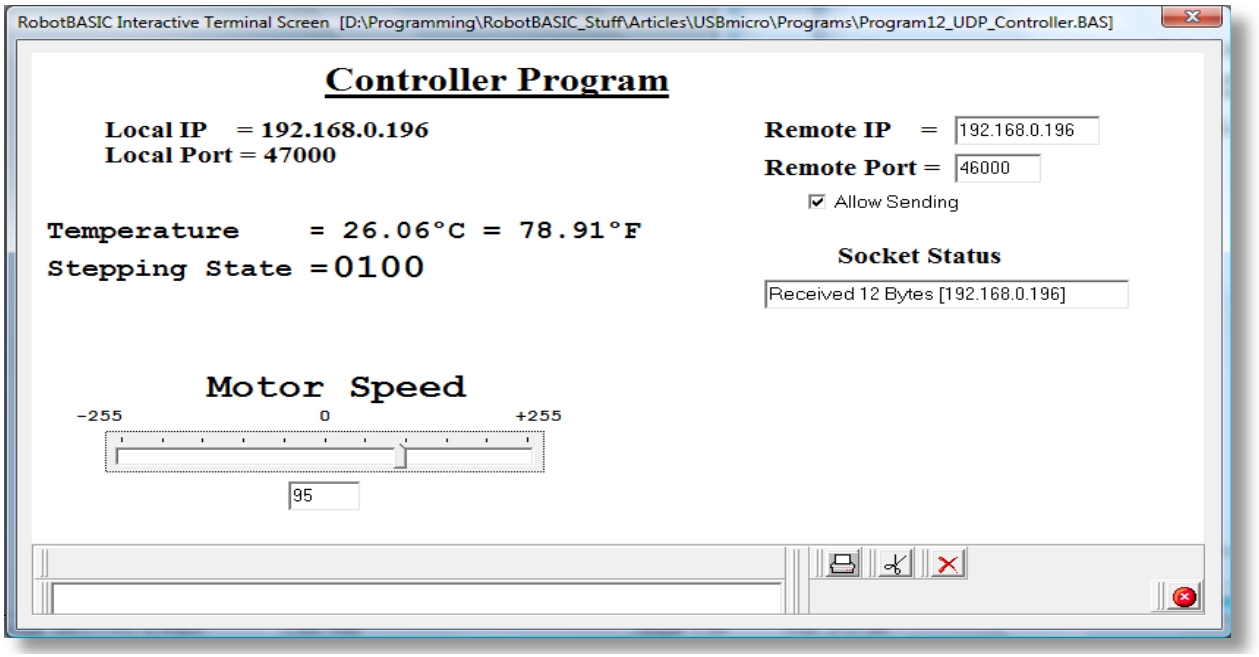


Figure 8: UDP Controller Program

```
MainProgram:
   GoSub Initialization
   while true
     SetEdit "Status",udp_STatus("u1")
   wend
end
//=============================================
Initialization:
  lclPort = 47000  \ Tc = -40
  x = udp_start("u1",lclPort)
  xyText 50,50,"Local IP    = "+TCP_LocalIP(),"Times new roman",14,fs_Bold
  xyText 50,70,"Local Port = "+lclPort,"Times new roman",14,fs_Bold
  xyText 500,50,"Remote IP    = ","Times new roman",14,fs_Bold
  xyText 500,80,"Remote Port = ","Times new roman",14,fs_Bold
  xyText 550,150,"Socket Status","Times new roman",14,fs_Bold
  AddEdit "rmtIP",630,50,100,0,tcp_LocalIP()
  AddEdit "rmtPort",630,80,60,0,46000 \ IntegerEdit "rmtPort"
  AddEdit "Status",500,180,250
  AddCheckBox "Allow Sending",530,110
  xyText 200,5,"Controller Program","Times new roman",20,fs_Bold|fs_Underlined
  xyText 10,130,"Temperature    =","",15,fs_Bold
  xyText 10,160,"Stepping State =","",15,fs_Bold
  AddSlider "Rate",50,300,300,-255,255 \ SetSliderPos "Rate",0
  AddEdit "Rate",175,340,50,0,0 \ ReadOnlyEdit "Rate"
  xyText 30,280,"-255","",10,fs Bold
  xyText 330,280,"+255","",10,fs_Bold
  xyText 196,280,"0","",10,fs_Bold
```

```
  xyText 120,250,"Motor Speed","",20,fs_Bold
  FocusSlider "Rate"
  onudp udpHandler \ onSlider sHandler
Return
//================================================
sHandler:
   ls = LastSlider()
   n = GetSliderPos(ls)
   SetEdit ls,n
   if GetCheckBox("Allow Sending")
      n = udp_Send("u1",BuffWrite("",0,n),GetEdit("rmtIP"),ToNumber(GetEdit("rmtPort"),0))
   endif
   onSlider sHandler
Return
//================================================
udpHandler:
   if udp_BuffCount("u1") >= 12
      SS = udp_Read("u1")
      Tc = BuffReadF(SS,0) \ Tf = 1.8*Tc+32
      Motor = BuffReadI(SS,8)
      xyText 200,130,Format(Tc,"##0.00°C = ")+Format(Tf,"0.00°F")+spaces(10),"",15,fs_Bold
      xytext 205,155,bin(Motor&0x0F,4),"",20,fs_Bold
   endif
   onUDP udpHandler
Return
//================================================
```

Program 12: UDP Controller

4.3- Observations

Instead of obtaining the data to set the motor speed and direction from a user it could be the result of a calculation depending on the received temperature value and a reference temperature. This would make for a ***Feed-Back-Control-System*** but with the ***Controller*** not being directly connected to the ***Process***. The link is achieved over the network. This means that this way you can have many remote PCs for collecting various data from widely dispersed processes where they all send ***sensory*** data to a central Controller that sends back ***actuations*** to each ***remote process*** depending on its data but also could be dependent on the overall data as collected from all sites.

This is a higher level of Feed-Back-Control, where the feedback is not necessarily from the process immediately under control, but, rather from other related and remote processes.

To implement such a setup you can use various PCs with RobotBASIC and multiple U4x1 devices. Each PC may also have multiple U4x1 devices doing multiple tasks in parallel.

The U4x1 combined with RobotBASIC opens up many possibilities for interesting projects that you would find hard to achieve with a different combination.