Utilizing The TCP & UDP Protocols Within RobotBASIC

RobotBASIC has a very easy to use interface that facilitates a communication link between networked computers over a Local Area Network or through the Internet. With just a handful of functions you can implement a bidirectional communication conduit between two or more networked computers using either the Transmission Control Protocol or the User Datagram Protocol.

With RobotBASIC's collection of TCP and UDP functions even the novice programmer can easily implement a whole range of interesting projects that would otherwise challenge an expert even with advanced development tools.

Imagine being able to collect instrumentation data on a PC in Australia and sending the data to a machine in the USA for display purposes and for parameter settings and so forth.

Imagine being able to text chat with a friend in Australia while you are in the USA. Yes, yes; you can do that on numerous web sites. But think of the pleasure of writing the program that can do this, yourself.

The two programs in the zip file TCP_Demo.zip were used to do precisely the above two actions between a PC in Australia and a PC in the USA. The data transfer was nearly instantaneous and the chat was great fun.

Imagine remotely controlling a robot in Australia through a PC in the USA. Also sending snap shots of the robot's environment to the controller side. In fact that is exactly what the programs in the zip file TCP_Robot.zip were used to do. Again, the control was nearly instantaneous, and the screen shots took a second over the LAN and a little longer half way around the world.

The programs above were all developed in RobotBASIC. The longest program is 179 lines of code and that includes 45 lines of instructions. Also see the demo programs in the RobotBASIC Help file in the TCP and the UDP sections.

1- Scope Of The Article:

This article deals with the details of how to use RobotBASIC's functionalities to send data between networked computers. The article does not attempt to explain the principles of networking using the TCP and UDP protocols; this would require an entire book. Nevertheless, as you shall see, there is no need to have such detailed knowledge to implement powerful networking projects with RobotBASIC.

2- Some Terminology:

Throughout the article there will be reference to certain terminologies that pertain to the field of networking. It is necessary to have a working knowledge of what these terms signify to be able to effectively utilize the functionalities provided in RobotBASIC. The given explanations are for practical use and are not meant to be an in depth explanation.

There are numerous ways of implementing a network using a plethora of hardware. It is not feasible to cover all possible combinations, so only the setup shown in Figure 1 will be considered. If your network differs from the arrangement shown you will need to consult with a network administrator if you wish to communicate across the Internet so as to handle firewalls and other issues. However, if you will be confining your projects to communicating computers inside the same LAN then the information in this article is all you need. Appendix B discusses how to configure the arrangement in Figure 1 to enable communication between two PCs across the Internet.

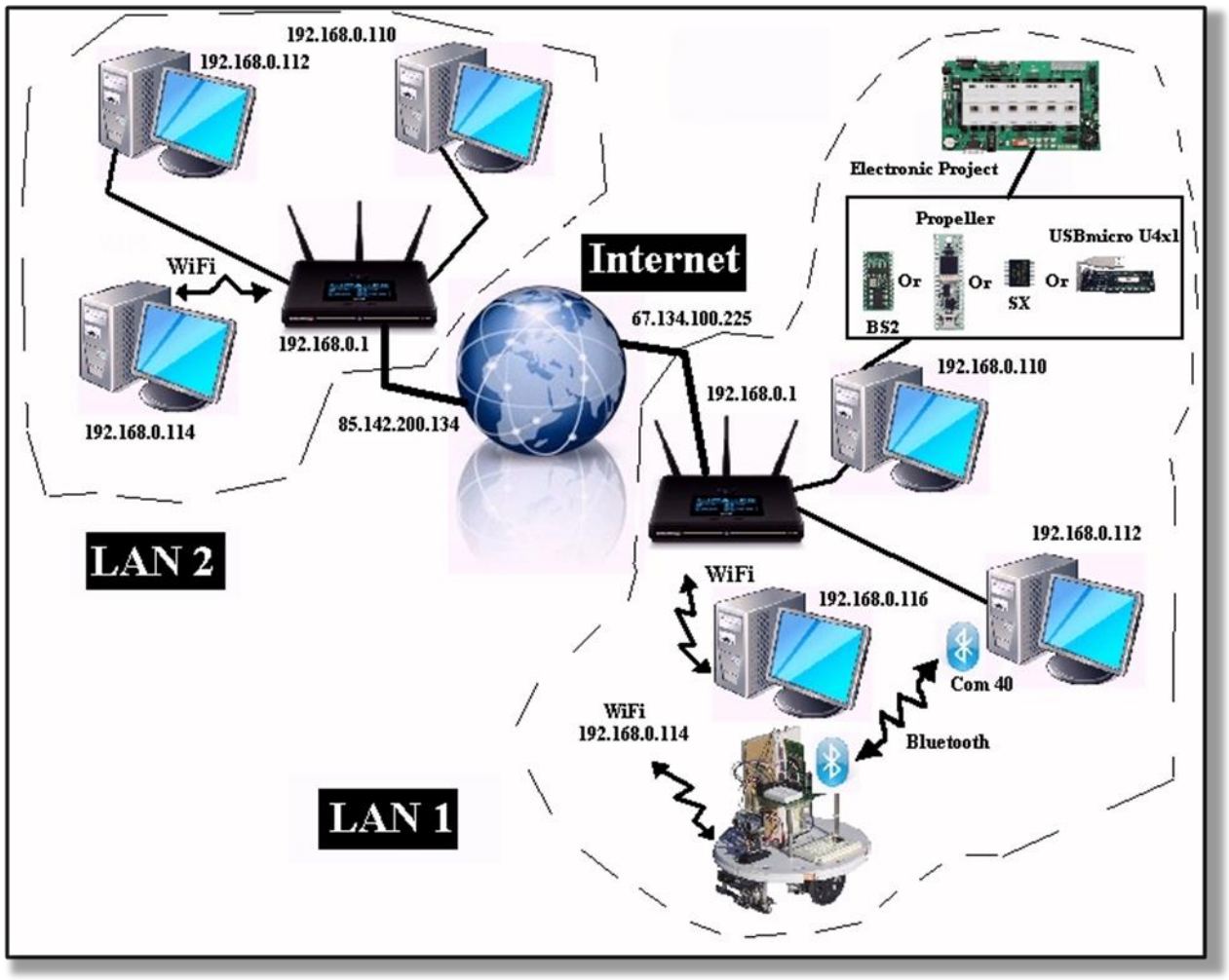


Figure 1: A Typical Network Arrangement.

In Figure 1 notice that one of the computers is connected to electronics. This is to indicate that any or all of the computers on the network can be performing data acquisition or control through external instrumentation. Furthermore, any or all of the computers can carry out additional channeling of communicated data over a wireless radio or through Bluetooth to a robot or any other microcontroller performing instrumentation. Additionally, you can have any number of autonomous robots that have their own WiFi TCP or UDP abilities and thus can be communicated with directly from any computer on the network (not through a secondary channel) just as if they were a computer on the network, which in fact, they would be.

A Local Area Network (LAN) is a group of computers dispersed over a relatively small area such as a home or office building. The machines are interconnected using physical wires or wirelessly using some interconnection system. Usually, there is a central server that serves as the communications coordinator. This server can be a computer or a specialized device called **Router**.

An Internet Service Provider (ISP) is a company that provides a computer system through which clients can obtain connectivity between their LANs and the **Internet**.

The Internet is a very large and complex network of interconnected ISPs around the world. An ISP provides a method for a computer in one LAN to be able to communicate with another computer in another LAN by managing the routing through other ISPs until eventually reaching the ISP of the remote LAN and then on to the final target computer.

A Router is a device that manages the connectivity of the various machines in the LAN. Computers in the LAN can be connected to the router through a physical wire or by **WiFi**. A router is sometimes referred to as a **NAT (Network Address Translator)** and often serves as the connection point between the LAN and the ISP which then provides a link to the Internet. Additionally, the Router often acts as a **Fire Wall** as well as a **DHCP (Dynamic Host Control Protocol)**.

WiFi is a system of hardware and software for linking a computer to the LAN wirelessly. For all intents and purposes the computer will appear as if it is connected to the rest of the LAN over a wire. Other than the convenience of mobility and lack of cumbersome wires the computer is no different, from the network's point of view, than a wired computer.

A Fire Wall is software and/or hardware used for blocking access from outside a LAN to the computers inside the LAN. This is a safety measure to prohibit illegitimate access to the computers in a LAN. A Fire Wall can also limit access from inside the LAN to the outside. You can configure a Fire Wall to allow certain communications while blocking others.

A Dynamic Host Control Protocol (DHCP), in short, assigns an **IP address** to each computer on the LAN.

A Network Address Translator (NAT) makes sure that computers inside the LAN appear to the outside world as valid computers with valid global IP addresses even though their actual IP addresses are only local addresses with significance only inside the LAN.

An IP Address is basically the name of the computer. It is a set of 4 numbers separated by a dot (**e.g. 192.168.0.120**), however the whole address is not a number; rather it is a text. Each individual number in the 4 fields ranges from 0 to 255. The network will not function without some method of addressing a particular computer and the IP address is this method.

A Socket is the endpoint of a bidirectional communication flow across the network. When data flows between two computers or when a connection is established between them it is achieved through a complex program called a Socket. This program is part of the internal structure of RobotBASIC with which you interact through a handful of function calls. To carry out TCP communications you do it through a TCP Client Socket (TCPC) to and from a TCP Server Socket (TCPS) [see later for more details]. Communications over the UDP are between two UDP Sockets (see later for more details). A Socket is identified by the IP address of the machine it is running on and a **Port number**. ***The port number must not be previously assigned to any other socket (UDP or TCP) on the same IP address.***

A Port Number is a further subdivision of a particular IP address. A machine has one IP address but it may have various Sockets (programs) that provide access to the LAN. These Sockets will be addressed on each machine through an additional number that can be considered as a sub-address on the machine. This number is called the Port number and is an actual 16-bit number that ranges from 1 to 65535 (0xFFFFF). Think of an IP address as the street address of a building and the port number as the number of particular apartment in the building. The building is the computer and apartments are the various Sockets (programs) running on the computer. ***This is why a port number associated with a socket must not be in use by any other active socket.***

The Transmission Control Protocol (TCP) is a sophisticated standard for moving data over a network. There is no need to understand this standard in depth in order to use the functions in RB. TCP is a client-server protocol. A server socket can accept connections from multiple client sockets but a client socket can only be connected to one server socket at a time. Once a client connects to a server it becomes as if there is a direct wire between them and data can be exchanged between the two sockets. See later for more details.

The User Datagram Protocol (UDP) is a simpler standard than TCP for exchanging data over a network. It is not necessary to know the details of the protocol to be able to use the functions in RB. UDP is not a connection oriented protocol. A UDP socket can send to any other UDP socket without establishing a link first. There is no exclusive or maintained connection between the two machines. See later for more details.

The Simple Mail Transfer Protocol (SMTP) is a protocol that ensures that an email is able to reach the intended recipient's computer from the sender's computer through the ISP of the sender around the Internet and finally to the ISP of the recipient and on to the recipient's computer.

A Remote IP Address is the address of the machine that is hosting the destination socket. That is the remote socket to which the data is going to.

A Local IP Address is the address of the machine that is hosting the originating socket. That is the socket being used to send data.

A Packet is a set of bytes (buffer or string) that is sent in one send action using TCP or UDP. When you send data over the protocol you are sending multiple bytes in one go. This is called a packet. A big amount of data can be sent in multiple chunks (Packets). UDP has a limit on the size of a packet (2048 bytes) while TCP does not.

3- Manipulating A Byte Buffer (String):

Note: You may wish to skip this section and go on to Section 4. However the information here is important for achieving effective communications using the TCP and UDP protocols, therefore, you will need to come back and read this section in detail. In fact the information in this section is also of importance to performing Serial I/O (see RobotBASIC_Serial_IO.pdf) and 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 using the TCP and UDP you are in fact sending and receiving a byte buffer that contains the data to be transferred between sockets. This buffer can be manipulated in one of two ways:

1- **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.
   Note: ***All string functions index characters starting with 1. That is the first character is index position 1.***
2- **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.
   Note: ***All buffer functions index bytes starting with 0. That is 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 all you are interested in is 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.

3.1- 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 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 the 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 32 32 32 32 32 32 32 79 75
3   .  5  E  2  0  0     *     2     +  5     =     7  E  2  0  0                              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 32 32 32 79 75 13 10
3   .  5  E  2  0  0     *     2    +   5     =    7   E  2  0  0                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.

3.2- Putting text and binary numbers in a buffer (string):

As far as text is concerned all the details in Section 3.1 apply. As far as binary numbers there are three situations:

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

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 (section 3.2 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).

Some processors (e.g. 68HC11) store integers in the Big-endian format (reverse) and if you are going to be sending numbers to devices based on these processors then it makes a difference what format is used. However, if you are going to be sending buffers between machines using the Little-endian format then 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.

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 area in the following code:

```
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 *  02  +   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 (also see section 3.3 below).

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

3.3- 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 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 the Code will be considered to be a byte, the Zip_Code will be an integer and the 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:

```
BuffPrintT SB,Code,"|",Name,"|",Address,"|",Zip_Code,"|",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:

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

Notice ***how we had to convert the zip code and balance to a 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 the zip code; it 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 length, 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 will be padded with spaces (see JustifyL()). However, this is wasteful and limiting. If the name is a lot shorter than 20 characters then 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= X+1
n=BuffReadI(SB,X) \ X=X+BytesCount_I \ Name=BuffRead(SB,X,n) \ X=X+n
n=BuffReadI(SB,X) \ X=X+BytesCount_I \ Address=BuffRead(SB,X,n) \ X=X+n
Zip_Code=BuffReadI(SB,X) \ X=X+BytesCount_I
Balance=BuffReadF(SB,X) \ 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, 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) and BytesCount_F(8) 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.

4- Utilizing The UDP System:

In RobotBASIC ***you can create multiple UDP sockets in a program***. Each socket must be assigned a unique and not currently in use Port number. When created, the socket will *automatically* use the IP address of the machine it is created on, but, you must assign it a particular port number.

Note: ***You must ensure that the port assigned to a UDP socket is unique and is not being used by another active socket (UDP or TCP) in the same program or other programs on the same machine. BE SURE OF THIS.***

To create and activate a UDP socket use UDP_Start(se_Name{,ne_ListenPort}). Notice that the port number is optional. If you do not specify a port number it will be assumed to be 50001 (see Appendix A for a list of preferred port numbers).

Another parameter of the function is the socket name. When you start a UDP socket you assign it a name. This name is then used to refer to the socket in further functions that utilize it. ***The name therefore has to be unique (within a program not across other programs) and is case sensitive.***

Once a socket is started you can begin to use it to communicate with other UDP sockets. These other sockets can be on the same machine (IP) or on a remote machine.

Note: ***You must not try to communicate with a non-existing socket on the same machine (IP) as the socket you are using. It is not a problem if you try to communicate with a non-existing socket on a remote machine, data will just not get there since the socket does not exist. However, there is a problem with the Windows 2000 and Windows XP operating systems (not others) where if you try to send data to a socket that does not exist on the same machine as the sender socket the OS hangs. Therefore, Do Not Send Data To A Non-Existing Socket.***

The functions discussed in the following sections enable the writing of a program to send and/or receive data packets over the UDP. A socket in a RobotBASIC program can communicate over the LAN or Internet with another socket that does not have to be another RB program. As long as the data format in the send/receive buffer is in accordance with a format that both sockets are in agreement upon, the other socket can be:

a- The same PC running another RB program.
b- The same PC running a non-RB program.
c- Another PC with another RB program.
d- Another computer (not just a PC) running a non-RB program.
e- A device (e.g. robot or microcontroller) that can use the UDP.

4.1- Sending data using the UDP:

To send data using a UDP socket use UDP_Send(se_Name,se_Data,se_TargetIP,ne_TargetPort). Notice that you must specify the destination IP and Port number. The port number is a numeric and the IP is a string but must be of a legal format ("n1.n2.n3.n4").

Notice also that you must specify the name of the socket you want to use to send the data. If you have started multiple sockets in a program you will need to specify which one to use to send the data. It does not matter which socket you use to send data. What matters is that you specify the IP and Port of the target socket. Nevertheless, you need to specify the name of the socket to be used in order to tell RB which one to use, even if you only have one.

The parameter se_Data is the data buffer to be sent. See Section 3 for how to create and manipulate the buffer.

You must specify the IP and Port number of the destination socket every time you send data through a UDP socket. This is because the socket does not actually establish a connection with the destination. It does not even guarantee that the data has arrived. The only error reporting is if the destination IP does not exist. If the destination IP does exist the data is sent even if the destination socket does not

and there is no way of knowing this. It is as if you are sending mail to an apartment in a building that is not occupied. The mail will be put through the door but there is no one there to read it. All that the mailman cares about is that there is a building.

UDP is a connectionless protocol in that there is no server-client or even pier-to-pier connection. A UDP socket can be used to send to *any* other UDP socket and it can receive from any other UDP socket. So you can use the one socket to send to multiple sockets.

Note: ***The send buffer must not exceed 2048 bytes since you cannot send more than 2048 bytes of data at a time using a UDP socket. If you need to send more than this amount do multiple sends where you divide the total bytes to be sent over multiple packets (chunks) of no more that 2048 bytes each. There is no limit on the receive buffer of a socket so it can receive more than 2048 bytes.***

4.2- Receiving data using the UDP:

When a UDP socket receives data it will be appended to the end of a receive buffer. This is performed every time a data packet is received by the socket. The buffer will continue to grow until you read it (the only limit to the size of the buffer is the memory). Once you read the buffer, the bytes in it will be returned and at the same time the buffer will be cleared.

To determine the number of bytes currently in the buffer use UDP_BuffCount(se_Name). The function returns the number of bytes currently in the buffer. Use this number to determine if there are bytes in the buffer and when to read the buffer depending on the required byte count.

To obtain a socket's receive buffer use UDP_Read(se_Name). The function will return the bytes already in the buffer and then will clear the buffer. The returned value is a string (buffer). See section 3.2 for how to extract individual data fields from the string (buffer).

4.3- Checking The Socket's Status:

The function UDP_Status(se_Name) returns a string that has information about the status of the UDP socket. The status string indicates if the socket has sent the data after a send command. It also indicates if data has been received after the socket has accepted received data. The receive status can also indicate the IP address of the originating socket. Other activities also cause the status string to change. You should check this string after utilizing the socket to ensure that the socket is actually carrying out the requested operation.

One situation where you should check the status is after a send operation. The status string will indicate if the send is successful and thus help in avoiding problems. Also the UDP_Send() function returns the number of bytes actually sent, so this can be another way of ascertaining any problems.

The UDP does not indicate if the sent buffer has not in reality been received by the destination socket. If you want to ensure this you must establish some kind of ***Acknowledgement Protocol*** between the sockets. This means that the receiver socket must send some form of ***ACK*** code back to the sender for the sender to be able to determine that the data has arrived.

The UDP does not have an extensive error correction protocol (unlike TCP). This means that a received buffer may have some errors that are not corrected by the rudimentary error correction mechanism that

the UDP utilizes (CRC). Another shortcoming with the UDP is sequencing. If you send two packets (buffers) consecutively it is possible for the latter to arrive at the destination socket before the former. This becomes more likely if the destination is not on the same LAN where one packet takes longer to arrive due to being routed over a longer link route than the other.

Therefore you need to establish a form of Error correction and Sequence verification protocols if you wish to ensure that the data has arrived uncorrupted and is reassembled in the correct sequence. One way to minimize problems is to send short packets (fewer bytes) and have a good ***ACK*** handshaking protocol.

Nonetheless, over a LAN the UDP can be quite fast and error free and it is not often necessary to establish an error protocol, however, it is advisable to establish an ***ACK*** system.

Receiver-side automatic header appending:

The data received in the receive buffer of a UDP socket will be appended to it regardless of which socket has sent the data. This means that if the socket receives data from multiple sockets there will be no way of distinguishing which data came from which socket. You can change this by using the UDP_Header(se_Name{,true|false}) function to turn on header appending to the received data.

When header appending is activated, every packet received will have a header appended to it that specifies the number of bytes and the originator IP address. Accordingly you can parse the received data and separate the received bytes into separate buffers for each sender. This can help in separating data received from multiple senders into separate buffers for actions that require different procedures according to sender. This also can help in rejecting data from invalid senders.

Sender-side manual header appending:

The receiver-side header appending can be quite powerful and useful, especially if you are communicating with systems that do not have the ability to append their own sender-side headers. Then again, a better method, which also provides for more versatility, is to have the sender append headers to the data packets. This header should have useful and application specific information. For example if multiple sockets on the same IP send data to a central socket that collects data from all the sockets, the automatic receiver-side header appending won't be able to distinguish between the sockets since they are all on the same IP and the header only holds the IP of the sender not its Port. This can be solved if the sender sockets appended a header to their data with their IP and Port and any other necessary extra bytes like the length of the packet. The length of the packet is a good thing to have especially if the packets can be of variable lengths.

4.4- Developing A UDP Program:

Sending and receiving data using UDP sockets in RobotBASIC is extremely easy. To illustrate this we shall develop 4 programs. You can also see a slightly more sophisticated demo program for UDP communications in the RobotBASIC Help file in the UDP Socket Functions section. For a more advanced demo see the programs in UDP_Demo.zip.

A very simple UDP program:

The first program is extremely simplistic as far as the user I/O is concerned and leaves much to be desired in functionality. Nevertheless, it serves to illustrate the actions of sending and receiving data through a UDP socket at its simplest without getting entangled with a GUI design or extraneous details. The program gets a key stroke from the user then sends it through a UDP socket to another. In the program listing below the remote socket is specified to be the very same socket.

Type the program and run it. If you wish you can start two instances of RobotBASIC and type the program in both instances but make sure that the variable lclPort is different in both instances. Also, make sure that the variable rmtPort on each instance is the value of the lclPort of the other instance.

If you run the two instances on different PCs then make sure that the rmtIP variable in each instance reflects the IP of the other PC. To find out the IP of a machine run this one line program on it:

```
Print TCP_LocalIP()
```

***Note:** If you run two instances on the same PC then make sure both instances are up and running before you start typing.*

```
//----------UDP_VerySimple.Bas-----------
//--make sure the other side is a different port number
lclPort = 46000
//--change rmtIP and rmtPort to send to another socket
rmtIP = TCP_LocalIP() \ rmtPort = 46000
n=udp_start("u1",lclPort)
while true
  if udp_BuffCount("u1")
     print udp_Read("u1"),
  endif
  GetKey K
  If K
     n=udp_send("u1",char(K),rmtIP,rmtPort)
     waitnokey 150
  endif
wend
```

That is all you need to achieve data sending and receiving simply between UDP sockets.

A simple UDP program:

The second program is a little better at user I/O than the first one. It is still a simple program but it illustrates how you can achieve a little better user I/O.

Just like in the previous program this one is set to send to itself. However, if you run the program in two instances of RB on the same machine or on different PCs then you need to change the variables rmtIP and rmtPort in each instance to be the values for the other instance. ***Also make sure the local Port numbers are different if running the two instances on the same PC***.

```
//----------UDP_Simple.Bas-----------
lclPort = 46000  //--make sure the other side is a different port number
//--change rmtIP and rmtPort to send to another socket
rmtIP = TCP_LocalIP() \ rmtPort = 46000
n=udp_start("u1",lclPort)
addmemo "m1",10,10,380,400
addmemo "m2",400,10,380,400 \ readonlymemo "m2"
s  = "Make sure the other side is running "
s = s+"before you start typing."
setmemotext "m1",s
SetMemoSelection "m1",1,1,length(s)
n = memochanged("m1")
focusmemo "m1"
while true
  if udp_BuffCount("u1")
     SetMemoText "m2",udp_Read("u1")
  endif
  if memochanged("m1") then n=udp_send("u1",getmemoText("m1"),rmtIP,rmtPort)
wend
```

***Note:** If you run two instances on the same PC then make sure both instances are up and running before you start typing.*

A more practical UDP program:

As a more practical use of the preceding information, we will develop an application that uses the UDP to send and receive textual and numerical data between two RobotBASIC programs that may run on the same machine or on separate machines within the LAN. ***The programs will also function between machines across the Internet, however, certain actions have to be taken with the Routers on both sides to allow for this. We shall describe these actions in Appendix B.***

Each program will have one UDP socket that will serve as the sender and receiver simultaneously. Remember that each socket is assigned a port number that must be unique. ***Consequently, if we are to be able to run the two programs on the same machine (for easier testing) then we must ensure that the two programs will not use the same port numbers for their sockets.***

The programs will intentionally be kept simple so as to facilitate understanding without too much complexity. The first program will have a user interface to obtain a text, an integer, a float and a number that is not bigger than 255 (one byte). These data will be sent to the other program that will add one to the numbers and will capitalize the text and then send it back to the sender which will display the resultant data.

We shall name the program that obtains the data from the user UDP_UserIO.Bas and the program that performs the calculations UDP_Calculator.Bas.

The UDP_UserIO.Bas program:

When the program starts running it will create an edit box so that the user can enter a Port number for the socket and press a push button to activate the socket. The Port number can be defined in the edit box only upon running the program. Once the socket has been started it will remain attached to the assigned port for the duration; the edit box will be disabled and the activation button will disappear. The user must note the Local IP and Port so as to enter them in the other program's Remote IP and Port fields.

Other edit boxes will then be created that will allow the user to specify the remote IP and Port number of the other program (UDP_Calculator.Bas) which can be on the same machine or another machine.

Another edit box will display the status of the socket as the program performs actions. The program will then create edit boxes to obtain the data from the user [text, two integers (one will be assumed to be less than 256) and a float]. There will also be a push button to initiate the data sending.

Once the Send button is pushed, the program will read the data from the edit boxes in order to create a buffer with the data in it that will be sent to the remote socket defined by the remote IP and Port address. Afterwards, the program will wait for data to be received. Once the data is received it will be read into a buffer from which the data will be extracted [byte, integer, float and text] and then assigned to the appropriate edit boxes for display. The entire action is repeatable as many times as the user wishes. See Section 3 for details on what functions to use to do the data extraction and insertion into a buffer.

Before data can be sent by pushing the Send button, the other program (UDP_Calculator.Bas) should be started and its socket activated. This is very important if you are running the two programs on the same PC (and under XP or 2000) due to the error situation discussed at the top of Section 4. But it is also important if a proper response to the sent data is to be assured, whether the two programs are running on the same machine or separate machines.

Note: The function TCP_LocalIP() is used to find out what is the local IP address of the machine the program is running on. This function can be used regardless whether you are using UDP sockets or TCP sockets. It is a general network related function despite its prefix.

```
//----------UDP_UserIO.Bas-----------
MainProgram:
  GoSub Initialization
  while true
    if LastButton() != "" then GoSub SendData
    SetEdit "Status",UDP_Status("U1")
  wend
End
//============================================================================
Initialization:
  xyText 0,10,Center(FileNAme(ProgName())," ",45),"",20,fs_Bold
  line 0,43,800,43,3
  xyText 10,50,"Local IP   = "+TCP_LocalIP(),"",14,fs_Bold
  xyText 10,75,"Local Port = ","",14,fs_Bold
  AddEdit "Local Port",145,75,50,0,47000 \ IntegerEdit "Local Port"
  AddButton "Start Socket",230,75
  AddEdit "Status",10,110,250
  repeat //wait until user pushes the button
  until LastButton() != ""
```

```
  x = UDP_Start("U1",ToNumber(GetEdit("Local Port")))  //--start the socket
  RemoveButton "Start Socket" \ EnableEdit "Local Port",false
  xyText 350,50,"Remote IP  =","",14,fs_Bold
  xyText 350,75,"Remote Port=","",14,fs_Bold
  AddEdit "Remote IP",500,50,100,0,TCP_LocalIP()
  AddEdit "Remote Port",500,75,50,0,45000 \ IntegerEdit "Remote Port"
  xytext 10,200,"   Byte:          +1 =","",14,fs_Bold
  xytext 10,230,"Integer:          +1 =","",14,fs_Bold
  xytext 10,260,"  Float:          +1 =","",14,fs_Bold
  xytext 10,290,"   Text:","",14,fs_Bold
  xytext 500,260,"   Result","",14,fs_Bold
  AddEdit "Byte",100,200 \ IntegerEdit "Byte"
  AddEdit "ByteRes",270,200 \ ReadOnlyEdit "ByteRes"
  AddEdit "Integer",100,230 \ IntegerEdit "Integer"
  AddEdit "IntegerRes",270,230 \ ReadOnlyEdit "IntegerRes"
  AddEdit "Float",100,260 \ FloatEdit "Float"
  AddEdit "FloatRes",270,260 \ ReadOnlyEdit "FloatRes"
  AddEdit "Text",100,290,300
  AddEdit "TextRes",420,290,300 \ ReadOnlyEdit "TextRes"
  AddButton "Send",420,220,100
Return
//=============================================================================
SendData:
   EnableButton "Send",false
   B = toByte(ToNumber(GetEdit("Byte"),0)) \ SetEdit "Byte",Ascii(B)
   I = ToNumber(GetEdit("Integer"),0)
   F = ToNumber(GetEdit("Float"),0)*1.0 //--multiply by 1.0 to ensure is a float
   T = GetEdit("Text")
   s = "" \ BuffPrintB s,B,I,F,T //--create the buffer
   x = UDP_Read("U1")   //--clear buffer by reading it to make sure it is empty
   RIP = GetEdit("Remote IP")
   RP = ToNumber(GetEdit("Remote Port"),1)
   x= UDP_Send("U1",s,RIP,RP)  //--send the buffer
   repeat //--wait for at least 13 bytes (1+4+8)
      SetEdit "Status",UDP_Status("U1")
   until UDP_BuffCount("U1") >=13
   delay 100 //--allow any more bytes time to arrive
   SetEdit "Status",UDP_Status("U1")
   s = UDP_Read("U1") //--read them
   SetEdit "ByteRes", BuffReadB(s,0) //--extract a byte
   SetEdit "IntegerRes",BuffReadI(s,1) //--extract integer 4 bytes
   SetEdit "FloatRes",BuffReadF(s,5) //--extract float 8 bytes
   SetEDit "TextRes",BuffRead(s,13,-1) //--extract the text the rest of it
   EnableButton "Send"
Return
//=============================================================================
```

Figure 2: User interface side of the system.

The UDP_Calculator.Bas program:

When the program starts running it will create an edit box for the user to assign a Port number for the socket and press a push button to activate the socket. The Port number can be defined in the edit box, but only upon running the program. Once the socket has been started it will remain attached to the assigned port number for the duration; the edit box will be disabled and the push button will disappear. The user must note the Local IP and Port so as to enter them in the other program's Remote IP and Port fields.

The program will create edit boxes to allow the user to specify the remote IP address and Port number where the other program (UDP_UserIO.Bas) is going to be run so as to be able to send the results back to it. It will also have another edit box to show the status of the socket for observing what is going on with the program.

Once the socket has been started the program will enter a loop waiting for data to arrive. Once data arrives, it will be read into a buffer from which the data will be extracted [the numbers and text]. The program will then perform the proper calculations and create a new buffer with the resultant data and then will send it to the remote socket as indicated by the remote IP and Port, and then loop waiting for incoming data again. See Section 3 for details on what functions to use to do the data extraction and insertion into a buffer.

```
//----------UDP_Calculator.Bas-----------
MainProgram:
  GoSub Initialization
  while true
     repeat //--wait for at least 13 bytes 1+4+8
       SetEdit "Status",UDP_Status("U1")
     until UDP_BuffCount("U1") >=13
     delay 100  //--allow time for rest of bytes to arrive
     s = UDP_REad("U1")
     B = BuffReadB(s,0) //--extract a byte
     I = BuffReadI(s,1) //--extract integer 4 bytes
     F = BuffReadF(s,5) //--extract float 8 bytes
     T = BuffRead(s,13,-1) //--extract the text the rest of it
     xyText 120,200,B,"",14,fs_Bold
     xyText 120,230,I,"",14,fs_Bold
     xyText 120,260,F,"",14,fs_Bold
     xyText 120,290,T,"",14,fs_Bold
     s = "" \ BuffPrintB s,toByte(B+1),I+1,F+1,Upper(T) //--create the buffer
     RIP = GetEdit("Remote IP") \ RP = ToNumber(GetEdit("Remote Port"),1)
     x= UDP_Send("U1",s,RIP,RP)  //--send the buffer
  wend
End
//=============================================================================
Initialization:
  xyText 0,10,Center(FileNAme(ProgName())," ",45),"",20,fs_Bold
  line 0,43,800,43,3
  xyText 10,50,"Local IP   = "+TCP_LocalIP(),"",14,fs_Bold
  xyText 10,75,"Local Port = ","",14,fs_Bold
  AddEdit "Local Port",145,75,50,0,45000 \ IntegerEdit "Local Port"
  AddEdit "Status",10,110,250
  xyText 350,50,"Remote IP  =","",14,fs_Bold
  xyText 350,75,"Remote Port=","",14,fs_Bold
  AddEdit "Remote IP",500,50,100,0,TCP_LocalIP()
```

```
  AddEdit "Remote Port",500,75,50,0,47000 \ IntegerEdit "Remote Port"
  AddButton "Start Socket",230,75
  repeat //wait until user pushes the button
  until LastButton() != ""
  x = UDP_Start("U1",ToNumber(GetEdit("Local Port")))  //--start the socket
  RemoveButton "Start Socket" \ enableEdit "Local Port",false
  xytext 120,170,"Received Data","",14,fs_Bold
  xytext 10,200,"   Byte:","",14,fs_Bold
  xytext 10,230,"Integer:","",14,fs_Bold
  xytext 10,260,"  Float:","",14,fs_Bold
  xytext 10,290,"   Text:","",14,fs_Bold
Return
//=============================================================================
```

Figure 3: The calculator side of the system.

4.5- Suggested improvements:

The two programs are pretty self explanatory and work quite adequately. However, there is one shortcoming in the system. Notice the lines in red in Figures 2 and 3. They both cause a delay of 100 ms after the loop that waits for at least 13 bytes.

The reason for the delay is that we do not know the length of a packet. We know that there are at least 13 bytes (1 + 4 + 8) which are the three numbers. But the text can be of any length. So we force a wait until at least 13 bytes come in and then delay 100 ms to ensure that the bytes for the text would also have enough time to come in before we read the buffer to get the data and then act upon it.

This delay is a guesstimate. If it is too long the program is not optimal but this is not a problem. What would happen if it is too short? The buffer would be read before all the text data has had time to arrive and we would be getting the incomplete data.

There are many ways we can get around this problem. The best solution is through the use of a header. The data packet should always have an initial field that indicates the length of the data in it. We can then wait for 4 bytes. Read the buffer (it may by then have more than 4 bytes). We extract the number of bytes to be expected (say X) from the header. We then wait for (X-length of buffer already read) more bytes to arrive. Once they arrive we append them to the previously read data and then start manipulating the data.

The above assures optimal waiting. There are many other ways to achieve a similar action. For example the sender can send the number of bytes to be expected and then wait for acknowledgement before it starts sending the actual data. The point is that it is up to you how to implement the Hand-Shaking protocol to achieve synchronized orderly and error free communications. RobotBASIC provides the link for the data, while the data content and synchronization are up to you.

Both programs use a waiting loop that waits for data to arrive at the receive buffer. This is not much of a problem in this application since the programs do not need to do much else. But, if the programs had to do other tasks while waiting for the data to arrive, then it becomes necessary to use ***EVENT*** handling.

RobotBASIC can be instructed to ***interrupt*** what it is currently doing and jump to a special routine whenever any bytes arrive at the receive buffer of ***any*** active UDP socket. In the routine (***event handler***)

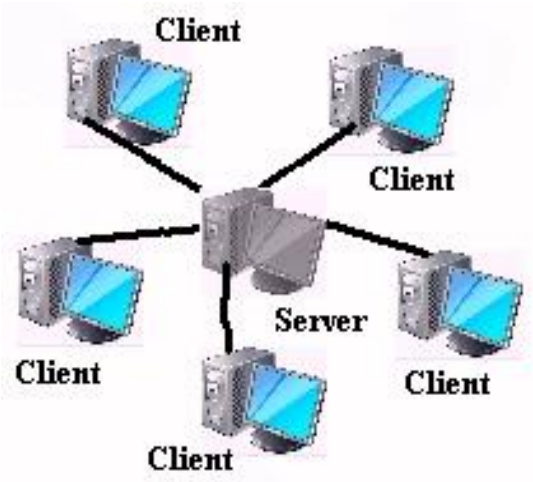
you would determine which active sockets have data in them and read these data and transfer them to a separate accumulator buffer for each. The handler also can initiate actions depending on which socket has received the required amount of data and so forth. Once the handler routine finishes, the program will return to doing what it was doing before the interruption.

This interrupt mechanism allows the program to do actions other than just sit in a loop waiting for data to arrive. The mechanism is activated with the statement OnUDP; read about it in the RobotBASIC Help file. Also for examples of how to use the mechanism see the demo program in the UDP section of the RobotBASIC Help file as well as the following zip file UDP_Demo.zip

5- Utilizing The TCP System:

A RobotBASIC program has access to ***one TCP server socket*** and ***one TCP client socket***.

Figure 4: Client Server Conceptual Layout



The TCP is a connection oriented Client-Server protocol. A server socket can accept multiple clients while a client socket can only be connected to one server at a time. You can disconnect a client from a server and then connect to another server; however, it can only be connected to ***one server at a time***.

The server socket can accept multiple clients and will be able to receive data from all the clients and can send data to all the clients.

Once a client has connected to a server it becomes as if there is a direct two-way com link between them. Figure 4 shows the logical conceptual layout once connections between the server and various clients have been established. ***Notice that the server and any of the clients can be on distinct LANs or the same LAN***.

A server socket must be assigned a unique and not currently in use Port number. When created, the socket will *automatically* use the IP address of the machine it is created on, but, you must assign it a particular port number. The client socket does not need to be assigned a Port number. It will automatically use any available port and will use the IP of the machine it is running on.

Note: ***You must ensure that the port assigned to a TCP Server socket is unique and is not being used by another active socket (UDP or TCP) in the same program or other programs on the same machine. BE SURE OF THIS.***

Since there is only one client socket and one server socket there is no need to name them when created or when using functions related to them

To create and activate a TCP Server socket use TCPS_Serve({ne_Port}). Notice that the port number is optional. If you do not specify a port number it will be assumed to be 50000 (see Appendix A for a list

of preferred port numbers). A server socket can be closed any time and reactivated to use a different Port number. To deactivate a server socket use TCPS_Close(). No parameter is required. ***It is important that you monitor the status string of the server socket to know if it has been successfully activated. See section 5.3.***

Use TCPC_Connect(se_ServerIPaddress{,ne_ServerPort}) to create and activate a TCP Client socket and at the same time connect to a server socket. Notice that the server port number is optional. If you do not specify a port number it will be assumed to be 50000. This function will try to connect to the server socket as specified by the IP and Port number combination. If the server is available and running the connection will be established. A client socket can be closed any time and disconnected from the server and then reactivated to connect to a different server. To deactivate a client socket use TCPC_Close(). No parameter is required. ***It is important that you monitor the status string of the client socket to know if it has been successfully activated. See section 5.3.***

Notice that the client socket itself is not assigned an IP or Port number. As mentioned earlier the client socket will automatically use any available free port and the IP of the machine it is running on.

The functions discussed in the following sections enable the writing of a program to send and/or receive data packets over the TCP. A server/client socket in a RobotBASIC program can communicate over the LAN or Internet with a client/server socket that does not have to be another RB program. So long as the data format in the send/receive buffer is in accordance with a format that both sockets are in agreement upon, the other socket can be:

a- The same PC running another RB program.
b- The same PC running a non-RB program.
c- Another PC with another RB program.
d- Another computer (not just a PC) running a non-RB program.
e- A device (e.g. robot or microcontroller) that can use the TCP.

5.1- Sending data using the TCP:

Once a client has established a connection with a server it can send packets to it and so can the server send to the client. Use TCPC_Send(se_Data) to send data packets from the client to the server. Notice that you do not need to specify an IP or port the only parameter is the data buffer (string). This is of course because there is already an established link and the client can only send and receive data over this link. A client cannot send to any other server socket without disconnecting from the currently connected server and then connecting to the other server socket.

A server can send data to ***all*** the clients that are connected to it. The server socket in RobotBASIC cannot send to a particular client only. It can send (broadcast) to all the clients connected to it. If you wish to make the data significant to only one client you need to establish a header mechanism as discussed in section 5.3. Use TCPC_Send(se_Data) to send data from the server socket to ***all*** the client sockets currently connected to the server. Notice that there is no IP or port parameter. This is because the connection link is already established and these values are already known from the link status.

In both the above functions the parameter se_Data is the data buffer to be sent. See Section 3 for how to create and manipulate the buffer.

Note: ***The send buffer in both the server and client sockets is limitless. However, from personal experience an optimal size exists. It is best if you limit the send packet to 2^{18} (262144=0x040000) bytes. This size seems to be able to reach its destination faster than smaller or bigger buffer sizes.***

4.2- Receiving data using the TCP:

When either the TCP Server socket or the TCP Client socket receives data it will be appended to the end of a receive buffer. This is performed every time a data packet is received by the socket. The buffer will continue to grow until you read it (the only limit to the size of the buffer is the memory). Once you read the buffer, the data in it will be returned and at the same time the buffer will be cleared.

To determine the number of bytes currently in the buffer use TCPC_BuffCount() for the client socket and TCPS_BuffCount() for the server socket. The function returns the number of bytes currently in the buffer. Use this number to determine if there is data in the buffer and when to read the buffer depending on the required byte count.

The client socket can only receive packets from the server it is connected to and thus there is no problem with knowing where the data came from. But, the server socket can receive data at any time from any of the clients it is connected to. So there would be a problem of knowing which data came from which client. See section 5.3 for solutions to this problem. Of course if you only have one client then there is no problem.

To obtain the sockets' receive buffers use TCPC_Read() for the client socket and TCPS_Read() for the server socket. The function will return the bytes already in the buffer and then will clear the buffer. The returned value is a string (buffer). See section 3.2 for how to extract individual data fields from the string (buffer).

5.3- Checking The Sockets' Status:

The function TCPC_Status() for the client and TCPS_Status() for the server socket returns a string that has information about the status of the socket. The status string indicates if the socket has sent the data after a send command. It also indicates if data has been received after the socket has accepted received data.

For the client socket the status string can also indicate if the client has connected to a server and which one. It is also possible to read the status to find when and if the server has disconnected from the client. Likewise, for the server socket there are similar status strings. Read the RobotBASIC Help file for a list of these status strings and their significance.

The receive status on the server has information about the IP and Port number of the client socket that sent the data. Other activities also cause the status string to change. You should check this string after utilizing the socket to ensure that the socket is actually carrying out the requested operation.

One situation where you should check the status is after a send operation. The status string will indicate if the send is successful and thus help in avoiding problems. Another situation where you may want to monitor the status string closely is while connecting a client to a server. The status string will indicate the progress of the operation and if successful or not.

The TCP implements a very rigorous error correction and sequence assurance mechanisms. Once a link between the client and server is established you can almost always be assured that a sequence of data packets sent in succession will arrive to the other side in the correct order and will be error free and will definitely arrive. If there is any problem the status string will indicate so.

However despite all this assurance you may want to establish some kind of ***Acknowledgement Protocol*** between the sockets. This means that the receiver socket must send some form of ***ACK*** code back to the sender for the sender to be able to determine that the data has arrived. This is not strictly necessary but it may be useful for synchronization purposes in certain situations.

To see how this is achieved see the programs in the zip files mentioned at the beginning of this article.

Server-side automatic header appending:

The data received in the receive buffer of a server socket will be appended to it regardless of which client socket has sent the data. This means that if the socket receives data from multiple clients there will be no way of distinguishing which data came from which client. You can change this by using the TCPS_Header({,true|false}) function to turn on header appending to the received data.

When header appending is activated, every packet received will have a header appended to it that specifies the number of bytes and the originator IP address and Port number of the sending client. This way you can parse the received data and separate the received bytes into separate buffers for each client. This can help in being able to receive data from multiple clients and to put the data from each in separate storage areas for actions that require different procedures according to the client. This also can help in verifying if a client is a valid one.

Note: There is no such mechanism for the client side as there is no need for it.

Sender-side manual header appending:

The server-side header appending can be quite powerful and useful, especially if you are communicating with systems that do not have the ability to append their own client-side headers. However, a better method which also provides for more versatility is to have the sender (server or client) append headers to the data packets it sends. This header should have useful and application specific information. For example a server acting as a central controller for multiple robots connected to it as clients (e.g. soccer team) can tell a particular client to do an action by appending a header to each message it sends that specifies the target client. So despite all the clients receiving the same message only one acts upon it since the header indicates which client should act.

It is important to realize that the TCP functionality in RB provides a bidirectional data link between the client and the server. It is up to you what the data content is and what methods you use for ensuring synchronization and data integrity as well as what the data content signifies.

5.4- Developing a TCP program:

Sending and receiving data using the TCP Server and TCP Client sockets in RobotBASIC is relatively easy. To illustrate this we shall develop 4 simple programs. You can also see a slightly more sophisticated demo program for TCP communications in the RobotBASIC Help file in the TCP Sockets Functions section. For more advanced demos TCP_Demo.zip and TCP_Robot.zip.

A very simple TCP program:

The first program is extremely simplistic as far as the user I/O and error situations detection is concerned and leaves much to be desired in functionality. Nevertheless, it serves to illustrate the actions of sending and receiving data through the TCP server and client sockets at its simplest without getting entangled with a GUI design or extraneous details.

The program gets a key stroke from the user and then sends it through the TCP Server (Client) socket to the TCP Client (Server) socket.

Note: ***In the program listing below the program acts, both as the server and client. However, you can run two instances of the program either on the same PC or on different PCs. You must decide which instance is going to act as a Server and which will act as a Client. In the Client instance change the variables rmtIP and rmtPort to be the values of the Server's instance, and make sure the variable IsServer is set to false. Additionally, make sure that in the server instance the variable IsClient is set to false. Remember, now you will only see text on the server side when you type on the client side and vice versa. Also make sure you run the server instance BEFORE the client instance.***

```
//----------TCP_VerySimple.Bas-----------
lclPort = 50000  //--This is only important for the server side
//--change rmtIP and rmtPort to to the server's IP and Port
rmtIP = TCP_LocalIP() \ rmtPort = 50000
IsServer = true \ IsClient = true
if IsServer then n=TCPS_Serve(lclPort)
if IsClient then n=TCPC_Connect(rmtIP,rmtPort)
while true
 if IsServer
    if TCPS_BuffCount()
      print TCPS_Read(),
    endif
    GetKey K
    If K
      n=TCPS_send(char(K))
      waitnokey
    endif
 endif
 if IsClient
    if TCPC_BuffCount()
      print TCPC_Read(),
    endif
    GetKey K
    If K
      n=TCPC_send(char(K))
      waitnokey
```

```
    endif
 endif
wend
```

That is all you need to achieve data sending and receiving simply between TCP client and server sockets.

A simple TCP program:

The second program is a little better at user I/O than the first one. It is still a simple program but it illustrates how you can achieve a little better user I/O.

Note: ***In the program listing below the program acts, both as the server and client. However, you can run two instances of the program either on the same PC or on different PCs. You must decide which instance is going to act as a Server and which will act as a Client. In the Client instance change the variables rmtIP and rmtPort to be the values of the Server's instance, and make sure the variable IsServer is set to false. Additionally, make sure that in the server instance the variable IsClient is set to false. Remember, now you will only see text on the server side when you type on the client side and vice versa. Also make sure you run the server instance BEFORE the client instance.***

```
//----------TCP_Simple.Bas-----------
lclPort = 50000  //--this is important for the Server side
//--change rmtIP and rmtPort to send to another socket
rmtIP = TCP_LocalIP() \ rmtPort = 50000
IsServer = true\ IsClient = true
addmemo "m1",10,10,380,400
addmemo "m2",400,10,380,400 \ readonlymemo "m2"
s = "Start typing"
ts  = "Make sure the "
ts2 = " is running before you start typing."
if IsServer & !IsClient then s = ts+"Client"+ts2
if !IsServer & IsClient then s = ts+"Server"+ts2
setmemotext "m1",s
SetMemoSelection "m1",1,1,length(s)
n = memochanged("m1")
focusmemo "m1"
if IsServer then n=TCPS_Serve(lclPort)
if IsClient then n=TCPC_Connect(rmtIP,rmtPort)
while true
  if IsServer
     if TCPS_BuffCount()
        SetMemoText "m2",TCPS_Read()
     endif
     if memochanged("m1") then n=TCPS_send(getmemoText("m1"))
  endif
  if IsClient
     if TCPC BuffCount()
        SetMemoText "m2",TCPC_Read()
     endif
     if memochanged("m1") then n=TCPC_send(getmemoText("m1"))
  endif
wend
```

A more practical TCP program:

As a more practical use of the preceding information, we will develop an application that uses the TCP to send and receive textual and numerical data between two RobotBASIC programs that may run on the same machine or on separate machines within the LAN. ***The programs will also function between machines across the Internet, however, certain actions have to be taken with the Routers on the server side to allow for this. We shall describe these actions in Appendix B.***

One program will be the server and the other will be the client. The programs will intentionally be kept simple so as to facilitate understanding without too much complexity. The client program will have a user interface to obtain a text, an integer, a float and number that is not bigger than 255 (one byte). These data will be sent to the server program that will add one to the numbers and will capitalize the text and then send it back to the client which will display the resultant data.

We shall name the client program which also obtains the data from the user TCPC_UserIO.Bas and the server program that also performs the calculations TCPS_Calculator.Bas.

The TCPC_UserIO.Bas program:

When the program starts running it will create edit boxes so that the user can enter the server's IP address and Port number (these should be obtained from the server's screen). A button will be shown that allows the user to activate the socket and connect to the server indicated by the Remote IP and Remote Port numbers. Before connecting to the server by pushing the Connect button, the server program (TCPS_Calculator.Bas) should be started and its socket activated. This is important if a connection is to be possible.

Another edit box will display the status of the socket as the program performs actions. The program will then create edit boxes to obtain the data from the user [text, two integers (one will be assumed to be less than 256) and a float]. There will also be a push button to initiate the data sending.

Once the Send button is pushed, the program will read the data from the edit boxes in order to create a buffer with the data in it which will then be sent to the server. The program will then wait for data to be received. Once the data is received it will be read into a buffer from which the data will be extracted [byte, integer, float and text] and then assigned to the appropriate edit boxes for display. The entire action is repeatable as many times as the user wishes. See Section 3 for details on what functions to use to do the data extraction and insertion into a buffer.

Note: The function TCP_LocalIP() is used to find out what is the local IP address of the machine the program is running on. This function can be used regardless whether you are using UDP sockets or TCP sockets. It is a general network related function despite its prefix.

```
//----------TCPC_UserIO.Bas-----------
MainProgram:
  GoSub Initialization
  while true
    if LastButton() != "" then GoSub SendData
    SetEdit "Status",TCPC_Status()
  wend
End
//=============================================================================
```

```
Initialization:
  xyText 0,10,Center(FileNAme(ProgName())," ",45),"",20,fs_Bold
  line 0,43,800,43,3
  xyText 10,50,"Local IP   = "+TCP_LocalIP(),"",14,fs_Bold
  AddEdit "Status",10,110,250
  xyText 350,50,"Remote IP  =","",14,fs_Bold
  xyText 350,75,"Remote Port=","",14,fs_Bold
  AddEdit "Remote IP",500,50,100,0,TCP_LocalIP()
  AddEdit "Remote Port",500,75,50,0,47000 \ IntegerEdit "Remote Port"
  AddButton "Connect",640,55,100
  repeat //wait until user pushes the button
  until LastButton() != ""
  RIP = GetEdit("Remote IP") \ RP = ToNumber(GetEdit("Remote Port"))
  x = TCPC_Connect(RIP,RP)  //--connect to server
  RemoveButton "Connect"
  xytext 10,200,"   Byte:          +1 =","",14,fs_Bold
  xytext 10,230,"Integer:          +1 =","",14,fs_Bold
  xytext 10,260,"  Float:          +1 =","",14,fs_Bold
  xytext 10,290,"   Text:","",14,fs_Bold
  xytext 500,260,"   Result","",14,fs_Bold
  AddEdit "Byte",100,200 \ IntegerEdit "Byte"
  AddEdit "ByteRes",270,200 \ ReadOnlyEdit "ByteRes"
  AddEdit "Integer",100,230 \ IntegerEdit "Integer"
  AddEdit "IntegerRes",270,230 \ ReadOnlyEdit "IntegerRes"
  AddEdit "Float",100,260 \ FloatEdit "Float"
  AddEdit "FloatRes",270,260 \ ReadOnlyEdit "FloatRes"
  AddEdit "Text",100,290,300
  AddEdit "TextRes",420,290,300 \ ReadOnlyEdit "TextRes"
  AddButton "Send",420,220,100
Return
//=============================================================================
SendData:
   EnableButton "Send",false
   B = toByte(ToNumber(GetEdit("Byte"),0)) \ SetEdit "Byte",Ascii(B)
   I = ToNumber(GetEdit("Integer"),0)
   F = ToNumber(GetEdit("Float"),0)*1.0 //--multiply by 1.0 to ensure is a float
   T = GetEdit("Text")
   s = "" \ BuffPrintB s,B,I,F,T //--create the buffer
   x = TCPC_Read()  //--clear buffer by reading it to make sure it is empty
   x= TCPC_Send(s)  //--send the buffer
   repeat           //--wait for at least 13 bytes (1+4+8)
      SetEdit "Status",UDP_Status("U1")
   until TCPC_BuffCount() >=13
   delay 100 //--allow any more bytes time to arrive
   SetEdit "Status",TCPC_Status()
   s = TCPC_Read() //--read them
   SetEdit "ByteRes", BuffReadB(s,0)   //--extract a byte
   SetEdit "IntegerRes",BuffReadI(s,1) //--extract integer 4 bytes
   SetEdit "FloatRes",BuffReadF(s,5)   //--extract float 8 bytes
   SetEDit "TextRes",BuffRead(s,13,-1) //--extract the text the rest of it
   EnableButton "Send"
Return
//=============================================================================
```

Figure 5: User interface side of the system.

The TCPS_Calculator.Bas program:

When the program starts running it will create an edit box for the user to assign a Port number for the server socket and press a push button to activate the socket. The Port number of the socket can be defined in the edit box, but only upon running the program. Once the socket has been started it will remain attached to the assigned port number as long as the program is running; the edit box will be disabled and the push button will disappear. The user must note the Local IP and Port so as to enter them in the client program's Remote IP and Port fields.

The program will also create an edit box to show the status of the socket for observing what is going on with the program. Also any received data will be displayed on the screen.

Once the socket has been started the program will enter a loop waiting for data to arrive. A client can connect to the server then start sending data to it. Once data arrives, it will be read into a buffer from which the data will be extracted [the numbers and text]. The program will then perform the proper calculations and create a new buffer with the resultant data and then will send it to the clients connected to it, and then loop waiting for incoming data again. See Section 3 for details on what functions to use to do the data extraction and insertion into a buffer.

```
//----------TCPS_Calculator.Bas-----------
MainProgram:
  GoSub Initialization
  while true
     repeat //--wait for at least 13 bytes 1+4+8
       SetEdit "Status",TCPS_Status()
     until TCPS_BuffCount() >=13
     delay 100  //--allow time for rest of bytes to arrive
     s = TCPS_Read()
     B = BuffReadB(s,0)    //--extract a byte
     I = BuffReadI(s,1)    //--extract integer 4 bytes
     F = BuffReadF(s,5)    //--extract float 8 bytes
     T = BuffRead(s,13,-1) //--extract the text the rest of it
     xyText 120,200,B,"",14,fs_Bold
     xyText 120,230,I,"",14,fs_Bold
     xyText 120,260,F,"",14,fs_Bold
     xyText 120,290,T,"",14,fs_Bold
     s = "" \ BuffPrintB s,toByte(B+1),I+1,F+1,Upper(T) //--create the buffer
     x= TCPS_Send(s)  //--send the buffer
  wend
End
//=============================================================================

Initialization:
  xyText 0,10,Center(FileNAme(ProgName())," ",45),"",20,fs_Bold
  line 0,43,800,43,3
  xyText 10,50,"Local IP   = "+TCP_LocalIP(),"",14,fs_Bold
  xyText 10,75,"Local Port = ","",14,fs_Bold
  AddEdit "Local Port",145,75,50,0,47000 \ IntegerEdit "Local Port"
  AddEdit "Status",10,110,250
  AddButton "Serve",230,75
  repeat //wait until user pushes the button
  until LastButton() != ""
  x = TCPS_Serve(ToNumber(GetEdit("Local Port")))  //--start the server
```

```
  RemoveButton "Serve" \ enableEdit "Local Port",false
  xytext 120,170,"Received Data","",14,fs_Bold
  xytext 10,200,"   Byte:","",14,fs_Bold
  xytext 10,230,"Integer:","",14,fs_Bold
  xytext 10,260,"  Float:","",14,fs_Bold
  xytext 10,290,"   Text:","",14,fs_Bold
Return
//=============================================================================
```

Figure 6: The calculator side of the system.

5.5- Suggested improvements:

The two programs are pretty self explanatory and work quite adequately. However, there is one shortcoming in the system. Notice the lines in red in Figures 5 and 6. They both cause a delay of 100 ms after the loop that waits for at least 13 bytes.

The reason for the delay is that we do not know the length of a packet. We know that here are at least 13 bytes (1 + 4 + 8) which are the three numbers. But the text can be of any length. So we force a wait until at least 13 bytes come in and then delay 100 ms to ensure that the bytes for the text would also have enough time to come in before we read the buffer to get the data and then act upon it.

This delay is a guesstimate. If it is too long the program is not optimal but this is not a problem. What would happen if it is too short? The buffer would be read before all the text data has had time to arrive and we would be getting the incomplete data.

There are many ways we can get around this problem. The best solution is through the use of a header. The data packet should always have an initial field that indicates the length of the data in it. We can then wait for 4 bytes. Read the buffer (it may by then have more than 4 bytes). We extract the number of bytes to be expected (say X) from the header. We then wait for (X-length of buffer already read) more bytes to arrive. Once they arrive we append them to the previously read data and then start manipulating the data.

The above assures optimal waiting. There are many other ways to achieve a similar action. For example the sender can send the number of bytes to be expected and then wait for acknowledgement before it starts sending the actual data. The point is that it is up to you how to implement the Hand-Shaking protocol to achieve synchronized orderly and error free communications. RobotBASIC provides the link for the data, while the data content and synchronization are up to you.

RobotBASIC provides the link for the data, but the data content and the data sequencing is all up to you.

Both programs use a waiting loop that waits for data to arrive at the receive buffer. This is not much of a problem in this application since the programs do not need to do much else. But, if the programs had to do other tasks while waiting for the data to arrive, then it becomes necessary to use ***EVENT*** handling.

RobotBASIC can be instructed to ***interrupt*** what it is currently doing and jump to a special routine whenever any bytes arrive at the receive buffer of ***any*** active UDP socket. In the routine (***event handler***) you would determine which active sockets have data in them and read these data and transfer them to a separate accumulator buffer for each. The handler also can initiate actions depending on which socket

has received the required amount of data and so forth. Once the handler routine finishes, the program will return to doing what it was doing before the interruption.

This interrupt mechanism allows the program to do actions other than just sit in a loop waiting for data to arrive. The mechanism is activated with the statements OnTCPS and OnTCPC; read about them in the help file. For examples of how to use the mechanism see the programs in the zip files mentioned at the beginning of this article as well as the demo program in the TCP section of the help file.

Appendix A – Selecting A Port Number:

When activating a TCP Server or when creating and starting a UDP socket the IP address for the socket is determined by the machine it is running on. But the Port number has to be assigned.

Either you as the programmer can hard code the Port number in your program, or you can provide a means for the user of your program to select a port number. Regardless of which option you opt for a Port number has to be selected and assigned to the socket.

How would you select a number? Well the easy answer from experience is that it is safer mostly to choose numbers greater than 40000 but any port that your system is not currently using will suffice.

The longer answer is that there are many ports that are universally agreed upon as standard ports for usage with ubiquitous programs. For example FTP is usually assigned to port 21, SMTP to port 25, HTML to port 80 and so forth. There are ports exclusively for UDP and others for TCP. Many other applications assign port numbers for UDP usage and TCP usage.

So try to avoid these ports. To help you in deciding what ports are likely to be used on your system and therefore to avoid, see these two web pages in order of preference:

http://www.iss.net/security_center/advice/Exploits/Ports/default.htm
http://www.portforward.com/cports.htm

It is generally true that ports above 40000 are safe to use. ***Do not ever use ports that are assigned for HTML, SMTP, FTP and other universal Internet services.***

Appendix B –Allowing Internet Throughput:

All the information given in this article is equally applicable whether you are running your application within the LAN or across the Internet. As long as you have the right IP address and the right Port number for the target UDP socket you can send data to it, and if you have the right IP address and Port number of a server socket a client socket can connect to it.

With UDP sockets you need the IP and Port number for both ends to allow bidirectional communications. With TCP you need to know the IP and Port of the server, the client can just connect and the link is then automatically bidirectional without ever knowing the IP address or Port number of the client. However you may want to know the IP addresses and Port numbers of valid clients so as to be able to restrict access to just valid clients.

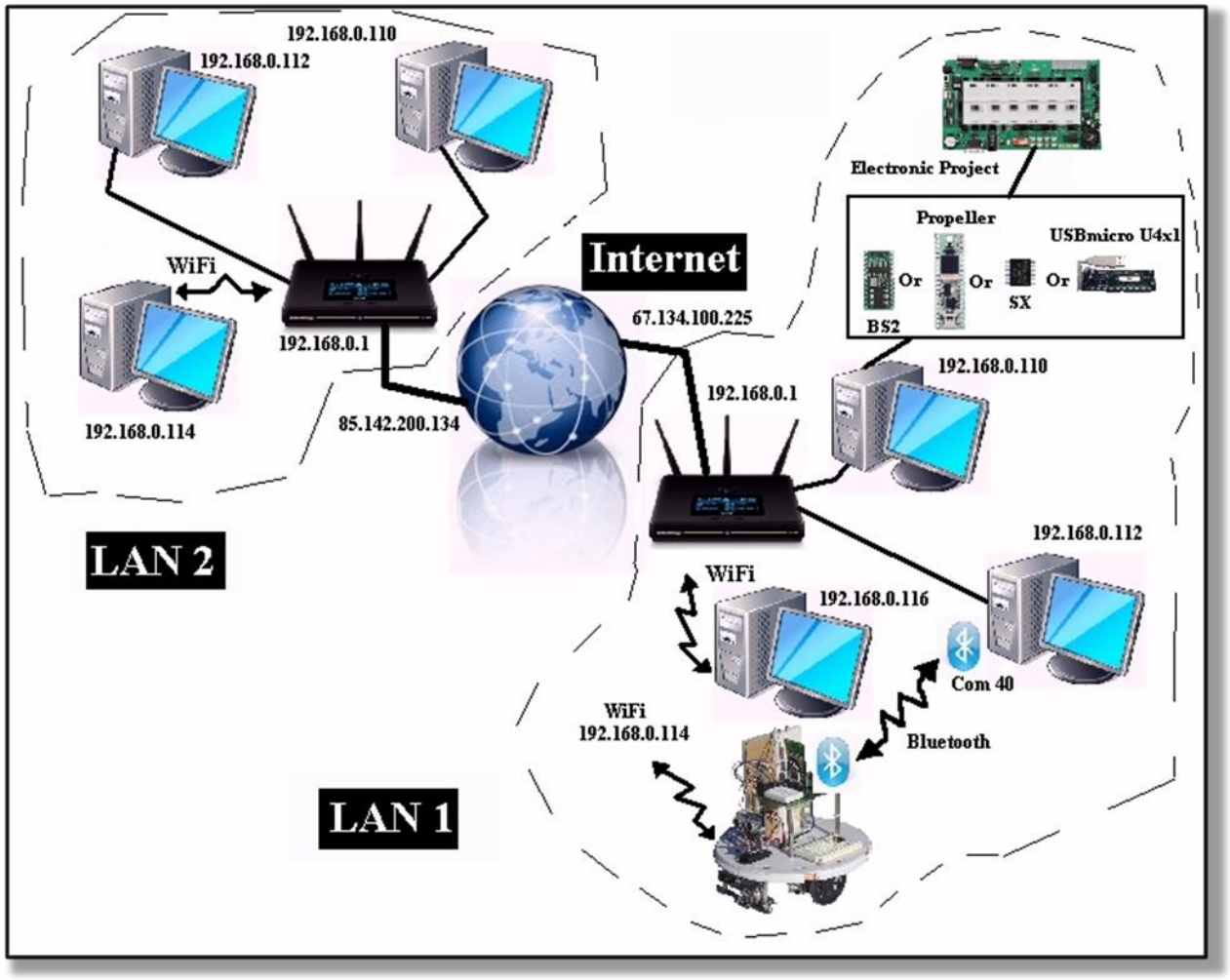


Figure 7: A typical network arrangement.

When it comes to communicating to a machine outside your LAN you are going to have to deal with the Fire Wall and NAT aspects of the system. An IP address within the LAN is not a valid address for usage by a machine outside the LAN. This means that even though your machines local IP address is known it cannot be used as a remote IP by another PC outside your LAN. It is a usable and legal IP address for machines within the LAN but not across the internet.

If you look at Figure 7 (a repeat of Figure 1) you will notice that the routers have two distinct addresses each. The addresses that start with 192.168 are what is called the inside the LAN address. This is the address you can connect to your router by a machine inside the LAN. Also notice that all the machines inside the LANs have addresses that start with 192.168 and that LAN1 machines have the same addresses as LAN2 machines. Normally it is not legal for two machines to have the same address on the internet. Each machine that is visible to the global internet must have a unique IP address, but since the machines inside the LAN are not directly visible to the internet, it is possible for this address duplication to occur.

The IP standard has set aside an address space that always starts with 192.168 which is designed to be a LAN group of addresses and that the global internet will not use or assign to actual real machines. This is what the NAT does. It translates LAN addresses so as to appear as if they are valid addresses on the Internet. How this is achieved does not concern us. But what concerns us is that we need to achieve a method for letting a machine on LAN2 to be able to communicate with a machine on LAN1.

As far as the internet is concerned your entire LAN is one machine given one address that is usually assigned by the ISP. In Figure 7 you can see that LAN1 is given the address 67.134.100.225. This is the address of your machine as far as any machine outside the internet is concerned. This IP is assigned by the ISP usually on a lease basis. That means that it is not a permanent address and if your LAN ever disconnects from the ISP and then reconnects you will end up being assigned another IP that, more often than not, is different. It is possible to have a permanent (static) address given to you but this is normally only possible for bigger companies that have big LANs.

The final outcome is that your LAN as far as the internet is concerned is one machine with one IP address. In the setup in Figure 7 the router is what the Internet sees and nothing else. So any machine trying to log on to your system can only go through the router.

Note: ***If your network setup differs from Figure 7 you will need to consult a network administrator on how to setup your system to allow access to a machine within the LAN from a machine outside the LAN. This discussion may help you get an idea of what is required in general but will not be applicable due to the difference.***

You can use the following web site to ascertain what your LAN's IP address is. But remember this can change if you ever disconnect your system from the ISP. http://www.whatismyip.com/

When you log on the web site you will see the screen below (other information on the site is also of interest). Another way you can get your Global (External) IP is by logging on to your router, which is also an operation that will be necessary for accomplishing the task of setting up your LAN so that an outside machine can connect with a machine inside the LAN. This action will be described shortly.

So now we know what IP address to give to the external machine to use in the UDP or TCP functions that require them (as described in sections 4 and 5). We also have decided on a port to use as described by Appendix A. So we have all we need. A machine say in LAN2 can now communicate with a machine in LAN1 using the IP address 67.134.100.225 and the chosen port, let's say 50000.

Well, not quite! There is an additional problem we need to resolve. The IP address is not in fact the address of any PC inside LAN1; it is really the address of the router. So how do we get the data packets that will reach the router to go to the PC that in fact is running the destination socket?

The answer to this is something called Port Forwarding. Simply, port forwarding is a way to tell the router that whenever it sees a data packet coming to it on a particular port to forward that packet to a PC inside the LAN. This PC will see the data packet and it will appear to it as if it came directly from the machine on LAN2.

To summarize the steps so far are:

1- Decide what type of socket you are going to use, TCP Server or UDP (e.g. UDP)
2- Decide what Port your socket is going to be associated with (e,g. 45000)
3- Find out what is the internal LAN address of the machine hosting the socket (e.g 192.168.0.110). You can use the TCP_LocalIP() function to do this.
4- Log on to your router and set it up to do port forwarding of Port 45000 for UDP traffic to machine 192.168.0.110.
5- Reboot your router. This is needed for the changes to take effect. It will also change you external IP address so there is no need to have found it before this step.
6- Find out what your External IP address is. You can do this by going to the web site above or by logging onto the router again and reading the information. Say 87.134.100.225
7- Give this external IP address and the port number (45000) to the machine on LAN2 that is going to send data to your UDP socket.
8- Start your program on the assigned machine and wait for data to come in.
9- Start the machine inside LAN2 and start a UDP socket and tell it to send data to the remote IP 87.134.100.225 and remote Port 45000.
10- That is all.

Note: ***For TCP you only need to set up port forwarding for the server side. A client, once connected to a server, will establish a bidirectional link automatically and there is no need to set up port forwarding on the client side.***

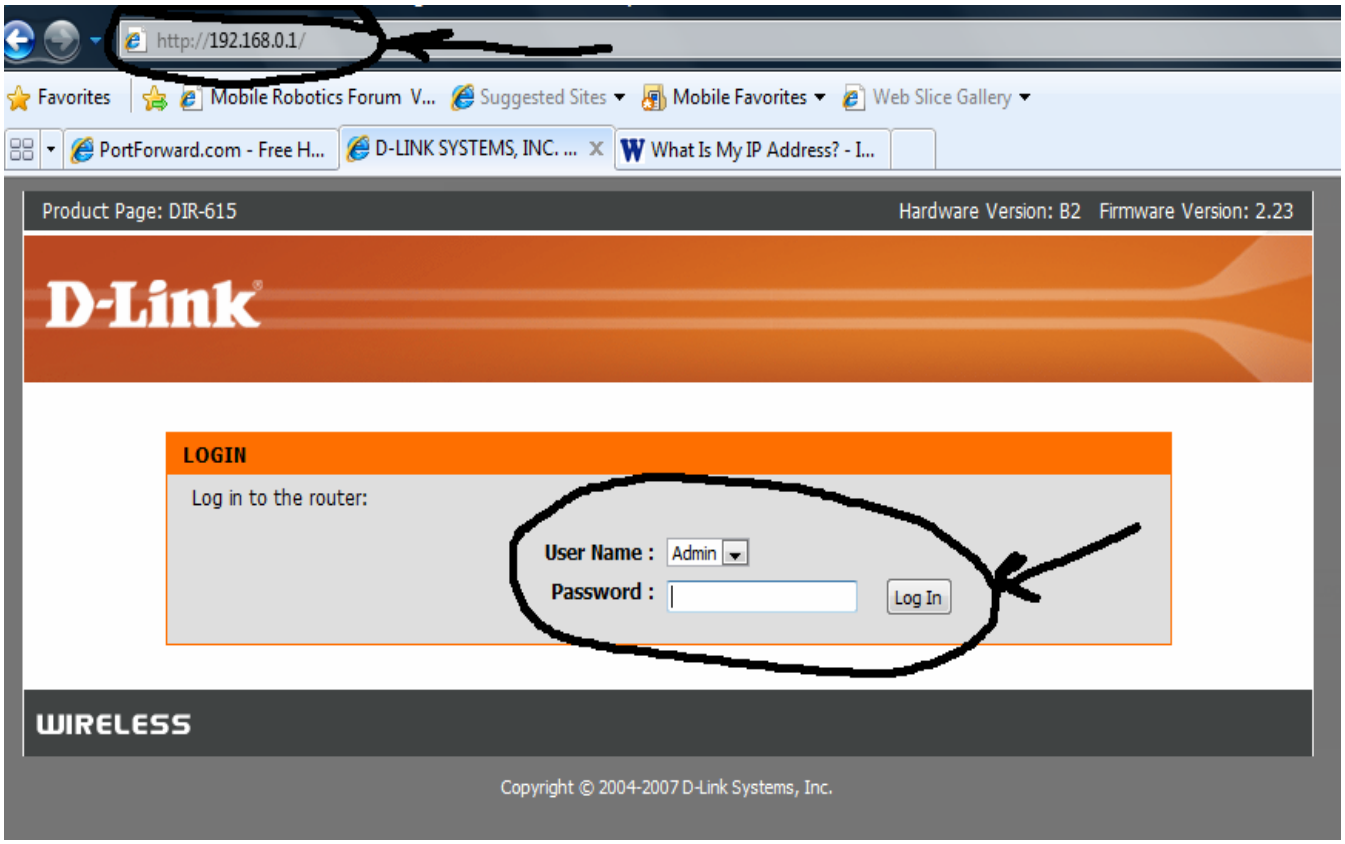
Note: ***For UDP you need to setup port forwarding on each side that is going to receive data.***

Configuring The Router For Port Forwarding:

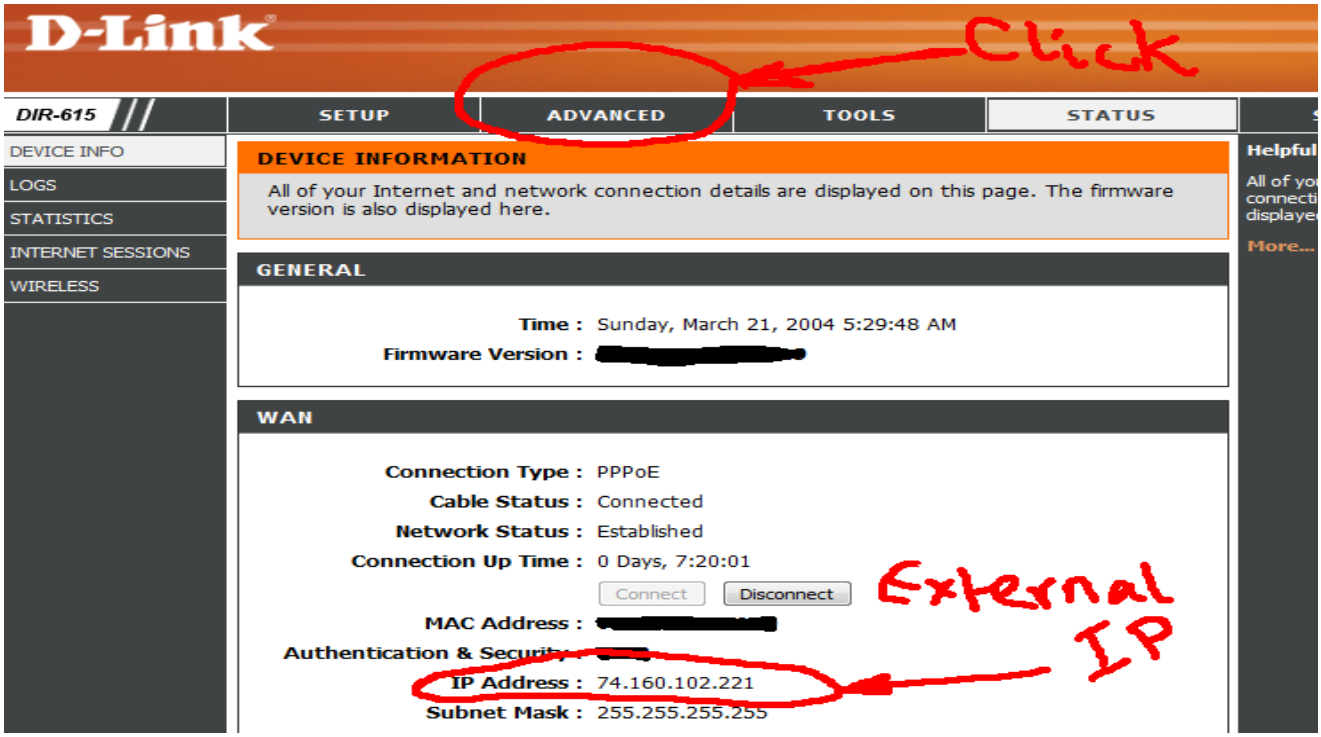
The following procedure will show how to set up port forwarding for a very specific router (the one I have). Your router will have a different layout. However, the procedure is generally similar and other than different menus or screen layouts the principles and most of the terminologies are the same on most routers. If you need help with your router ask technical support from your vendor or visit this website: http://portforward.com/.

To access your router use your Internet Browser and type in the URL space the IP address of your router. Most of the time this is 192.168.0.1 but it may be 0.0 or 1.0 or 0.2 etc. Consult the router's manual or help desk.

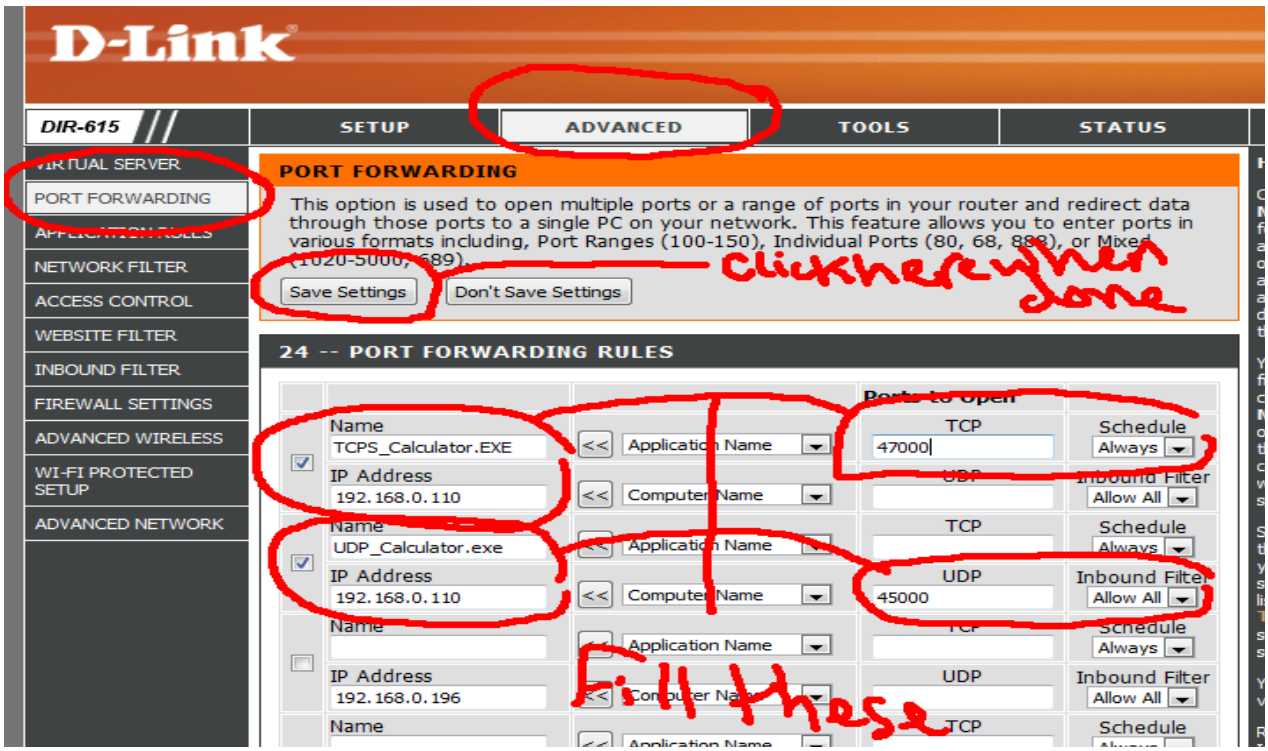
You will then see a logon screen on your browser. You will have to login using a password. If you do not know it, ask your technical support.



Next you will see the screen shot below. Notice the External IP address information. But this is going to change later so ignore it for now. Click on the advanced menu option.



When you have navigated to the screen below, fill in the indicated areas:



After you finish click 'Save Settings' which will cause the router to reboot and will also cause it be assigned a new external IP which you can find out by going back the 1st screen (you have to log in again) or by going to the web site mentioned earlier.

That is all! You now have Port Forwarding. As far as the internet is concerned the machine on LAN1 with the IP address 192.168.0.110 will appear to the outside world as IP 87.134.100.225 for any TCP traffic to Port 47000 and UDP traffic to Port 45000.

In the case of the UDP example we developed in section 4 you will also have to do the same for the machine on LAN2. In the case of TCP you do not have to do port forwarding on the other machine since it will be the client and thus will have automatic two way traffic.

Note: The above is a very specific example for a specific network layout and specific router. If yours is a different setup or router you will have to take the above as a general guide. While the specific actions may vary, the principles are the same. You need to establish a port forwarding system.

WARNING!!!: Port forwarding is a hole in your fire wall. While it is not much use to any hackers you may want to experiment with a machine that does not have any sensitive data.