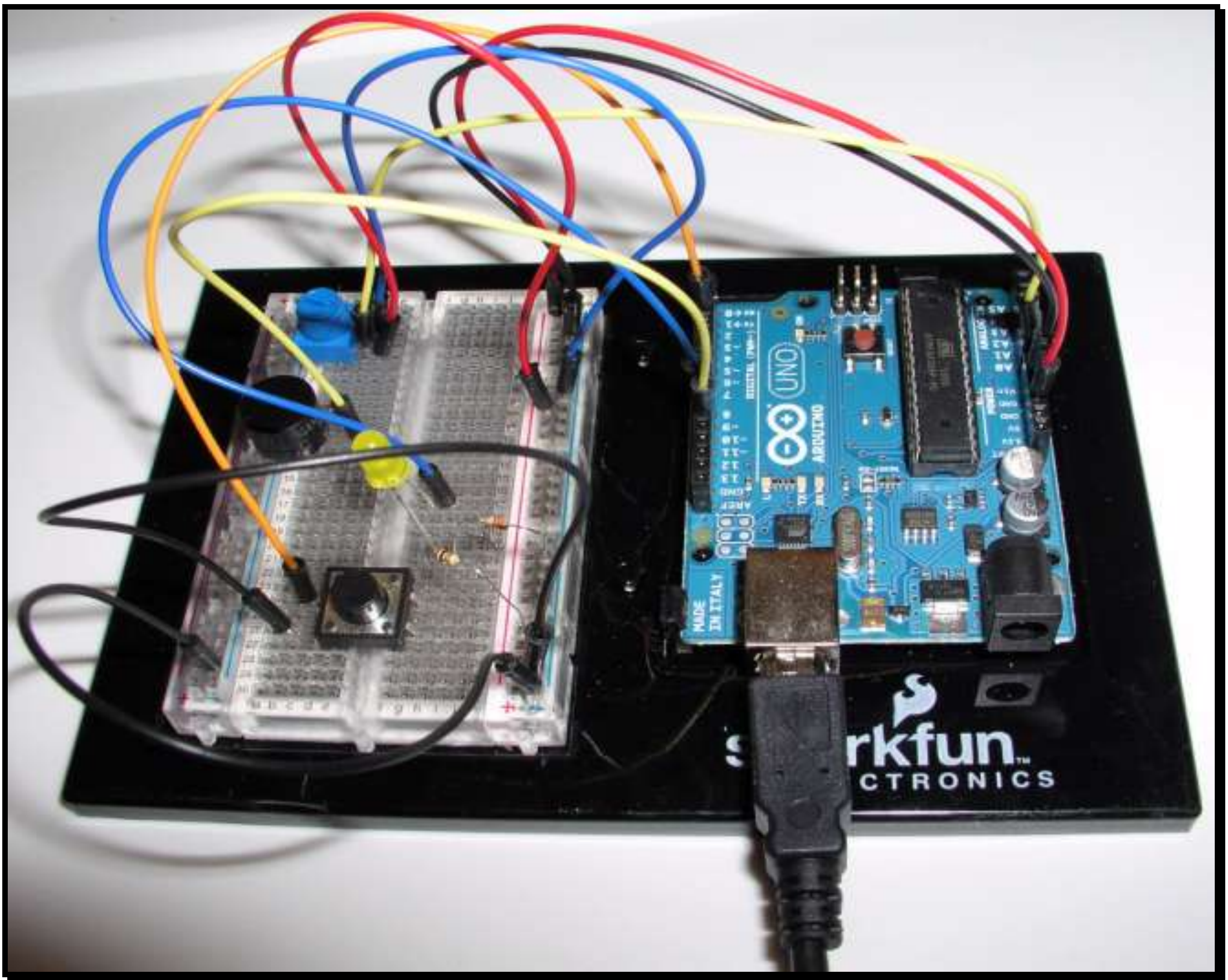


Interfacing The Arduino With A PC Using RobotBASIC's Protocol

Interfacing The Arduino With A PC Using RobotBASIC's Protocol

The Arduino is a convenient and nifty development board based on the capable Atmel AVR ATmega series of microcontrollers. We shall develop programs to achieve serial communications between an Arduino UNO board fitted with the ATmega328P processor and the PC. There are a variety of Arduino boards with various versions of the ATmega processor. The methods and programs expounded here will be applicable to all versions with no (or very minor) changes.

The Arduino also provides a powerful development environment for the Windows OS and a specialized adaptation of the C++ language to simplify program development for the board. The programming is for all intents and purposes C++ with some slightly specialized syntax for the Arduino IDE.

This paper will not attempt to explain the nuances of the Arduino system; you are assumed to be familiar with its IDE and its version of the C++ language. Also you are expected to be familiar with the RobotBASIC programming language.

The developed programs will be of sufficient complexity to illustrate communications most likely to be needed with projects that require interfacing a PC using RB and the Arduino's C++ languages. In addition we shall implement a version of the protocol proposed at length and in detail in the book *A Hardware Interfacing and Control Protocol* (paper and eBook). The protocol firmware will then be used to run (with small changes) two programs developed in the book but for a very different hardware and firmware. This goes to illustrate the ***power, flexibility and versatility*** of the protocol.

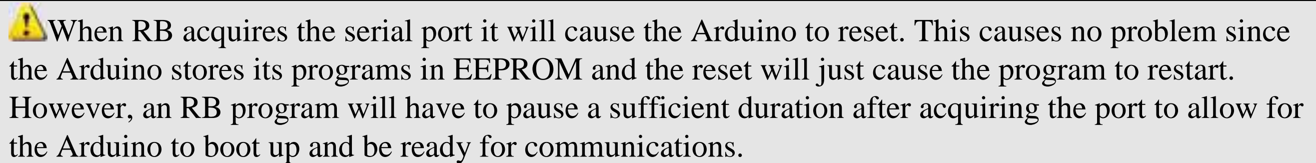
1- The Arduino Layout

Figure 1 shows the basic layout of the Arduino UNO board which can be powered from the PC's USB port or from an external power supply. We shall do all the work here using the USB power which should be sufficient for the current draw required in the developed projects. If your projects require a heavier current draw, using external power is seamless on the UNO board; all you have to do is connect an appropriate power supply.

In addition to providing the Arduino IDE with the ability to program the Arduino, the USB port also provides a means for effecting serial communications between an appropriate program running on the PC and the board. So the same USB port can be used to program the board and also to communicate with it.

No additional device will be needed to communicate the PC to the board. However, if wireless communications are required you will have to use the XBee Shield which is an additional board designed

to sit on top of the Arduino Uno board to enable wireless communications using the XBee over the same USB port by adjusting a jumper. We will not deal with this, however the shield provides a seamless connection to the XBee and ***the same programs will work wirelessly without any changes***.



When RB acquires the serial port it will cause the Arduino to reset. This causes no problem since the Arduino stores its programs in EEPROM and the reset will just cause the program to restart. However, an RB program will have to pause a sufficient duration after acquiring the port to allow for the Arduino to boot up and be ready for communications.

Figure 1: The Arduino Board.

2- The Arduino Software Serial UART

The Arduino provides a software UART that is interrupt driven and has a circular buffer of 128 bytes. This facilitates a capable and easy-to-program serial communications that allow the Arduino to do sending and receiving while also doing other tasks.

The UART can support baud rates up to 115200 bps which is specified upon activating the system with the statement **Serial.begin(baudrate)**. Once initiated it will receive bytes in the background and put them in a buffer. We can find out how many bytes there are in the buffer using the **Serial.available()** statement. We can also look at the value of the ***first*** byte in the buffer using **Serial.peek()** which returns the value of the first byte in the buffer ***but without extracting it*** from the buffer. To extract the first byte in the buffer **Serial.read()** is used; the next call will return the next byte and so on. **Serial.flush()** will remove all bytes from the buffer to clear it in case there are any bytes left from an old communication.

To send binary bytes out on the port **Serial.write(parameter{,length})** is used. ***Parameter*** can be a byte value or an array of bytes. If it is a byte or a string (0 terminated) then the ***length*** is not needed. If it is an array (buffer) of bytes then the number of bytes has to be specified using ***length***.

To send numbers as text instead of binary values use **Serial.print(value{,format})** or **Serial.println (value {,format})**. They will convert a number to the specified format (Decimal if not specified) and will send the bytes that represent the number in that format. If the value is a floating point then the format is

the number of decimal places (default two). The **println()** version will also send a carriage return (13) and line feed (10) characters at the end of the string sent out.

See Appendices A and B regarding buffer creation. They are written from the perspective of the RobotBASIC language however the principles and methodologies apply to the Arduino language too.

The Arduino circular buffer holds 128 characters before it starts overwriting the bytes from the beginning. Thus, in order to not lose data you need to assure that the bytes are extracted (**read()**) from the buffer before it overflows.

3- The RobotBASIC Software Serial UART

RobotBASIC provides a buffered UART with a circular receive buffer of 4096 characters and sending and receiving in the background. RB can do three types of serial communications:

- By periodically checking the number of bytes in the buffer while it is doing other tasks.
- By waiting with a time out period for a certain number of bytes to arrive.
- By using an event handler (interrupts), which is a little complex. RB can trigger an event handler whenever bytes arrive in the receive buffer. The handler can then do any action necessary like reading the bytes into another buffer and then doing something only when the required number or set of bytes have been accumulated. We will not consider this third method in this paper since it requires more advanced programming principles.

3.1 RobotBASIC's Serial Communications Commands

To accomplish serial communications with RB the program has to be able to:

- Indicate the serial port to be used and the Com parameters like the baud rate and so forth (**SetCommPort** command).
- Clear the receive buffer (**ClearSerBuffer** command).
- Check the amount of data in the receive buffer (**CheckSerBuffer** command).
- Get bytes from the receive buffer (**SerIn** and **SerBytesIn** commands).
- Put bytes into the send buffer (**SerOut** and **SerialOut** commands).
- Change the timeout period (**SetTimeOut** and **GetTimeOut** commands).

The above 9 commands are all that you need to send and receive serial data. However, there are also a few functions that are needed to manipulate the data to be sent and received as discussed in Appendix B.

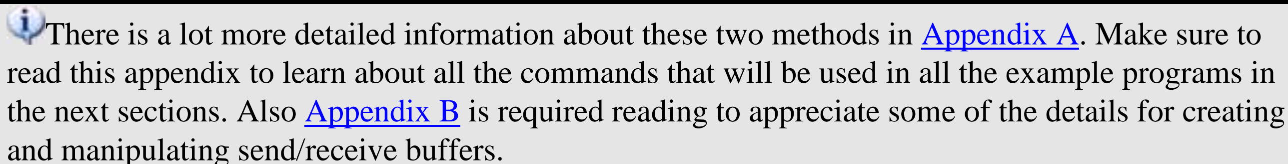
3.2 Sending and Receiving Data With RobotBASIC

When sending and receiving data there are two modes for doing so:

- Textual mode
- Binary mode

In the textual mode, data is received/sent as a series of bytes. The bytes are characters representing the text being received/sent. If there are numbers to be sent then they are converted to their decimal format whether integers or floating points. This is achieved with the **SerOut** command. When the data being received has numbers in it then these numbers are the decimal text forms of the numbers. They can be extracted and converted back to proper integers or floats using the **ToNumber()** function.

In the binary format the data is received/sent as a series of bytes. The bytes are binary byte values either representing characters if the data is text or byte sized numbers (8-bit integers) if the data is numerical data. But also the bytes may be a set of 4 (or 2) bytes to represent a long (or word) integer (32 or 16-bit integers) or a series of 8 (or 4) bytes to represent double (or single) precision floating point numbers (64 or 32-bit). This is achieved using **SerialOut.**

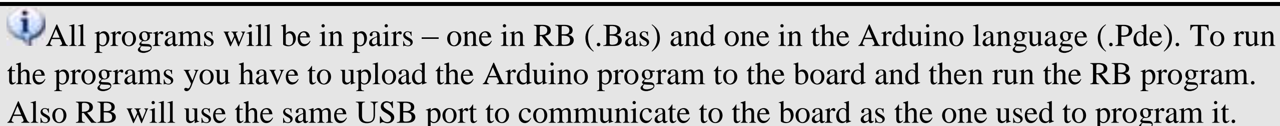

There is a lot more detailed information about these two methods in Appendix A. Make sure to read this appendix to learn about all the commands that will be used in all the example programs in the next sections. Also Appendix B is required reading to appreciate some of the details for creating and manipulating send/receive buffers.

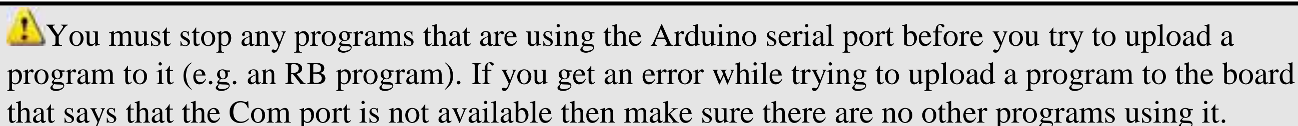
4- Example Projects

The RobotBASIC programs developed in this section use some advanced buffer manipulation techniques. These are described in detail in Appendices A and B. You may need to refer to these appendices and you also should refer to the RobotBASIC Help File.

Initially we shall develop a simple program to demonstrate basic communications between the Arduino and RobotBASIC using a pushbutton and an LED on the Arduino and a simulated pushbutton and LED on the RB side. We will then develop a slightly more complex program using analog input and output using a potentiometer and PWM on the Arduino and two sliders on the RB side. Afterwards we will develop a slightly more complex program to demonstrate sending and receiving data of all possible variable types with some processing of the data in the Arduino.

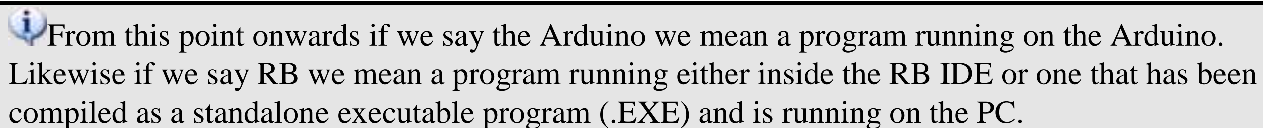
In Section 5 we shall develop a sophisticate ***protocol*** to allow for ***robust*** and ***versatile*** communications between the Arduino and the PC and to allow for many devices on the Arduino all working together.

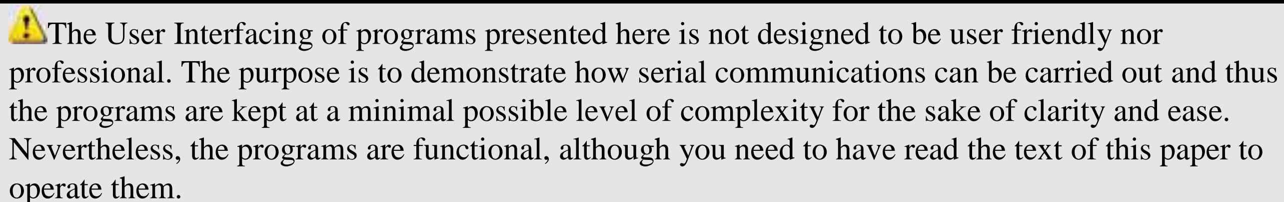

All programs will be in pairs – one in RB (.Bas) and one in the Arduino language (.Pde). To run the programs you have to upload the Arduino program to the board and then run the RB program. Also RB will use the same USB port to communicate to the board as the one used to program it.


You must stop any programs that are using the Arduino serial port before you try to upload a program to it (e.g. an RB program). If you get an error while trying to upload a program to the board that says that the Com port is not available then make sure there are no other programs using it.

When an Arduino is installed on the PC it establishes a virtual serial port. This port number will be used in RobotBASIC to set the communications parameter. You need to know this port number, which was told to you when you first installed the Arduino on your PC. If you are not familiar with the number you can run the one line RB program given below to see what serial ports are available and then experiment with them to see which one is the one connected to the Arduino. Alternatively you can use the Windows OS facilities to view the port assigned to the Arduino.

```
SerPorts n \ print n
```


From this point onwards if we say the Arduino we mean a program running on the Arduino. Likewise if we say RB we mean a program running either inside the RB IDE or one that has been compiled as a standalone executable program (.EXE) and is running on the PC.


The User Interfacing of programs presented here is not designed to be user friendly nor professional. The purpose is to demonstrate how serial communications can be carried out and thus the programs are kept at a minimal possible level of complexity for the sake of clarity and ease. Nevertheless, the programs are functional, although you need to have read the text of this paper to operate them.

4.1 A Basic Program

In this program we will have a pushbutton and an LED on the Arduino. Also on the RB side there will be a pushbutton and a ***simulated*** LED.

The Arduino will continuously examine the serial buffer and if there is a byte it will read it and set the LED to on or off accordingly. Also at the same time it will monitor the pushbutton and send its status ***if changed*** to the serial port. The RB program will do the exact same thing as the above except the button is used as a toggle switch. This means that the Arduino should always light the LED according to the button toggle on RB and RB will light its simulated LED according to the pushbutton on the Arduino.

The LED and pushbutton on the Arduino should be wired according to Figure 2.

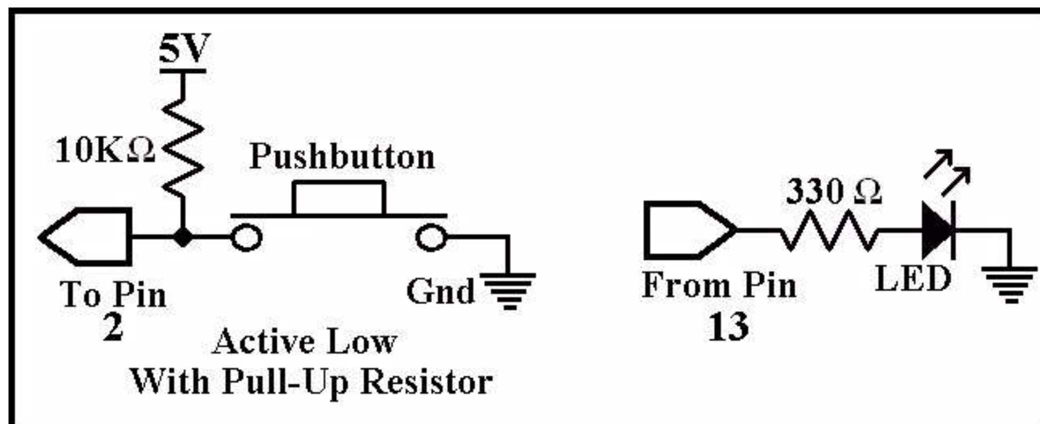


Figure 2: Schematic of LED and Pushbutton wiring on the Arduino.

Program_01.Pde

```
//Arduino to RB Communications Project
//Program_01.Pde
#define LED_Pin 13     //led pin
#define Pb_Pin 2       //pushbutton pin
int new_Pb, old_Pb = HIGH;
//-----------------------------------
void setup()
{
  Serial.begin(115200);     //init serial comms
  pinMode(LED_Pin, OUTPUT);
  pinMode(Pb_Pin,  INPUT);
}
//-----------------------------------
```

```
void loop()
{
  if(Serial.available() > 0)    //if byte available
      digitalWrite(LED_Pin,Serial.read()); //Set LED accordingly

  new_Pb = digitalRead(Pb_Pin); //read button
  if (new_Pb != old_Pb)  //if state changed
  {
      old_Pb = new_Pb;
      Serial.write(!new_Pb);    //send the status only if changed
                                //but inverted to make it active high
      delay(100);               //delay to allow for bounce
  }
}
```

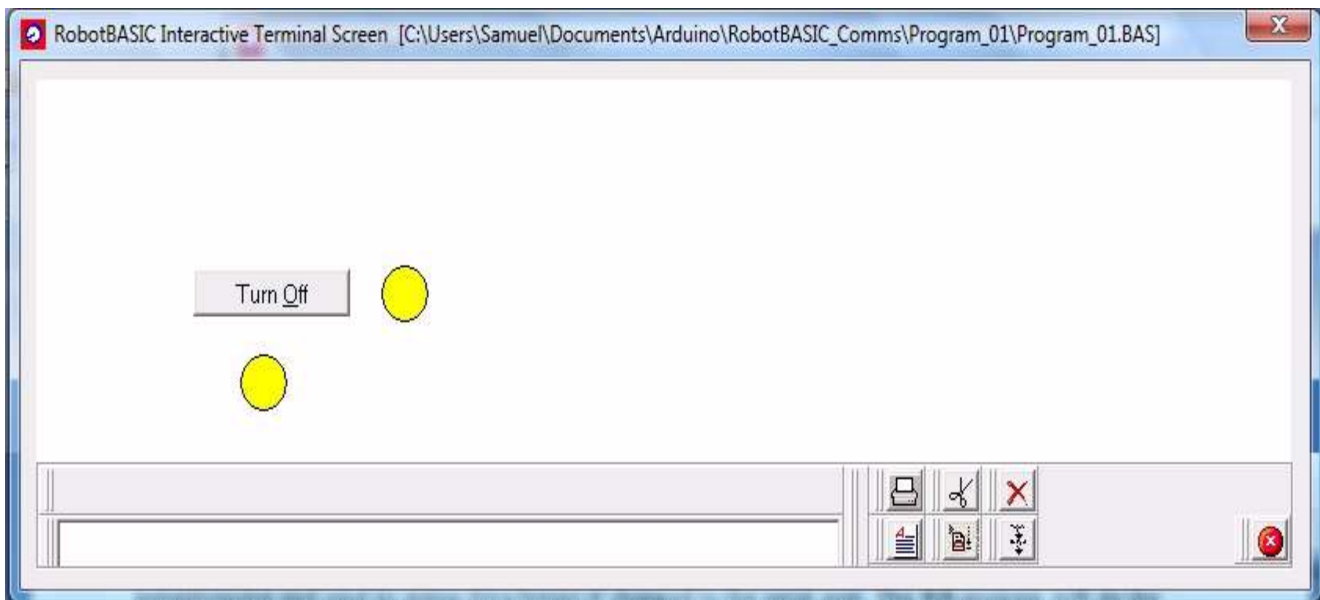


Figure 3: Screenshot of Program_01.Bas in action

Program_01.Bas

```
//Arduino to RB Communications Project
//Program_01.Bas
Port = 3   //change this as per your system
Main:
   setcommport Port,br115200    //init the serial port
   data bname; "&Turn On", "Turn &Off"  //button captions
   data LEDColor;white,yellow           //LED unlit/lit colors
   addbutton bname[0],100,100,100       //create the button
   old_Pb = 1 \ new_Pb = 0
   circleWH 130,145,30,30,0,LEDColor[0] //simulated LED
   while true
      btn = lastbutton()                //get button status
      if btn == bname[0]                //if turn on
         renamebutton btn,bname[1]      //rename button
         new_Pb = 1
      elseif btn == bname[1]            //if turn off
         renamebutton btn,bname[0]      //rename button
         new_Pb = 0
      endif
```

```
      if new_Pb != old_Pb               //only send status if changed
         SerialOut new_Pb \ old_Pb = new_Pb       //send it
         circleWH 220,98,30,30,0,LEDColor[new_Pb] //display LED as it should
      endif                                       //be on the Arduino side
      CheckSerBuffer n                   //check serial receive buffer
      if n                               //if there is data from the Arduino?
         Serin x                         //read the buffer
         x = (GetStrByte(x,1) > 0)       //extract the byte
         circleWH 130,145,30,30,0,LEDColor[x] //simulate an LED according to
      endif                                   //button on the Arduino side
   wend
End
```

4.2 A More Complex Program

In this program we have a potentiometer and an LED on the Arduino. On the RB side there are two sliders. You can manipulate one slider (bottom in Figure 5) to vary the brightness level of the LED on the Arduino board. When you change the slider its position value is sent to the Arduino which sets the PWM on the LED pin to control its brightness in proportion to the slider's position. The level is sent and received as a byte value.

The Arduino continuously reads the analog voltage level on the potentiometer and if there is a change it sends two bytes representing (Most Significant Byte First) the value read on the Arduino's analog input pin. RB receives the two bytes and reconstitutes the 16-bit number and sets the position of the second slider (top one) to correspond to the position of the potentiometer on the Arduino. The slider cannot be controlled on the RB side it only reflects the Arduino potentiometer level. ***The Arduino's ADC is a 10-bit ADC but it is sent as an integer (16-bits)***. The LED and potentiometer on the Arduino will be wired according to Figure 4.

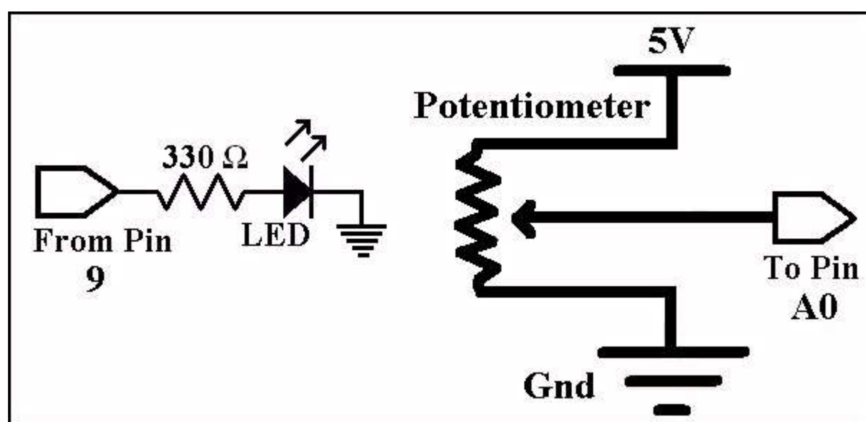


Figure 4: Schematic of LED and Pushbutton wiring on the Arduino.

Program_02.Pde

```
//Arduino to RB Communications Project
//Program_02.Pde
#define LED_Pin 9    //led pin can do PWM
#define Pot_Pin 0    //Pot on Analog A0 pin
int new_Pot, old_Pot = -1;
//-----------------------------------
void setup()
{
  Serial.begin(115200);     //init serial comms
}
```

```
//---------------------------------
void loop()
{
  if(Serial.available() > 0)   //if byte available
      analogWrite(LED_Pin,Serial.read()); //Set LED brightness accordingly
  new_Pot = analogRead(Pot_Pin); //read pot level
  if (new_Pot != old_Pot)  //if level change
  {
      old_Pot = new_Pot;
      Serial.write(new_Pot >> 8);  //send MSByte
      Serial.write(new_Pot & 0xFF); //send LSByte
      delay(100);  //delay to allow for bounce
  }
}
```

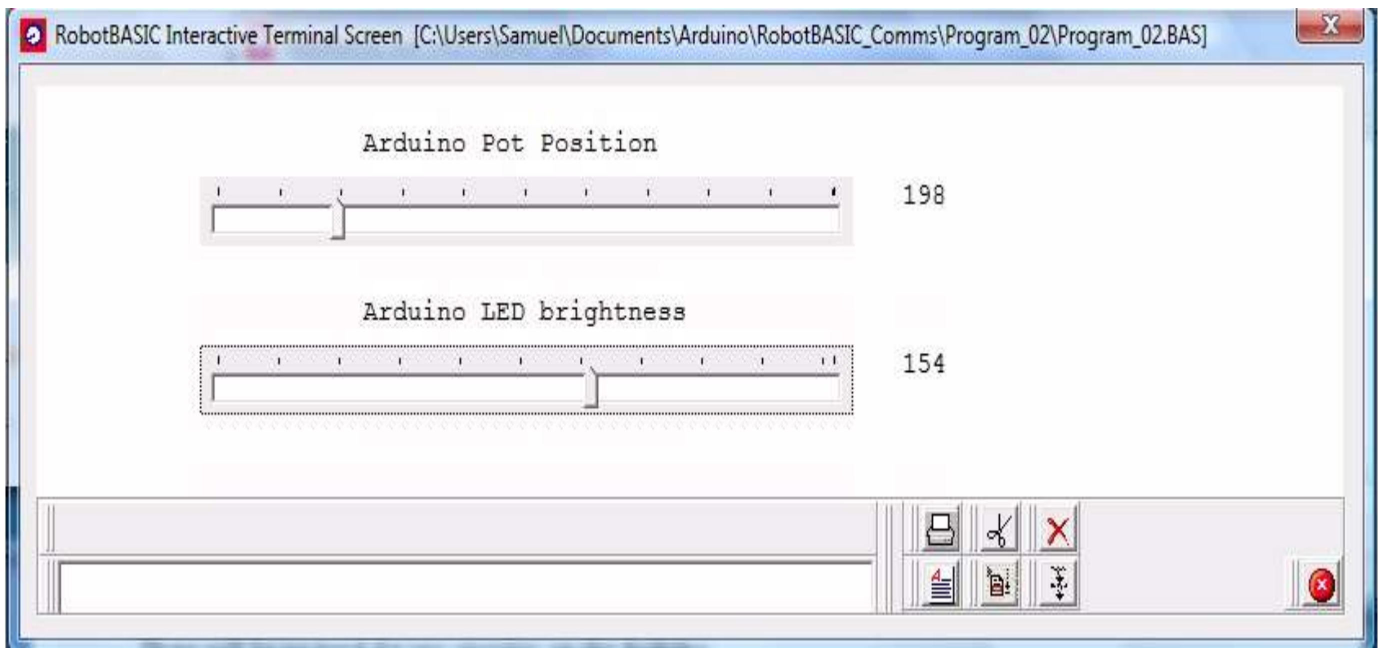


Figure 5: Screenshot of Program_02.Bas in action.

Program_02.Bas

```
//Arduino to RB Communications Project
//Program_02.Bas
Port = 3   //change this as per your system
Main:
   GoSub Initialization
   while true
      new_LED = GetSliderPos("LED")
      if new_LED != old_LED
         xystring 530,180,new_LED,spaces(30)
         old_LED = new_LED
         serialout new_LED
      endif
      call GetPot(new_Pot)
      if new_Pot != old_Pot
         xystring 530,100,new_Pot,spaces(30)
         SetSliderPos "Pot",new_Pot
         old_Pot = new_Pot
```

```
      endif
   wend
End
//--------------------------------------------
Initialization:
   old_Pot = 2000 \ old_LED = 2000
   new_Pot = 0 \ new_LED = 0
   setcommport Port,br115200    //init the serial port
   addslider "Pot",100,100,400,0,1024
   EnableSlider "Pot",false
   addslider "LED",100,180,400,0,255
   xystring 200,75,"Arduino Pot Position"
   xystring 200,155,"Arduino LED brightness"
Return
//--------------------------------------------
sub GetPot(&a)
   CheckSerBuffer n
   if n >1
      Serin s
      a = (GetStrByte(s,1) << 8)+GetStrByte(s,2)
   endif
Return
```

4.3 Sending and Receiving All Types of Data

In this program RB will present a set of edit boxes where you can enter numbers and text. The numbers will be a byte, a word (16-bits) integer, a long (32-bit) integer and a single precision float. Once a pushbutton is pressed the data will be sent to the Arduino which will capitalize the text and add one to all the numbers and then will send them back to RB which will then display these returned values.

This demonstrates sending and receiving all possible variable types. The 16-bit and 32 bit integers are sent in the ***Little-Endian*** format. That means the Least Significant Byte is sent first and so forth. RB and the Arduino share the Little-Endian order of storing integers in memory which simplifies the sending and reconstitution procedures.

In addition, the Arduino will also blink an LED on Pin 13 connected as shown in Figure 2. This demonstrates how multitasking can be performed. The LED is blinked at a pre-specified rate. However we shall have it so that the 16-bit number in the sent numbers is also used to specify the blink rate just to see how we can use the value as an example of other than just incrementing.

The RB program is slightly complex, but is easily understood if you study Appendices A and B. It is very similar to the example explained in the last section of Appendix B.3. Also many of the program's buffer manipulation commands and functions are explained in those two appendices.

In the Arduino program the only section that might present any difficulty is the **manipulateBuffer()** function. This function uses the buffer that was received from RB with all the data fields in it. The text fields are allowed to be of variable length but up to a maximum of 255 characters. So every text field will have a byte value just before the start of the text. This byte specifies the length of the text. This way we can extract the text and also know where the next field after it starts. Numerical fields are of course fixed in length depending on the type.

If you study the RB program you will see how the buffer was created. All numbers are in little-endian format (LSBF).

The **manipulateBuffer()** function starts by calculating the indexes into the buffer for the starts of all the fields. For the int, long and float fields we use a powerful pointer construct to point into the buffer but to also use the bytes sequence as a long or int or float. This allows us to do math on the field but without having to extract it from the buffer first.

So the construct

```
      (*(long *)(buff+ZC))++;    //increment the Zip code (long 32-bit)
```

Uses the buffer's start address and adds to it the index within the buffer for the Zip Code field. This is a character pointer to the first byte where the field starts. But then we ***recast*** it to be a pointer to a long which enables us to use the 4 bytes starting at the index byte as a long integer (32-bits). This then can be incremented or used in any operation to read it or save data into it just like any normal long variable.

The result is that we do not have to extract the bytes, reconstitute the number, increment it then deconstruct it and put it back in the buffer. Instead we just use the number as it is inside the buffer as if it were a number stored as a variable on its own. This also is done for the float and int values.

Notice how in the **loop()** function we keep updating the blinker LED when needed according to the blink rate despite also checking for data in the serial buffer and copying the bytes as they arrive into the bigger buffer. Copying the bytes as they arrive into a bigger buffer is needed because the data may be up to 521 bytes while the Arduino buffer can only hold 128 bytes. So we need to remove the bytes to a bigger buffer to prevent buffer overflow and loss of data. When all the bytes arrive we then manipulate the buffer as discussed above to increment the numbers and to convert the text fields to upper case.

How do we know when all the bytes have arrived? The first 4 bytes received are a Little-Endian format for a long integer value that specifies the number of bytes to be expected (including the first 4 bytes).

This project is slightly complex but should be easily understood if you have appreciated the principles discussed in the appendices and in the previous sections. It demonstrates how data can be exchanged between RB and the Arduino no matter what type and format.

Program_03.Pde

```
//Arduino to RB Communications Project
//Program_03.Pde
#define LED_Pin 13   //LED digital pin
int Interval =500; //blink rate
char Mode = 0;       //LED status
long t;              //stop watch timer
char buff[600];      //buffer to hold data
int received = 0;
long count=0;
//-------------------------------
void setup()
{
  Serial.begin(115200);     //init serial comms
  pinMode(LED_Pin,OUTPUT);
  t = millis();
}
```

```
//--------------------------------
void manipulateBuffer()
{
   //add 1 to all numerical fields
   buff[4]++ ; //Code
   int N,BY,Add,ZC,Bln;  //field position index
   //calculate indexes into the buffer for all fields
   N = 6;
   BY = buff[5]+6;
   Add = BY+3;
   ZC = Add+buff[Add-1];
   Bln = ZC+4;

   //increment Birthyear (int 16 bit)also use it to set the Blinker rate
   Interval = (*(int *)(buff+BY))++;
   (*(long *)(buff+ZC))++;   //increment the Zip code (long 32-bit)
   (*(float *)(buff+Bln))++; //increment the Balance (float 32-bit)


   //capitalize the Name and Address
   for(int i=0; i< buff[5];i++)
      Capitalize(buff+i+N);
   for(int i=0; i< buff[Add-1];i++)
      Capitalize(buff+i+Add);
}
//--------------------------------
void Capitalize(char *c)
{
   if (*c >= 'a' && *c <= 'z')
      *c += 'A'-'a';
}
//--------------------------------
void loop()
{
  if(millis()-t > Interval)  //if timed out toggle the LED
  {
     Mode = !Mode; t = millis(); digitalWrite(LED_Pin,Mode);
  }
  if(Serial.available() > 0)   //any bytes
      buff[received++] = Serial.read(); //copy them to the secondary buffer
  if((received >= 4) && (count == 0))
     count = *(long *)buff;
  if((count > 0) && (received >= count))
  {
    manipulateBuffer();
    Serial.write((unsigned char *)buff,count);
    Serial.flush();received = count = 0; //restart the waiting
  }
}
```

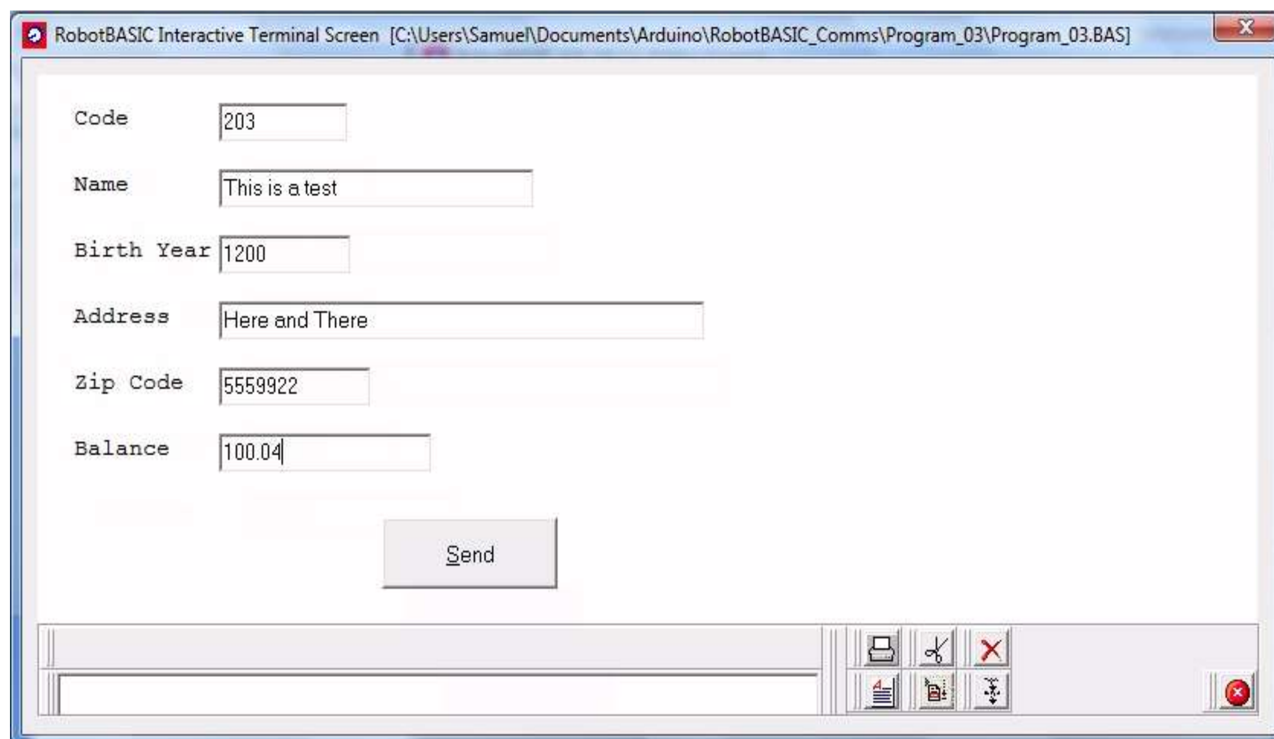


Figure 6: Program_03.Bas in action.

Program_03.Bas

```
//Arduino to RB Communications Project
//Program_03.Bas
Port = 3   //change this as per your system
Main:
   GoSub Initialization
   while true
      if LastButton() != ""  //if button pushed
          xyText 500,200,spaces(50),,20,fs_Bold
          call CreateTheBuffer(s,x)
          Serialout s \  SerBytesIn x,r,n
          if n == x //buffer received back correctly
             call ExtractTheBuffer(r)
          else //communications process failed
              xyText 500,200,"Comms Error",,20,fs_Bold,red
          endif
      endif
   wend
End
//------------------------------------------
Initialization:
   setcommport Port,br115200
   xyText      40,55,"Code"
   xyText      40,95,"Name"
   xyText      40,135,"Birth Year"
   xyText      40,175,"Address"
   xyText      40,215,"Zip Code"
   xyText      40,255,"Balance"
   AddEdit     "Code",135,55,85
   AddEdit     "Name",135,95,207
```

```
   AddEdit     "Birth Year",135,135,87
   AddEdit     "Address",135,175,319
   AddEdit     "Zip Code",135,215
   AddEdit     "Balance",135,255,140
   AddButton   "&Send",263,355,115,43
   IntegerEdit "Code"     \ IntegerEdit "Birth Year"
   IntegerEdit "Zip Code" \ FloatEdit "Balance"
   FocusEdit "Code"
Return
//------------------------------------------
sub CreateTheBuffer(&s,&x)
    //creat the buffer to be sent with all the data from the edit boxes
    s  = toByte(ToNumber(GetEdit("Code"),0))
    a  = GetEdit("Name") \ L = Limit(Length(a),0,255)
    s += tobyte(L)+Left(a,L)
    n  = ToNumber(GetEdit("Birth Year"),0)
    s += toByte(n)+toByte(n >> 8) //LSBFirst
    a  = GetEdit("Address") \ L = Limit(Length(a),0,255)
    s += tobyte(L)+Left(a,L)
    s += BuffWrite("",0,ToNumber(GetEdit("Zip Code"),0))
    s += BuffWriteF32("",0,ToNumber(GetEdit("Balance"),0)*1.0)
    x  = Length(s)+4
    s  = BuffWrite("",0,x)+s  //prefix the buffer with its length
Return
//------------------------------------------
sub ExtractTheBuffer(s)
   //extract the fields from the received buffer and set the edit boxes
   //accordingly
   SetEdit "Code",GetStrByte(s,5)
   n = GetStrByte(s,6)
   SetEdit "Name",BuffRead(s,6,n) \ n += 6
   SetEdit "Birth Year",(GetStrByte(s,n+2)<<8)+GetStrByte(s,n+1) //LSBFirst
   n+=2
   c = GetStrByte(s,n+1) \ n++
   SetEdit "Address", BuffRead(s,n,c) \ n+=c
   SetEdit "Zip Code",BuffReadI(s,n) \ n+=BytesCount_I
   SetEdit "Balance",BuffreadF32(s,n)
   FocusEdit "Code"
Return
```

5- Implementing a Protocol

All the programs we developed so far have been very useful for demonstrating the principles of effecting the communication between RB and the Arduino. However, the programs were not very ***robust*** or ***versatile***.

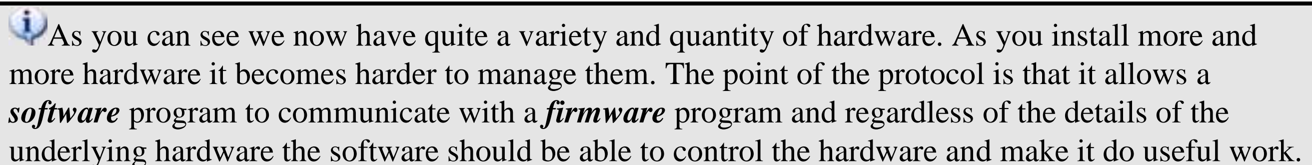
In this section we will develop a standard ***firmware*** to put on the Arduino that will allow us to combine many sensors and actuators all working together and controlled by an RB program.

To do this we need a ***protocol*** that can be easily extended and at the same time allows for clean recovery from any drop of the communications. For example if the power is interrupted on the Arduino side the protocol should be such that when the power is restored RB and the Arduino are be able to recommence communications without the need to reset the RB side (and vice versa).

Such a powerful and versatile protocol is described in extensive details in *A Hardware Interfacing and Control Protocol* (paper and eBook). The book describes all the principles and shows numerous projects that utilize it.

As we shall demonstrate below, the ***very same*** RB programs developed in the book can run on the Arduino. These programs were developed to run with the protocol as implemented on the Propeller Chip. However, the whole idea of the book and the aim of the protocol is that programs on the PC should be able to work with the hardware regardless of what it is as long as it obeys the protocol. The programs we develop in the next few sections actually were developed for the book. They have been ported over to work in this paper with hardly any modification.

In order to achieve the above, we of course need to develop the protocol on the Arduino. While doing so we shall combine the hardware we used as shown in Figures 2 and 4. Additionally we shall add a Piezo speaker connected as shown in Figure 7.



As you can see we now have quite a variety and quantity of hardware. As you install more and more hardware it becomes harder to manage them. The point of the protocol is that it allows a ***software*** program to communicate with a ***firmware*** program and regardless of the details of the underlying hardware the software should be able to control the hardware and make it do useful work.

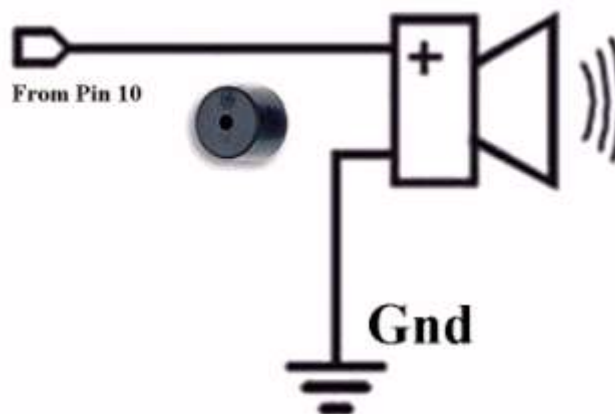


Figure 7: Schematic for a Piezo speaker to be added to our hardware.

5.1 The Firmware

The program Arduino_Firmware.Pde, shown below, implements a protocol that allows an RB program to control the following hardware:

- A Piezo Speaker connected as in Figure 7 to Pin 10 to be driven using the Arduino frequency generation facility (**tone()**).
- An LED connected as in Figure 2 to Pin 13 that will be blinked at a rate that can be changed through an RB program via the protocol.
- An LED connected as in Figure 4 to Pin 9 to be driven using PWM to vary its brightness level as commanded by an RB program.
- A potentiometer as shown in Figure 4 connected to the analog input Pin A0 so as to be able to read it and return its voltage level to an RB program when requested.
- A Pushbutton connected as in Figure 2 to the digital input Pin 2 so as to be able to send its status to an RB program when required.

The protocol is described in *A Hardware Interfacing and Control Protocol* (paper and eBook). Here we shall use a slightly modified version to allow for the different characteristics of the Arduino:

1. RB sends two bytes to the Arduino. The first byte is a command code and the second is a parameter value.
2. The Arduino will wait for the first byte forever, but will do other tasks while it is waiting. When the byte arrives, it will be stored in a buffer and then the Arduino will wait for the second byte but with a timeout (still doing other tasks). If the second byte comes in within the timeout the Arduino will then proceed as below. If no second byte arrives the system will go back to waiting for the first byte after discarding the previously arrived byte.
3. The system receives these two bytes and decides what RB wants done and how depending on the command and its parameter.
4. Within a timeout period the Arduino should carry out the command then fill the first three bytes of a 5-byte buffer with data that will always be returned in those three bytes (but see step 6). In our system we shall use the first byte for the pushbutton status. The other two will not be used but are filled with 0. They could be bumpers or infrared sensors and so forth.
5. If the executed command also returns a value then the 4th and 5th bytes are used to return the value (but see step 6). We shall do this for the value of the Potentiometer.
6. If the command requires returning values of more than 2 bytes then it uses all the 5 bytes from the top and forgoes sending the three bytes in step 4. We won’t need this in this paper.
7. If RB does not receive the 5 bytes within a timeout it will assume a communications failure and will take action accordingly.
8. When the 5 bytes arrive RB will use the first three to update the values in its own memory and discard the extra bytes unless the command is one that returns a value as in steps 5 or 6.
9. If the command also sends data back in the last two bytes RB will extract that from the last two bytes and create the number (Using Most Significant Byte First standard MSBF) and use it accordingly.

You shall see how this is implemented in the Arduino_Firmware.Pde program. The RB side of the action will be illustrated in later sections.

If at some stage you add new hardware you can extend the firmware very easily to incorporate the new hardware. You also have to assign it a new command code and to specify its required parameters. The table below shows the commands as implemented in our system so far. As you add new commands you need to update the table.

Table 1: List of protocol command codes at this stage.

Command	Code	Parameter	Data Returned
Read the Pot	2	0	4th and 5th bytes
Set LED brightness	3	Level (0-255)	None
Set LED Blink duration	4	10ms intervals (0-255)	None
Set Speaker Sound	73	Note Number (0-84)	None

The speaker will play the note for 5 seconds unless a new frequency or a 0 is commanded.

Arduino_Firmware.Pde

```
//Arduino to RB Communications Protocol Implementation
//Arduino_Firmware.Pde
#define LED_Pin    13    //led pin  digital out
#define Pb_Pin      2    //pushbutton pin digital in
#define Pot_Pin     0    //A0 analog input
#define Dimmer_Pin  9    //PWM analog out
#define Speaker_Pin 10   //speaker with tone()

long blinker_timer,serial_timer;
long serial_timeout = 5000;
int received_bytecount=0;
char command[2];
char sendbuffer[5];
long blinkrate = 500;
int dimmer_level = 127;

int Notes [] =  //array of frequencies for sound generation
{
0,32.703,34.648,36.708,38.891,41.203,43.654,
  46.249,48.999,51.913,55.0,58.27,61.735,

65.406,69.296,73.416,77.782,82.407,87.307,
92.499,97.999,103.83,110.0,116.54,123.47,

130.81,138.59,146.83,155.56,164.1,174.61,
185.0,196.0,207.65,220.0,233.08,246.94,

261.63,277.18,293.66,311.13,329.63,349.23,    //middle C
369.99,391.99,415.31,440.0,466.16,493.88,

523.25,554.37,587.33,622.25,659.26,698.46,
739.99,783.99,830.61,880.0,932.33,987.77,

1046.5,1108.7,1174.7,1244.5,1318.5,1396.9,
1480.0,1568.0,1661.2,1760.0,1864.7,1975.5,

2093.0,2217.5,2349.3,2489.0,2637.0,2793.8,
2960.0,3136.0,3322.4,3520.0,3729.3,3951.1
};
//-----------------------------------
```

```
void setup()
{
  Serial.begin(115200);     //init serial comms
  Serial.flush();
  pinMode(LED_Pin, OUTPUT);
  pinMode(Pb_Pin,  INPUT);
  blinker_timer = millis();
}
//-----------------------------------
void Blink_LED()
{
 static int Mode = 0;
    if(millis()- blinker_timer > blinkrate)
    {
        Mode = !Mode;
        digitalWrite(LED_Pin,Mode);
        blinker_timer = millis();
    }
}
//-----------------------------------
void Read_ThreeBytes()
{
  sendbuffer[0] = !digitalRead(Pb_Pin);//push button status
                                       //inverted for active high
  sendbuffer[1] = 0;  //nothing for now
  sendbuffer[2] = 0;  //nothing for now
}
//-----------------------------------
void SetSpeaker(int which_note)
{
  which_note = constrain(which_note,0,84); //array limits
  if(which_note > 0)
     tone(Speaker_Pin,Notes[which_note],5000);
  else
    noTone(Speaker_Pin);
}
//-----------------------------------
void ReadPot()
{
   //set the last two bytes of the send buffer to the
   //value of the pot using MSBFirst
   int x = analogRead(Pot_Pin);
   sendbuffer[3] = char(x >> 8);
   sendbuffer[4] = char(x);
}
//-----------------------------------
void loop()
{
  Blink_LED();          //blink on P13
  Read_ThreeBytes();    //read top three bytes of send buffer
  //SetSpeaker(10);
  if(Serial.available() > 0)   //if byte available
  {
     command[received_bytecount++] = Serial.read();  //receive
     serial_timer = millis(); //start second byte timeout counter;
```

```
  }
  if((serial_timer != 0) && (millis()-serial_timer > serial_timeout))
  {
     //second byte did not arrive on time
     received_bytecount  =  serial_timer = 0;
     command[0] = command[1] = 0;
     Serial.flush();
  }
  if(received_bytecount == 2)
  {
     //command and param have arrived
     switch(command[0])  //execute the command
     {
         case 2: ReadPot();break;
         case 3: analogWrite(Dimmer_Pin,command[1]);break;
         case 4: blinkrate = command[1]*10;break;//blinker rate 255
                                                  //steps of 10ms
         case 73: SetSpeaker(command[1]);break;
     }
     Serial.write((unsigned char *)sendbuffer,5);  //send the send buffer
     command[0] = command[1] = 0;
     received_bytecount  =  serial_timer = 0;
  }
}
```

5.2 RobotBASIC Programs Using the Firmware

In order to illustrate the power of the protocol we shall use programs that were developed in the book mention previously. The Piano.Bas program has no changes other than to allow for a different communications port and to add a two seconds delay just after acquiring the port to allow the Arduino to boot up and start waiting for commands. The program Program_05.Bas is extremely similar to another program in the book but with all hardware not supported our projects here removed from the code. The simple Program_04.Bas is not from the book. It is a very simple one with barebones action to illustrate how easy it can be to utilize the protocol.

A Simple Program

This program uses a very simple user interface to allow utilizing the hardware by varying its parameters and viewing its results. The idea is to show how the protocol can be utilized even with a very simple program. The main thing of interest is the **SendCommand()** subroutine. This routine sends the command and parameter bytes and then waits for the 5 bytes reply to arrive back with a timeout. If there is an error it displays a message saying so and returns a false. If there is no error it extracts the first byte which as per our hardware so far is the status of the pushbutton on the Arduino board. It also reconstitutes the 4th and 5th bytes into a 16-bit number (MSByte First). This, in our case, is the Pot level if a pot command is issued – which is the command being issued all the time unless another is requested.

Notice the use of the **SerBytesIn** command to do a timeout waiting for the 5 bytes to come in.

After running the program, try this: push the reset button on the Arduino and keep it pushed down. Notice how the program reports an error. Now allow the Arduino to reboot then use the program. Notice how it recovered so easily. This is an important advantage of the protocol. Error recovery is an important feature of the protocol.

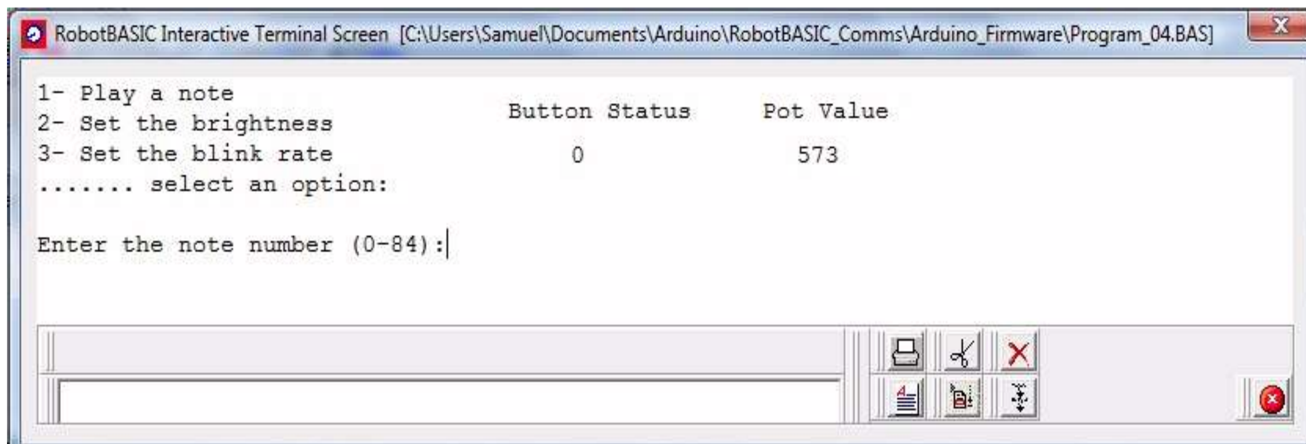


Figure 8: Program_04.Bas in action.

Program_04.Bas

```
//Arduino to RB Communications Protocol Implementation
//Program_04.Bas.....runs with Arduino_Firmware.Pre
Port = 3   //change this as per your system
Main:
  InlineInputmode
  setcommport Port,br115200
  print "Initializing ....please wait...."
  delay 2000
  C = 0 \ P = 0
  while true
    clearscr
    print "1- Play a note"
    print "2- Set the brightness"
    print "3- Set the blink rate"
    print "....... select an option:",crlf()
    repeat
        m = spaces(40)
        Call SendCommand(2,0,Button,Pot,Ok)
        if !Ok then m = "Comms error"
        xyString 600,10,m
        xyString 300,10, "Button Status     Pot Value"
        xystring 340,35,Button,spaces(15),Pot
        GetKey K \ K = char(K)
    until Instring("123",K)
    if K == "1"
      Input "Enter the note number (0-84):",N
      call SendCommand(73,N)
    elseif K == "2"
      Input "Enter the desired brightness level (0-255):",N
      call SendCommand(3,N)
    elseif K == "3"
      Input "Enter the blink rate (0-255):",N
      call SendCommand(4,N)
    endif
  wend
End
//------------------------------------------
```

```
sub SendCommand(Command,Parameter,&A,&B,&Status)
  A = -1 \ B = -1 \ Status = false
  if Command < 0 then return false
  ClearSerBuffer \ serialout Command,Parameter
  serbytesin 5,s,x
  if x < 5 then  return false
  A = GetStrByte(s,1)                            //get the Pushbutton
  B = (GetStrByte(s,4) << 8)+GetStrByte(s,5)  //create the return value
  Status = true
Return true
//----------------------------------------------
```

A More Advanced Program

In this program we provide a GUI system that allows the user to interact with the hardware through the protocol and to use all the available commands through a GUI interface. Compare this program to the one in the book mentioned earlier.

The program is not very hard to understand, however, it utilizes a few advanced techniques and tricks. It may look daunting, but the purpose is to illustrate what can be done. If you find it too complex do not worry. As the simple program above proved, you can still use the hardware and protocol even with a very simple user interface and straightforward code.

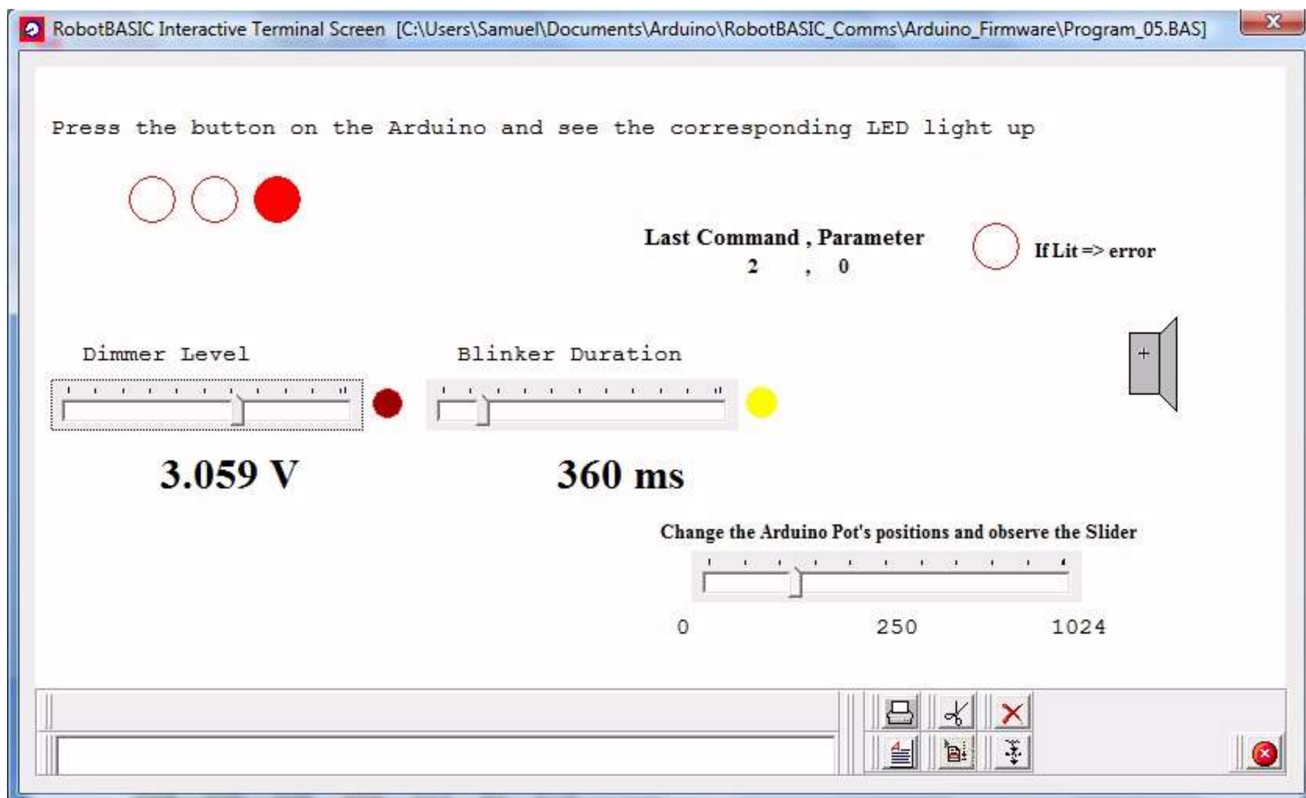


Figure 9: Program_05.Bas in action.

Program_05.Bas

```
//Arduino to RB Communications Protocol Implementation
//Program_05.Bas.....runs with Arduino_Firmware.Pre
Port = 3   //change this as per your system
Main:
  gosub Initialization
  GoSub SetAllLevels
  while true
    call CheckMouseKeyboard()
    if Command == -1 then call DoPot()
    call SendCommand(Command,Parameter)
    if timer()-P1_Timer > sValue[1]  //blinker time out
       P1_State = !P1_State \if !sValue[1] then P1_State=0
       CircleWH 455,205,20,20,FactorColor(ConsToClr(yellow),-20)\
                              ,LEDState[P1_State*2]
       P1_Timer = timer() //reset timer
    endif
  wend
End
//-------------------------------------------
Initialization:
  fnt = "Times New Roman"
  data LEDState;white,red,yellow
  setcommport Port,br115200   //set serial comms
  delay 2000
  xyText 390,100,"Last Command , Parameter",fnt,11,fs_Bold
  xystring 10,30,"Press the button on the Arduino and see the "+\
                 "corresponding LED light up"
  xyText 640,110,"If Lit => error",fnt,10,fs_Bold
  Spx = 700 \ Spy=170
  call DrawSpeaker(Spx,Spy)
  data sLabel;"Dimmer Level","Blinker Duration"
  data sCommand; 3,4
  data sValue; 0,0,127,127
  for i=0 to 1
    xyString 240*i+30,175,sLabel[i]
    AddSlider "P"+i,10+240*i,200,200,0,255
    SetSliderPos "P"+i,sValue[i+2]
    call MakeValue(i,sValue[i+2])
    xyText 80+240*i,245,MakeValue__Result,fnt,20,fs_Bold
  next
  call DrawDimmer(sValue[2])
  Command = 0 \ Parameter = 0
  P1_Timer = timer() \ P1_State = 0
  LastSlider()  'clear any slider events
  GoSub  InitPot
  onSlider sHandler
Return
//---------------------------------------------
SetAllLevels:
  for i=0 to 1 //force blinkers and dimmers to be set
    serialout sCommand[i],getsliderpos("P"+i)
  next
Return
//---------------------------------------------
```

```
sub MakeValue(which,Level)
  if which == 0  'dimmer
     n = 5*Level/255.0
     Level = Format(n,"0.000 V  ")
  elseif which == 1  'blink duration
     n = Level*10
     Level = Format(n,"####0 ms  ")
  endif
  sValue[which] = n
return Level
//----------------------------------------------
sub DrawDimmer(Level)
  CircleWH 205,198,37,37,white,white
  if !Level then return
  x = 3+(20*Level/255)/2
  Circle 225-x,215-x,225+x,215+x,red, RGB(Limit(Level,70,255),0,0)
return
//-------------------------------------------
sub SendCommand(Command,Parameter,&s)
  if Command < 0 then s="" \ return false
  xyText 450,120, Format(Command,"##0"+spaces(10))+\
            ",  "+Format(Parameter,"  ##0         "),_fnt,10,fs_Bold
  ClearSerBuffer \ serialout Command,Parameter
  _Command = -1 \ _Parameter = -1
  serbytesin 5,s,x
  circlewh 600,100,30,30,red,LEDState[(x<5)]  //set the error LED
  if x < 5 then s = "" \ return false
  for i=0 to 2
    circlewh 60+i*40,70,30,30,red,\
            LEDState[Getstrbyte(s,3-i)&1]  //set the LEDs
  next
Return true
//----------------------------------------------
sub sHandler()
  ls = LastSlider()
  if left(ls,3) !="Pot"
    _Parameter = GetSliderPos(ls)
    n = tonumber(substring(ls,2))
    _Command = sCommand[n]
    if n == 0 then Call DrawDimmer(_Parameter)
    call MakeValue(n,_Parameter)
    xyText 80+240*n,245,MakeValue__Result,_fnt,20,fs_Bold
  endif
  onSlider sHandler
return
//----------------------------------------------
InitPot:
  xytext 400,290,"Change the Arduino Pot's positions and"+\
                             " observe the Slider",fnt,9,fs_bold
  AddSlider "Pot0",420,310,250,0,1024
  enableslider "Pot0",false
  xystring 410,350,GetSliderMin("Pot0")
  xystring 650,350,GetSliderMax("Pot0")
  xystring 520,350,Format(GetSliderPos("Pot0"),"####0")
Return
```

```
//----------------------------------------
sub DoPot()
  call SendCommand(2,0,s)
  if !SendCommand__Result then return
  P = (getstrbyte(s,4)<<8)+getstrbyte(s,5)
  setsliderpos "Pot0",P
  xystring 520,350,Format(GetSliderPos("Pot0"),"####0")
Return
//----------------------------------------
sub DrawSpeaker(Spx,Spy)
  rectanglewh Spx,Spy,20,40
  line Spx+20,Spy,Spx+30,Spy-10
  lineto Spx+30,Spy+50
  Lineto Spx+19,Spy+39
  xystring Spx+5,Spy+5,"+"
  floodfill Spx+2,Spy+2,gray
  floodfill Spx+22,Spy,gray
return
//--------------------------------------------
sub CheckMouseKeyboard()
    P= 1\ C = -1 \ readmouse x,y,mb
    if mb=1
       if within(x,_Spx,_Spx+30) && within(y,_Spy,_Spy+40) then C=73 \P=80
                           //if speaker icon is clicked play note 80
    endif
    if C == -1 then return
    call SendCommand(C,P)
return
//--------------------------------------------
```

A Piano Keyboard

The interesting thing about this program is that it is exactly the same as another program that runs with a totally different hardware as shown in the book mentioned above.

The reason this program works with two totally different systems is that both obey the same protocol. Despite the fact that the implementation of the protocol on each is different the hardware is controllable by the very same software because the ***firmware*** (protocol) achieves a ***separation layer*** between the ***software*** (RB program) and the ***hardware***. Thus despite the hardware being different the software is able to control either version in the very same manner without any change to the software because the protocol is the same.

Another point of interest in this program is the way the ***hardware pushbutton*** can also be used to play the tune (Jingle Bells) or stop it playing. If you look at the firmware in the Arduino there is nothing there that makes it know what to do when the button is pushed. All it does is send it to RB. But the partnership between the software and firmware makes the hardware perform thing it was never programmed (directly) to do. This is a powerful advantage.

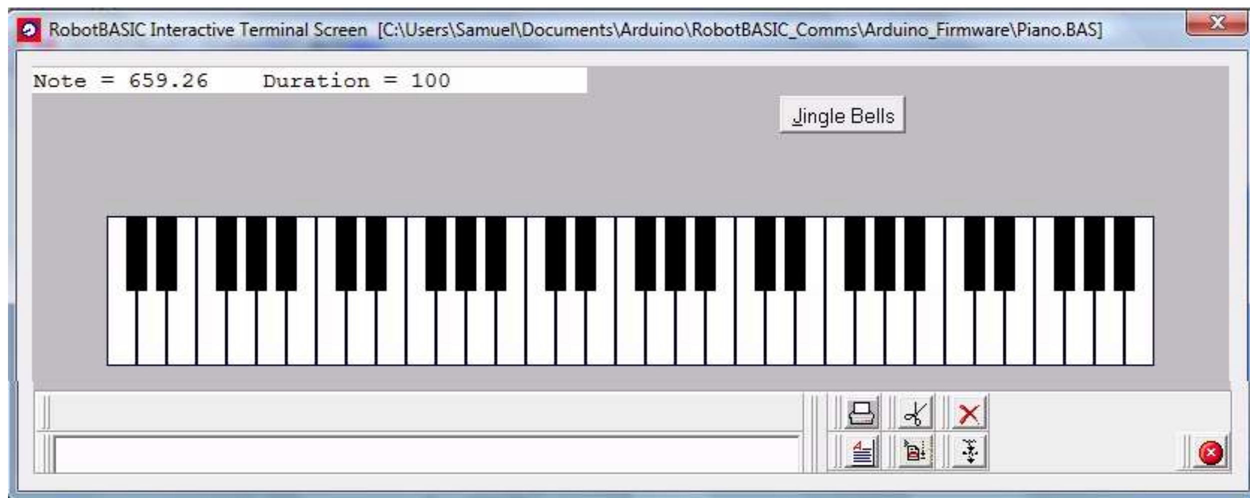


Figure 10: Piano.Bas in action.

Piano.Bas

```
//Arduino to RB Communications Protocol Implementation
//Piano.Bas.....runs with Arduino_Firmware.Pre
Port = 3 //set this as per your system
Main:
   GoSub Initialization
   while true
      call CheckMouse()
      while PlayTune
         call Play_Tune(Tempo)
      wend
   wend
end
//=========================================================
Initialization:
   GoSub SetUpNotes
   GoSub SetUp_Jingles
   clearscr gray
   WOffset = 50 \ WW=20
   for i=0 to 7*5-1  //draw the normal keys
      rectanglewh WOffset+WW*i,100,WW,100,rgb(0,0,50)
   next
   BW = 14 \ BOffset = 63
   for i=0 to 7*5-1  //draw the sharp keys
      rectanglewh BOffset+WW*i,100,BW,50,black,black
      if i#7 == 1 || i#7 == 5 then i++  //some sharps are not allowed
   next
   data NoteMap; 0,2,4,5,7,9,11
   data SharpMap; 1,3,0,6,8,10
   setcommport Port,br115200 \ delay 2000
   clearserbuffer
   PlayTune = false
   AddButton "&Jingle Bells",500,20
   onButton bHandler
Return
//=================================================================
```

```
sub bHandler()  //button interrupt handler
   lb = LastButton()
   if left(lb,3) == "&Ji"
      RenameButton lb,"&Stop"
      _PlayTune = true
   else
      RenameButton lb,"&Jingle Bells"
      _PlayTune = false
   endif
   onbutton bHandler
return
//=================================================================
SetUpNotes: //frequencies array
   data Notes;0
   data Notes;32.703,34.648,36.708,38.891,41.203,43.654
   data Notes;46.249,48.999,51.913,55.0,58.27,61.735

   data Notes;65.406,69.296,73.416,77.782,82.407,87.307
   data Notes;92.499,97.999,103.83,110.0,116.54,123.47

   data Notes;130.81,138.59,146.83,155.56,164.1,174.61
   data Notes;185.0,196.0,207.65,220.0,233.08,246.94

   data Notes;261.63,277.18,293.66,311.13,329.63,349.23    'middle C
   data Notes;369.99,391.99,415.31,440.0,466.16,493.88

   data Notes;523.25,554.37,587.33,622.25,659.26,698.46
   data Notes;739.99,783.99,830.61,880.0,932.33,987.77

   data Notes;1046.5,1108.7,1174.7,1244.5,1318.5,1396.9
   data Notes;1480.0,1568.0,1661.2,1760.0,1864.7,1975.5

   data Notes;2093.0,2217.5,2349.3,2489.0,2637.0,2793.8
   data Notes;2960.0,3136.0,3322.4,3520.0,3729.3,3951.1

   S=-2 \ P=-1\ C=0 \ CS=1 \ D=2 \ DS=3 \ E=4
   F=5 \ FS=6 \ G=7 \ GS=8 \ A=9 \ AS=10 \ B=11
Return
//=================================================================
SetUp_Jingles: //RTTTL codes for the tune
   Tempo = 1500
   data Song;S,5,E,8,E,8,P,32,E,4,P,32,E,8,E,8,P,32,E,4,P,32
   data Song;E,8,G,8,P,32,C,4,D,16,P,32,E,2,P,16
   data Song;F,8,F,8,P,32,F,8,F,16,P,32,F,8,E,8,P,32
   data Song;E,8,E,16,P,32,G,8,G,8,F,8,D,8,P,32,C,2
Return
//=================================================================
sub Play_Tune(Tempo)  //play all the notes in the tune's list
  Scale = 4
  FOR i = 0 TO MaxDim(Song,1)-1 step 2
     if Song[i] = _P  //if a pause
       Note = 0
       Duration = Tempo/Song[i+1]
     elseif Song[i] = _S   //if scale change
       Scale = Song[i+1]
```

```
       continue
     else
       Note = Song[i]+12*Scale+1    //determine the note number
       Duration = Tempo/Song[i+1]   //determine duration from tempo
     endif
     call PlayNote(Note,Duration)
     if !_PlayTune || !PlayNote__Result then _PlayTune = false \ break
  next
Return
//=================================================================
sub PlayNote(N,D,&s)
   xystring 1,1,"Note = ",Notes[N];"Duration = ",D,spaces(10) //display data
   s ="" \ c = 5
   if _Port == 0        //if not serial
     //sound Notes[N],D   //play on PC speaker...only XP machines
   else
     serialout 73,N     //send the command and note number
     delay D            //delay
     serbytesin 5,s,c
     if c == 5 then B = getstrbyte(s,1) //get the P5 button state
      if (B) & 0%001  //the buttons are active high and P5 is the LSBit
         if _PlayTune
            RenameButton "&Stop","&Jingle Bells"
         else
            RenameButton "&Jingle Bells","&Stop"
         endif
         _PlayTune = ! _PlayTune
         delay 200  //delay to eliminate button bounce
      endif
   endif
Return (c==5) //return true or false if 5 bytes received
//=================================================================
sub CheckMouse() //determine which key is pushed
     readmouse x,y,b                            //read mouse
     if !b then call PlayNote(0,1) \ return     //if no click then no sound
     c = pixelclr(x,y)                          //get the color under the mouse
     if c == white
       x = (x-_WOffset)/_WW                     //convert x to key number
       Scale = 1+x/7 \ Note = NoteMap[x#7]      //convert to note number and scale
     elseif c == black
       x = (x-_BOffset)/_WW                     //convert to key number
       Scale = 1+x/7 \ Note = SharpMap[x#7]     //convert to note number and scale
     endif
     if c == white || c == Black                //if there is a note
         Frequency = Note+12*Scale+1            //get its number
         call PlayNote(Frequency,100)           //play it
     endif
Return
```

Appendix A: Sending and Receiving Data

To send/receive data over a serial link (or using TCP and UDP) you are actually sending/receiving a ***chunk of bytes (a buffer)***. There are many means for creating this chunk or for extracting data from it, as will be described in Appendix B, but before you do so, you have to decide upon the meaning of the individual bytes in the buffer. Certain bytes can be text and others can be byte sized integers (0-255). Yet others can be either long integers (32-bits) or words (16-bits) or double precision floating point numbers (64-bits) or single precision floats (32-bits).

Text and byte numbers are straight forward, however when it comes to other numbers there can be some complications as we will see below.

The following are the actions needed to accomplish serial communications:

- Indicate the serial port to be used and the Com parameters like the baud rate and so forth (**SetCommPort** command).
- Clear the receive buffer (**ClearSerBuffer** command).
- Check the amount of data in the receive buffer (**CheckSerBuffer** command).
- Get bytes from the receive buffer (**SerIn** and **SerBytesIn** commands).
- Put bytes into the send buffer (**SerOut** and **SerialOut** commands).
- Change the timeout period (**SetTimeOut** and **GetTimeOut** commands).

A.1 Sending Text Data (ASCII data)

To send text or numbers converted to text use the **SerOut** command. This command takes parameters that can be either strings or numbers. If a parameter is a number it will be converted to its string equivalent. So the number 234 will be sent as three separate bytes with each byte containing the numbers 50, 51 and 52. Why these values? These are the ASCII codes of the alphanumeric digits 2, 3 and 4. If you want to send the number 234 as a binary one byte value of 0xEA see the next section.

So let’s say you want to send the string “hello” then the numbers 45 and 0xAD (=173) as strings. You would do this:

```
SerOut "hello ",45,0xAD
```

What will be sent over the serial line is 10 bytes with the following values:

```
104  101  108  108  111  52  53  49  55  51
 h    e    l    l    o    4   5   1   7   3
```

To send text data or numbers converted to their text equivalent use the **SerOut** command.

A.2 Sending Binary Byte Data

In the above section when we sent the number 173 (0xAD) we used 3 bytes to send it. Depending on the application this might be necessary. However, there is a more efficient way of sending a number. We can send the number as a binary value. An integer like 32128765 would require 8 bytes to send as text. It becomes even worse when we consider a float. π for instance would require 16 bytes if sent as a text. Conversely, if we send integers as binary values we would need 4 bytes for even the largest integer supported by RB, and floating point numbers can be sent as 8 bytes.

Sending binary numbers bigger than one byte over the serial line can be achieved in many ways. Which method you use depends on what format the receiver expects the data to be in. In this section we will consider sending numbers not bigger than 255 (0xFF), and thus they are one byte values and can be sent as one byte. In the next section we will have a look at how to send numbers bigger than one byte.

As in the previous section we want to send "hello" then 45 then 173 (0xAD). We do this:

```
SerialOut Left("hello there",5),32+13,0xAD
```

SerialOut as opposed to **SerOut** will treat numeric parameters as one byte integers. Even if the number is bigger than one byte it will be truncated to one byte. So if you have a number like 345 which is two bytes 0x0159, only one byte will be sent –the first byte from the right, or the least significant byte LSByte. So only 0x59 will be sent; ***therefore, be careful with this***. If you want to send numbers bigger than one byte see the next section.

In the above statement, instead of 45 we used 32+13 to illustrate that any parameter is actually an expression and you can either use a literal number (or string), or a calculated number (or string) and the calculation can of course use any functions or operators and variables and so on. See the Language Overview section in the RobotBASIC help file for a definition of an expression.

The statement above will cause 7 bytes to be sent over the serial line (notice the last two bytes are the actual numbers):

```
104  101  108  108  111  45  173
 h    e    l    l    o
```

To send data consisting of text and numbers not bigger than a byte (0 to 255) use the **SerialOut** command.

A.3 Sending Binary Multiple-Byte-Number Data

The method to use to send numbers bigger than one byte depends on the receiving device. This can become a complex situation with different processors expecting numbers like integers to be in formats that are not the same as other processors. For example on the PC (Intel) a 32-bit number (4 bytes) is stored in the ***Little-endian*** format, while on Motorola processors they are stored using the ***Big-endian*** format. This becomes an even bigger mess when you consider floating point numbers and negative numbers and so forth.

RobotBASIC has functions that facilitate manipulating a ***String Buffer*** as an ***array of bytes*** to insert integers, floats, bytes and text into it and read from it. See **BuffWrite(), BuffWriteB(), BuffRead(), BuffReadI(), BuffReadB(), BuffReadF(), toByte(), char(), PutStrByte(), GetStrByte(), ToNumber(), ToString()** and much more. Once you have formed a buffer with all the necessary binary data (and text if needed) you can send it using **SerOut** or **SerialOut** as a one string parameter.

For a more in depth information about creating buffers to send and extracting data from received buffers with many examples see Appendix B.

In this section we will consider sending a two-byte-integer. We will be using the Most Significant Byte First order (MSBF). That means that an integer value like 0xA3CD will be sent as two bytes 0xA3 then 0xCD. The 0xA3 will be sent first.

RobotBASIC integers are 4 bytes long. If you know that the value in the integer is no bigger than 0xFFFF (65535) then you can treat the integer as a two-byte integer. There is one other point of import that should be considered. If you want to use negative numbers you must consider an appropriate format. One ubiquitous format is the ***2's Compliment*** format. But this is another topic that is too complex to consider here. We will just stick to unsigned numbers. Later we shall see how to send Longs (4-byte integers) and 32-bit floats. See the functions **SignExtend16()** and **SignExtend8()** if you need to use 2's Compliment formats.

RobotBASIC has various bit-manipulation functions and operators. Operators like >> and << and &, @, ~ and | are able to manipulate bits in a number. Also functions like **MakeBit(), MakeByte(), GetBit()** and **GetByte()** can be used to set, reset or get particular bits or bytes from an integer.

So for example to obtain the second byte (from right to left and first byte is byte number 0) from the number 4567 stored in the variable **n**, you would use **`GetByte(n,1)`**.

To send the string "hello" and the number 4567 over a serial line with the format discussed above we do:

```
SerialOut "hello",GetByte(4567,1),GetByte(4567,0)
```

Or

```
SerialOut "hello", (4567 >> 8), 4567
```

SerialOut is much more convenient to use in this case but you can still use **SerOut** if you wish. Here is how:

```
SerOut "hello", char((4567 >> 8)), char(4567)
```

Notice that the **char()** function is used to make sure that the resulting integer will be made into one byte instead of being converted to a text equivalent of the number. You can also use **toByte()** instead. Although for sending binary bytes, it is much more convenient to use **SerialOut**.

A.4 Receiving Text Data (ASCII data) Or Binary data:

RobotBASIC has a receive buffer that can hold up to 4095 bytes. So an RB program can perform actions while data is being received from the serial port. However, you should make sure that any data in the buffer is read and stored before the buffer becomes full. It is a circular buffer so data that comes in after the buffer is full will start to write over the old data from the beginning again. ***Reading the buffer will clear it***.

There are three ways you can wait for the data to arrive into the buffer:

- ***Polling.*** With this method the RB program can be doing other actions and periodically check the buffer to see if the required number of bytes have arrived and then act accordingly. To do this use the **CheckSerBuff** command to obtain a count of the bytes in the buffer. Then when the time comes to read the buffer use **SerIn** to read the entire buffer into a string (array of bytes).

- ***Timeout Waiting***. In this method the program stops and waits for the right number of bytes to arrive. If the bytes arrive they would be read and returned into a string (array of bytes). If the right number of bytes do not arrive before a specific timeout period has elapsed then the characters that have arrived will be returned into a string and the count of how many into a variable and then the program flow goes on. Use **SerBytesIn** to get the required bytes and use **SetTimeOut** to set the required timeout if you need to change the default which is 5000 milliseconds.

- ***Interrupt***. This method utilizes event driven programming. Using **OnSerial** you can signal RB to branch to a particular subroutine (event handler) whenever the serial buffer receives a byte or more. The program will then be able to do whatever other processing is required, but whenever a byte comes in on the serial line the program flow will branch to the event handler routine. In the handler you read the bytes already there and store them and if more actions need to be taken they are carried out, but afterwards the program will go back to where it was before branching to the handler. ***This method is too complex for the scope of this paper and will not be discussed here.***

Whether you use the first or second method you will eventually end up with a string that has the bytes that were received. The first byte in the string is the first byte to have arrived and so on.

You can treat this string as a normal string. So if you know all the bytes in it are ASCII characters that represent text or numbers that were converted to their textual representations, you can use this string for all intents and purposes as any other string in RB. If you want to convert string numbers to actual numbers you can use **ToNumber()** to convert them to a numeric (integer or float depending on its value).

If the data contains binary numbers whether single byte or multi-byte numbers, you must treat the string as an ***array of bytes (buffer).***

For a more in-depth information about creating buffers to send and extracting data from received buffers with many examples see Appendix B.

There are various functions in RB that help in obtaining the value of a particular byte from an array of bytes (string). **BuffReadB()** treats the buffer as 0 indexed array (1st character is character number 0 and so forth) and **GetStrByte()** treats the buffer as a string and thus the 1st character is character 1 and so forth. However, both functions will return a numerical result which is actually an integer but with only the LSByte (first from right) filled with a value which is the value of the byte from the desired position in the buffer. You can also use any of the string functions like **SubString()** and so on. Additionally, see the functions **BuffRead(), BuffReadI(), BuffReadF(), BuffWrite(), BuffWriteB(),** and **PutStrByte()**.

Let's assume we have just received a buffer and we stored it in the variable **SB** and it has the following bytes (byte numbers start with 0 for the first byte):

Table A-1: Byte buffer byte order.

Byte Number	Content
0-4	Text string
5	Byte (8-bit) integer value
6-7	Word (16-bit) integer value using the Most Significant Byte First order (MSBF)
8-11	Long (32-bit) integer value using the Little-Endian format
12-15	Normal float (32-bits) as per the IEEE 754 standard format
16-23	Double precision float (64 bits) as per the IEEE 754 standard format

This will create the buffer above to test with the extraction code below:

```
SB = "Tests"+tobyte(205)+tobyte(401 >> 8)
SB += tobyte(401)+buffwrite("",0,35609)
SB += buffwriteF32("",0,3.234)+buffwrite("",0,5.334)
```

To extract the individual data from the buffer **s** we do the following:

```
Print Left(SB,5)       //prints the text (5 character from 1 to 5)
Print BuffRead(SB,0,5) //same as above using bytes 0 to 4
Print GetStrByte(SB,6) //prints the 8-bit number in the 6th character
Print BuffReadB(SB,5)  //prints the 8-bit number which is one byte
                       //and is the 6th character but is therefore
                       //byte number 5 in a zero indexed basis
Print (GetStrByte(SB,7)<<8)+GetStrByte(SB,8)//creates the 16-bit number
                                            //from characters 7 and 8
Print (BuffReadB(SB,6)<<8)+BuffReadB(SB,7)  //does the same as above…
                                            //notice indexing..bytes 6,7
Print BuffReadI(SB,8)    //prints the 32-bit integer from byte 8-11
Print BuffReadF32(SB,12) //the single precision float from bytes 12-15
Print BuffReadF(SB,16)   //the double precision float from bytes 16-23
```

The above should print out:

```
Tests
Tests
205
205
401
401
35609
3.23399996757507   ← notice loss of accuracy
5.334
```

BuffReadI(), **BuffReadF()** are other functions that can also help with numbers like floats and 32-bit integers. Read the RobotBASIC help about these functions carefully.

Appendix B: Manipulating a Byte Buffer (String)

The information in this appendix is important for achieving effective communications using a serial link or over the TCP and UDP protocols. In fact the information in this section is also of importance to performing Low Level File I/O as well as for utilizing the suite of USBmicro functions to control the U4x1 USB I/O devices.

In functions that send and receive data over a serial link or using the TCP and UDP you are in fact sending and receiving a byte buffer that contains the data to be transferred. This buffer can be manipulated in either of two ways:

- **As a string** which can be manipulated with numerous string functions. There are functions to extract parts of the string, extract a particular character, extract a particular character as a byte, insert characters or bytes.

All string functions index characters starting with 1; the first character is index position 1.

- **As a byte array** which can be managed with a set of specialized functions and commands to insert in it or extract from it bytes, integers, floats, or text.

All buffer functions index bytes starting with 0; the first byte is index position 0.

We shall refer to the buffer sometimes as a ***string buffer*** and at other times as a ***byte buffer***. It does not matter how it is referred to, it is an ***array of bytes***. The individual bytes can be ASCII characters (actual text) or they can be binary values. As the programmer, you are mainly interested in how to extract data from the buffer when you receive it and how to insert data into the buffer in preparation for sending it. To do these operations RobotBASIC provides a suite of functions and commands for treating the buffer as an array of bytes. Additionally all the string functions in RB can be used to handle the buffer as a string.

B.1 Putting text and textual numbers in a buffer (string)

The buffer is in fact a string variable that can be handled in all the normal ways you treat any normal string variables. So for example if you want to have a buffer that consists of the text “Hello there” then you would do:

```
SB = "Hello there"
```

The buffer becomes the variable **SB**, and it now contains 11 bytes which have the values:

```
72 101 108 108 111 32 116 104 101 114 101
 H  e   l   l   o      t   h   e   r   e
```

If you later say:

```
SB = SB+" "+ (-45.7) //conversion of the numeric is performed implicitly
```

Or you can say

```
SB = SB+" "+ToString(-45.7) //conversion is performed explicitly
```

The buffer will now be holding 17 bytes with the additional 6 bytes being:

```
32 45 52 53 46 55
    -  4  5  .  7
```

Notice how the number has been converted to its textual representation and that the characters in the string are the ASCII codes for the individual digits of the number including the minus sign and the decimal point.

You can build a string buffer this way with as many expressions as you need over as many lines of code as required. Once the variable **SB** is populated with the data it can then be used as the parameter for the **SerialOut, SerOut, UDP_Send(), TCPS_Send()** or **TCPC_Send()** functions which send the bytes in the string over the corresponding protocol.

A command you may find useful for populating the string (buffer) with text or textual numbers is the **BuffPrintT** command. This command is very much like the **Print** command except it will put the resulting text of the expressions in a specified string (buffer). Numbers will be put as their textual representations and you can even use **;** and **,** to control tab spacing just like you would with the **Print** command. For example:

```
X = 3.5e200
BuffPrintT SB,X," * ",2," +5 = ",2*X+5,
BuffPrintT SB,sRepeat(" ",3);"OK",
```

Notice that with the first usage of **BuffPrintT**, **SB** does not exist and therefore will be created as an empty string. With the second usage, **SB** already exists and has data in it, so the new data will be appended to it. Remember this when using this command. If **SB** already exists and you do not wish to append to it you will need to say `SB = ""` to make it into a blank string before you use it with the command.

After the above sequence of code **SB** will contain:

```
51 46 53 69 50 48 48 32 42 32 50 32 43 53 32 61 32 55 69 50 48 48 32 32
3   .  5  E  2  0  0     *     2     +  5     =     7  E  2  0  0
32 32 32 32 32 32 32 32 79 75
                        O  K
```

Notice the **,** at the end of each line in the program. This is necessary if you do not want a CR/LF [char(13)+char(10)] to be part of the buffer **SB**. The following lines:

```
X = 3.5e200
BuffPrintT SB,X," * ",2," +5 = ",2*X+5
BuffPrintT SB,sRepeat(" ",3),"OK"
```

Will cause **SB** to have the following:

```
51 46 53 69 50 48 48 32 42 32 50 32 43 53 32 61 32 55 69 50 48 48 13 10
3   .  5  E  2  0  0     *     2    +   5     =    7   E  2  0  0
32 32 32 79 75 13 10
          O  K
```

Notice the characters with the byte values 13 and 10 right after 200 and after the OK. These are the Carriage Return and Line Feed character pairs that normally result in a new line. Since **BuffPrintT** behaves just like a **Print** then these characters will be there if you do not end the command with a comma to stop it from inserting a CR/LF character pair.

B.2 Putting text and binary numbers in a buffer (string):

As far as text is concerned all the details in Section B.1 apply. As far as binary numbers there are three situations:

- Byte numbers, which are numbers that range from 0 to 255.
- Integers, which are 4 bytes long (in RB) and range from **MinInteger()** to **MaxInteger()**
- Floats are 8 bytes long (in RB) and can range from +/-**MinFloat()** to +/-**MaxFloat()**. Floats in other systems can be 4 bytes long.

Bytes

RobotBASIC does not have a byte type. Nonetheless, an integer can be truncated to become a byte. There are two functions that can convert an integer to a byte value that will be usable for adding to a string (buffer). These are **Char()** and **toByte()**. They do the exact same job but are named so as to be appropriate for whatever situation requires the function.

Both functions in reality return a buffer (string) of one byte (character). The byte will have the value of the **Least Significant Byte** (LSByte) of its integer parameter. This is the first byte from the right if you represent the number as binary (or hex).

For example if you say `SB = char(156)` or `SB = toByte(156)` **SB** will become a one character string (one byte buffer) with the byte having the value 159.

However if you say `SB = char(456)` or `SB = toByte(456)` **SB** will become one character string (one byte buffer) with the byte value being 200. Why 200? Because if you look at the hex equivalent of 456 (=0x01C8) you will see that it is 2 bytes long and the LSByte is 0xC8 (=12*16+8 = 200).

For example if you want to create a buffer with the text “Hello” and the byte numbers 34 and 211, you would write:

```
SB = "Hello"+char(34)+char(211)
```

OR

```
SB = "Hello"+toByte(34)+toByte(211)
```

SB will then contain the following bytes (notice the last two bytes are the numbers specified):

```
72      101     108     108     111     34      211
H        e       l       l       o
```

Another function that can be more convenient in certain instance is PutStrByte(). With this function you can insert a byte value at a specific position in a string (buffer). For example:

```
SB = "Hello"
SB = PutStrByte(SB,Length(SB)+1,34)
SB = PutStrByte(SB,Length(SB)+1,211)
```

The above code will result in **SB** being exactly as listed above. This looks more complicated in this situation; the previous method is more convenient. However, if you read the details of this function in the RobotBASIC Help file, you will see that it facilitates certain actions that are required in certain situations and is a good function to remember when the need arises. Its converse **GetStrByte()** is more frequently needed.

BuffWriteB() is another function that is very similar to **PutStrByte()** but treats the buffer as a byte array and thus indexes using 0 as the first byte and so forth. The following example will result in **SB** being exactly as above:

```
SB = "Hello"
SB = BuffWriteB(SB,-1,34)  //-1 means the end of the existing buffer
SB = BuffWriteB(SB,-1,211)
```

Read about these two functions in the RobotBASIC Help file. They are useful and once you know how they work you will see that they are needed and necessary in certain situations that you may not appreciate from the simple example given above (in the next section for example).

Integers and Floats

How floats and integers are represented depends on the computer you are using. RobotBASIC runs under the Windows operating system which usually runs on a PC that has an Intel processor. Integers supported by RB are 4 bytes long and are stored in what is called the ***Little-endian*** format. This format stores the 4 bytes of an integer in the order from left to right with the LSByte as the first byte.

Therefore, a number like 0xA412B8D7 will be stored as D7,B8,12,A4. Notice it is reverse to the way we normally look at binary numbers. This is just the way it is and you just have to accept it (actually it makes sense if you consider the low level data transfer mechanisms within the processor's circuitry).

Some processors (e.g. 68HC11) store integers in the Big-endian format (reverse) and if you are going to send numbers to devices based on these processors, it makes a difference what format is used. However, if you are going to send buffers between machines using the Little-endian format, you do not have to be concerned with how the numbers are stored, RB takes care of that.

With floats they are stored in a format called the IEEE 754 standard. This can get quite complex and shall not be discussed here. Just know that a float in RB is stored as 8 bytes long, what these bytes are and what values and order and so forth is immaterial. RB will take care of it. RB also has functions to insert a float as a 32-bit float (4 bytes) called a single precision float.

The command **BuffPrintB** is one way to create a buffer with binary integers and floats in it. For example:

```
X = 0x00A243C1
BuffPrintB SB,X,"*",2,"+5=",2*X+5,"OK"
```

This will result in the variable **SB** holding the following bytes:

```
193 67 162 0  42 2  0  0  0  43 53 61 135 135 68  1 79 75
C1  43  A2 00 *  02 00 00 00  +  5  =  87  87 44 01  O  K
```

The numbers in bolded red are the hex values of the bytes and are arranged in the Little-endian format for 0x00A243C1 (=10634177) and 0x01448787 (=21268359 = 10634177 *2+5) so you see how **BuffPrintB** can be quite useful.

Notice how there is no CR/LF after OK even though there was no comma at the end. This is because **BuffPrintB** does not behave quite like the **Print** command. It is a binary formatter and will not therefore insert CR/LF as is required with text. Also there is no tabbing. A semicolon will have no effect. If you wish to insert a CR/LF use the **CrLf()** function.

Notice how the numeric value 2 was inserted as an integer (4 bytes) even though it can fit in a byte. This is because the **BuffPrintB** command will not make assumptions. If you wish to treat an integer as a byte you need to convert it to one.

Notice the blue bolded code in the following lines:

```
X = 0x00A243C1
BuffPrintB SB,X,"*",toByte(2),"+5=",2*X+5,"OK"
```

This will result in **SB** holding the following bytes (notice how 2 is now only one byte long):

```
193 67 162 0   42  2  43 53 61 135 135 68  1 79 75
 C1 43  A2 00   *  2   +   5  =   87  87 44 01  O  K
```

The functions **BuffWrite()** and **BuffWriteB()** are also of use. The following example will result in **SB** holding the same bytes as the example just above.

```
X = 0x00A243C1
SB = BuffWrite("",0,X)
SB = BuffWrite(SB,-1,"*")+BuffWriteB("",0,2)+"+5="
SB = BuffWrite(SB,-1,2*X+5)+"OK"
```

B.3 Extracting text and numbers from a buffer (string)

When you receive a buffer with data in it you must know in what arrangement the data is organized. There has to be an agreement between the sender and receiver so that data can be inserted and extracted in the correct manner. This is especially important when there is a mixture of number types and text. This is best illustrated with a concrete example.

An example

We are going to create a buffer that is a ***record*** of data. The record will be divided into ***fields***. Let's say you are sending a database with data about people. Each record will be transmitted as one buffer. In each of our hypothetical records there are the following fields: **Code, Name, Address, Zip_Code, Balance.**

In our example **Code** will be considered to be a byte, **Zip_Code** will be an integer and **Balance** a float.

We must also decide the following:

- Will the numbers be stored in the record as strings or as binaries?
- Will the text fields be of fixed lengths or variable lengths?
- If the text fields are to be of variable lengths how do we know where they end?

These questions have to be answered carefully in order to be able to store the data in the buffer to be transmitted, but even more crucially, so as to be able to extract the values for each field correctly.

Let's say we used the following code to create the buffer **SB** to hold the record:

```
Code    = 1
Name    = "Sam"
Address = "Here and there"
Zip_Code= 55667
Balance = 100.23
SB = ToString(Code)+Name+Address+Zip_Code+Balance
```

Will result in **SB** holding

```
1SamHere and there55667100.23
```

There is no problem at all in creating the buffer. The problem, though, arises when we try to extract the data from the buffer. Where does the **Name** field start and end? Where does the **Zip_Code** field start and end? As you can see we have not created the buffer in a good way. This code is a lot better:

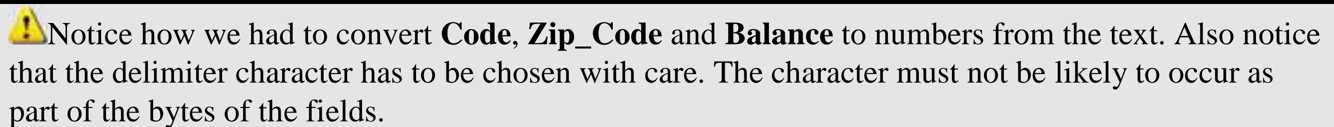
```
D = "|"
BuffPrintT SB,Code,D,Name,D,Address,D,Zip_Code,D,Balance,
```

Will result in **SB** holding

```
1|Sam|Here and there|55667|100.23
```

With a record like this it is very easy to extract the various fields:

```
D = "|"
Code     = toNumber(Extract(SB,D,1),0)  //defaults to 0 if bad text
Name     = Extract(SB,D,2)
Address  = Extract(SB,D,3)
Zip_Code = ToNumber(Extract(SB,D,4),0)  //defaults to 0 if bad text
Balance  = ToNumber(Extract(SB,D,5),0.0)//defaults to 0.0 if bad text
```



Notice how we had to convert **Code**, **Zip_Code** and **Balance** to numbers from the text. Also notice that the delimiter character has to be chosen with care. The character must not be likely to occur as part of the bytes of the fields.

Another design

With the above scheme we stored the numbers in the buffer as text. This can be wasteful. For instance, if **Code** is 234 it will occupy 3 bytes (‘2’, ‘3’, and ‘4’). Conversely, if we store it as a byte value, it will only be one byte. Also notice **Zip_Code** is 5 bytes long, but if we store it as an integer we would save one byte. Additionally, the **Balance** field is a float. Imagine if Sam was a rich guy and had 10,000,000.23 in his account (wishful thinking). That would require 11 bytes to store as a text (no commas). You can see that it is better to use 8 bytes to store the float as a binary rather than text.

However, if we store numbers as binary there will be no possible delimiter character to use to delimit where the text fields end since binary numbers could be any values and there would be no way of having a byte value that cannot occur as part of a field. The problem is not with the numbers since we know their lengths; it is the text fields that pose a problem.

A possible solution is to fix the length of the text. So we would say that **Name** cannot be longer than 20 characters (bytes) and if it is less than 20 it is padded with spaces (see **JustifyL()**). However, this is wasteful and limiting. If the name is a lot shorter than 20 characters we would be storing too many unnecessary characters. If it needs to be longer than 20 characters then we have deteriorated the flexibility of the application.

The most efficient and flexible design

A better solution is to store a number before each text field that specifies the length of the text to follow. If we know that the text cannot be any longer than 255 character we can store the number as a byte. If 255 is too short then we can store the number as an integer (4 bytes). Either way is a good method and will result in the most efficient usage of the buffer.

These program lines:

```
Code    = 1
Name    = "Sam"
Address = "Here"
Zip_Code= 55667
Balance = 100.23
BuffPrintB SB,toByte(Code),Length(Name),Name
BuffPrintB SB,Length(Address),Address,Zip_Code,Balance
```

Would result in **SB** holding the following bytes:

```
1 3 0 0 0 83 97 109 4 0 0 0 72 101 114 101 115 217 0 0 31 133 235 81 184 14 89 64
1    3     S  a  m     4    H    e    r    e     55667                100.23
```

The bytes in blue are one byte numbers. The bytes in red are the Little-endian format for the integers and the bytes in purple are the IEEE 754 representation of the float.

Accordingly, we now send the buffer **SB**. On the receiver side, 28 bytes would be received and stored in a buffer variable (say **SB**). How do we extract the individual fields from the buffer? Remember, we know that just before every text field there is an integer that indicates how long the text that follows is.

Here is how we can extract the fields' values:

```
X = 0
Code    = BuffReadB(SB,X)  \  X++
n       = BuffReadI(SB,X)  \  X += BytesCount_I
Name    = BuffRead(SB,X,n) \  X += n
n       = BuffReadI(SB,X)  \  X += BytesCount_I
Address = BuffRead(SB,X,n) \  X += n
Zip_Code= BuffReadI(SB,X)  \  X += BytesCount_I
Balance = BuffReadF(SB,X)  \  X += BytesCount_F
```

We are able to specify the exact positions for reading the various fields from the array of bytes (buffer) using the appropriate **BuffRead/B/I/F()** function. Notice a postfix of **I** means integer, a postfix of **F** means float (64-bits), **B** means byte and no postfix means text. Notice that with the text form of the function we must also specify the number of bytes to be read. This is where the previously extracted integer that specifies the length comes in use. Also notice how we keep track of the next position within the buffer to read from using the counter **X**. The constants **BytesCount_I** (4 for now) and **BytesCount_F** (8 for now) are defined in RobotBASIC to specify the lengths of an integer and a float in bytes, so you won't have to fix these values in case of future modifications to the internal representation of integers and floats within RB.