Complete Socket FAQ

Background:

Previously, REALbasic only supported one socket protocol, which is TCPIP. This is a connection-oriented protocol which has been an internet standard for years. There are two different modes of operation for a socket, synchronous and asynchronous. Synchronous sockets operate under the assumption that once the socket call has been made, it will not return control to the calling program until after it has received some form of response. This means that either an error has occurred, or the function has completed. An example would be: if you say Socket.Connect, control will not be given back to your program until the socket has connected, or it has determined there was an error with the connection process. Synchronous sockets, while easy to program for, have the tendency to be slow. This is because the majority of the time is spent in waiting for the socket instead of doing something useful, like queuing data to send once the connection is complete. The other mode of operation is asynchronous. In this mode, the socket function calls return control back to the calling program immediately. The calling program will receive a notification via a callback function letting it know whether there was an error, or the function completed properly. To go back to the previous example, an asynchronous call to Socket.Connect will return immediately, but you won't know the outcome of the process until you receive a Socket.Connected event, or a Socket.Error event. Using sockets in asynchronous mode tend to be faster than their counterpart because it gives the calling program the control it needs to do other things, such as process user input, etc. REALbasic operates using asynchronous mode for it's Socket class. This means your sockets will work faster. But due to this feature, there are some caveats you need to be aware of.

With version 5.0, there are many changes to the Socket class, including new features, and a better underlying implementation. This FAQ only describes the Socket functionality for version 5.0 of REALbasic, and may or may not be backwards compatible with information that held true for earlier versions of RB.

Please note that the ServerSocket and the SSLSocket are Pro-Only features of REALbasic.

Major Overhaul:

With REALbasic 5.0, sockets have gone through a major overhaul. The first major change was the introduction of a new base class which sockets derive from. The new class is named SocketCore, and it handles the core functionality for the sockets that RB provides. This base class provides all the common functionalities between the two protocols that REALbasic provides. The current two protocols are TCP/IP, and UDP sockets. The SocketCore base class is an abstract class, and cannot be instantiated by itself.

From this new SocketCore base class springs two subclasses, TCPSocket and UDPSocket. The old Socket class has been deprecated (tho it is still functional), and so you are encouraged to use the new TCPSocket class in it's stead. The new TCPSocket class has gotten new functionality added on to it, as well as being re-written for Mac OS 9 and OS X.

You can check out the specs for all socket related classes at the end of this document.

Caveat One:

Just because you have called TCPSocket.Connect does not guarantee that you are connected to the remote side. All this means is the connection process has begun. Sometimes, calling TCPSocket.Connect (especially to localhost, or 127.0.0.1) will result in what seems like an instant connection. Do not be fooled by this into thinking it will work that way always! Due to network latency, among other issues, it is possible for your socket to take a long time to finish the connection process. Due to this functionality, you should not call TCPSocket.Write until you know you are connected. The two ways to know that you are connected is to either wait until you receive a TCPSocket.Connected event, or by testing the SocketCore.IsConnected status. If you fail to adhere to this, you will most likely (though not always) cause the connection process to halt, and your application will receive an error (102).

Caveat Two:

When you say TCPSocket.Write, you are beginning the process of sending data across the wire. Certain low-level providers have limits on the max amount of data the socket can send in one batch. This is dependent on a few factors. What library is providing the TCPIP services (such as OpenTransport on Mac and WinSock on Windows), and how much data you are trying to send. You might think that the provider will only affect transfers of large data, but this is not true. I would suggest this: never assume how much data will get sent by REALbasic between calls to TCPSocket.SendProgress. It is natural to see this fluctuate. Both providers we use internally

specify the max and min amount of data it will send. If you are trying to send data larger than the max, it will not send it all in one chunk. Instead, we will loop until your data is completely sent, giving you periodic TCPSocket.SendProgress events. If you try to send too little data, the underlying provider will queue your data up. This doesn't always mean your data has been sent (though you WILL receive a TCPSocket.SendProgress and SocketCore.SendComplete event). This is due to the underlying provider implementing the Nagle algorithm. This algorithm helps network productivity. For every chunk of data that is sent across the network, there is a header attached to the beginning of that data. You never will have to deal with these headers because they are taken care of for you by the underlying provider. The reason this is important though is, if you are sending one and two bytes at a time across the network, you are also attaching these 40 or so bytes of headers to each send. This can bog down a network extremely easily, unless the Nagle algorithm is implemented. Both WinSock and OpenTransport implement this algorithm. Currently, REALbasic does not allow the user to turn this feature off, and leaves it set to the default. Note that, if you send only one byte of data, and never send anymore, the system will still send your one byte out even though the Nagle algorithm is enabled. The conclusion of this caveat is: do not assume knowledge of when your data has completed being sent. Rely on the SocketCore.SendComplete event to tell you when the data has completed. Also, do not rely on the bytesSent parameter of the TCPSocket.SendProgress event to be the same value every time. This value will change based on how many bytes of data the underlying provider was able to send. Note: if you are going to be sending small chunks of data across a network (especially a small network), it might make more sense to use the UDPSocket class instead.

New Functionality One:

One of the new features of sockets in REALbasic is the ability to orphan a socket. This has become a necessity because of the new ServerSocket class. In making this new functionality possible, we have introduced some new behavior to the socket class. When you say SocketCore.Connect, TCPSocket.Listen, or return a socket in ServerSocket.AddSocket(*see special note at the end of this description), your socket has it's reference incremented. This means that the socket does NOT have to be owned by the window in order for it to continue functioning. This proves helpful in certain circumstances. An example would be: You wrote your own socket subclass that implements all of the events for the socket, called MySpiffySocket. Then somewhere in the action event for a push button, say, you can use the following snippet of code:

```
Dim s as MySpiffySocket

s = new MySpiffySocket
s.port = 7000
s.address = "somecool.server.com"
s.connect
```

and the socket will continue to live and stay connected, even though there is nothing owning a reference to that socket (except the within the push button's action event). Due to this new functionality, a socket will continue to live until you tell it to die. If you have dragged a socket out to a window, and then said SocketCore.Connect, there are now two references to the socket, whereas in previous versions of REALbasic there was only one reference (the window's). When you close your window, your socket will automatically be closed and the connection terminated, and it's reference released. As stated before, an orphaned socket will continue to function until it has an error (or the application quits), and then it will be released.

*Note about ServerSocket.AddSocket: The reference isn't incremented as soon as you return the socket in that event. Instead it is pooled internally, and it's reference is incremented when the server hands off a connection to that socket. So if the server socket is destroyed before it uses one of these pooled sockets, the unused sockets get destroyed as well. If a socket returned from the ServerSocket.AddSocket event has been handed a connection, and then the server socket is destroyed, the socket will remain connected and continue to function.

Caveat Three:

The new functionality described above tells you that a socket can be orphaned. This does not hold true for a server socket. A server socket MUST have a reference holder. If it does not, then once the server socket goes out of scope in your code, it is destroyed.

Caveat Four:

A ServerSocket can only return a TCPSocket in it's AddSocket event. Since UDP is a connectionless

protocol (see description of the UDP protocol), it does not make sense for a ServerSocket to deal with with UDPSockets.

Caveat Five:

This really is more of a "watch out for this" than a caveat. And it's been around since the dawn of time with EVERY application that uses sockets, not just REALbasic. When you say TCPSocket.Write, the string gets added to an internal buffer, and we start writing the data out to the socket. If you have multiple calls to TCPSocket.Write in a sequential order, sometimes the writes will get strung together. An example would be: if you have a chat program, and the user clicks the send button 5 or 6 times in a row really fast to send the string "Hello World" to another socket, the other socket will sometimes receive "Hello WorldHello World". This is because you have added information to the buffer fast enough that the previous send hadn't completed yet, so you have two of your messages on the buffer for the next time thru the send loop. This is not a bug! It's expected functionality. In order to avoid this process, I would recommend you packetize your information. Have a header, and data attached to it. An example would be: send <Msg>Hello World</Msg> instead of just Hello World. You have a beginning tag that your receiving end can parse out, followed by data, and an ending tag. Then if the information gets strung together into <Msg>Hello World</Msg><Msg>Hello World</Msg>, you can parse it, and realize there are two messages there instead of one. Another approach is to have the length of the packet in the header so you know how much data belongs to one packet.

Caveat Six:

Due to the nature of the UDP protocol, there are certain things you cannot take for granted with a UDPSocket than you can with a TCPSocket. UDP does not guarantee that your data will make it to it's intended target. Also, UDP does not guarantee the order in which you send packets out will be the same as the order in which the remote side receives the packets. See the discussions about the TCP and the UDP protocols for more details.

Caveat Seven:

Due to the way the RAS Manager works on Windows, REALbasic cannot determine which connection in your dialup connection list is the default. This is a limitation due to the APIs the OS allows us. Because of this fact, if you call System.PPPConnect in non-user interactive mode (by passing in false), RB will use the FIRST entry it finds in the connection list.

Known Issue One:

Currently, setting SSLSocket.Secure to true and then saying SSLSocket.Listen will NOT create a secure listening socket. Doing so will create a non-secure listening port on your machine at the port specified, and any connections received will not be secured. A secure listening feature may be added later to REALbasic.

Known Issue Two:

On Mac OS X, attempting to bind to a port < 1024 will cause a SocketCore.Error event to fire with an error # 105 unless you’re application is running with root permissions. This is due to Apple's implementation with OpenTransport, and currently, there are no workarounds.

Known Issue Three:

Setting the SendToSelf property may not work on all versions of Windows. MSDN states that the SendToSelf property on Windows 95 and NT 4 can not be turned off. A mutlicasting socket will always receive the data it sends out.

Known Issue Four:

There is a limit to the number of sockets your application can have opened concurrently on Mac OS X. This is because OpenTransport uses the underlying BSD system to implement sockets. BSD uses a file descriptor for each open socket connection. The standard limit on OS X is set to 256 file descriptors. This means that at most you can have 256 sockets connected at a time per application. This number tends to be less than 256 in practice though, because your application may have files open, or the underlying API calls might be using a file descriptor for their purposes. This is not an issue on Windows, or Mac OS 9, and it is not a bug with REALbasic. It is a caveat of the underlying BSD system. For more information, and a possible workaround, check out this page: http://support.realsoftware.com/feedback/index.php?p_public_id=uvadknxt

Old Workaround No Longer Needed:

In previous versions of REALbasic, you could speed up your send process by a fair amount by "tickling" the socket with a timer. You would do this by having a timer say Socket.Poll in it's action event while the socket was doing a send. This trick should no longer be needed, though it isn't detrimental to leave it in. Sockets now internally keep track of the fact that they have more data to send, and so are in a tighter send loop. Once the data has finished sending, the socket no longer takes up as much processor time.

Deprecation One:

Socket.PPPConnect, Socket.PPPStatus and Socket.PPPDisconnect is now deprecated. This is because it really doesn't make much sense. A PPP connection (point to point protocol) is a dialup connection that must occur before a TCPIP connection can be used. It is a feature that is used when ethernet is not available to dial in to a server for your internet access. This is not a feature that is used on a socket-by-socket basis. So saying Socket1.PPPConnect and Socket2.PPPDisconnect will affect ALL sockets on the system. In fact, saying Socket.PPPDisconnect affects sockets that do not even belong to REALbasic! These features are system-wide feature, and thus have been moved to the System class. Instead of using the Socket.PPP functions, please begin using their System counterparts. These are System.PPPConnect, System.PPPStatus and System.PPPDisconnect. The functionality of these features remains the same.

Deprecation Two:

The old Socket class has been deprecated, and replaced with the new TCPSocket class. The old Socket class still functions identically to the new TCPSocket class. We highly recommend that you use TCPSocket for version 5.0 of REALbasic.

New Features and Nifty Ideas:

* TCPSocket send speeds are faster. Due to the internal loops automatically tightening, and sending the maximum amount of data that the underlying protocols allow per send, you should see an increase in the speed of your TCPSocket.Write calls. This new feature is a Mac only feature.

* Better socket stability. The socket code now gracefully handles the disconnection process. It tries to do an orderly disconnect when possible (which allows for all data transfers to finish), and will only fall back on an abortive disconnect when it is not possible to do an orderly one. Sockets now handle flow control, so sending large amount of data out will not drop data during the send or the receive process. It is possible for packets of data to come in before the socket has finished the connection process. In the event this happens, the new socket code is prepared to handle this, so your initial packets will not be lost either. This new feature is a Mac only feature; our Windows sockets were already prepared to handle these situations.

* TCPSocket.SendProgress event. This new event allows you to determine send speeds, and tells you how many bytes of data you have sent since your last SendProgress event, as well as how many bytes are left to send. By returning true from the send progress event, you are canceling the current transfer. This does NOT close the socket's connection, just clears the send buffer. You can use this new event to determine that a connection is too slow, and cancel it. Once all of the data has been transferred, you will get a last TCPSocket.SendProgress event, followed by a SocketCore.SendComplete event. This is a new feature to both Mac and Windows.

* SocketCore.SendComplete now gets a parameter. This parameter will let you know whether the transfer has completed or has been cancelled by returning true from TCPSocket.SendProgress. You can use this information to update different status variables, or alert the user of transfer success or failure. If the user aborted, this parameter is true, and if the send was completed, this value is false. This is a new feature to both Mac and Windows. SocketCore.SendComplete's parameter will always be false for UDPSockets since there is not a SendProgress event for that class.

* TCPSocket.RemoteAddress for a connecting socket now returns the IP address of the remote host connection for Windows sockets. In previous versions of REALbasic, a connecting socket on Windows would return a blank string if you asked it for the TCPSocket.RemoteAddress. This issue has been fixed, and it will now return to you the correct IP address of the remote connection. Some handy things you can do with this information (on either platform): test to make sure the IP address you WANTED to connect to is the same as the IP address you are currently connected to.

* TCPSocket.Disconnect will now close your socket, and fire the SocketCore.Error event, passing in a 102 error to let you know that the socket has been disconnected.

* SSLSocket has been added as a subclass of TCPSocket. With this new feature, you can now create

Secure Socket Layer sockets to connect with. You can choose the protocol you wish to connect with by setting the SSLSocket.ConnectionType property. There are currently four different protocols supported by REALbasic. They are:

0 - SSLv2	SSL (Secure Sockets Layer) version 2
1 - SSLv23	SSL version 3, but can rollback to 2 if needed
2 - SSLv3	SSL version 3
3 - TLSv1	TLS (Transport Layer Security) version 1

The default protocol used is SSLv23, and is compatible with most SSL servers. You must set this property BEFORE you say SSLSocket.Connect. Trying to set the protocol after you have begun the connection process will have no effect.

* System.PPP functions. In addition to deprecating the Socket.PPP functions, some work was done on the Windows side to improve them. PPP connections are now established using RAS (Remote Access Service) connections. On Windows 2000/XP, the standard RAS connection dialog will appear when you say System.PPPConnect, and allow the user to choose which phonebook entry they wish to connect with. On all other Windows platforms, these dialogs are not available. Due to this fact, REALbasic will attempt a dialup connection using the default phonebook entry, including the username and password supplied. If either of these fields are missing, the connection process will fail, and a dialog box will appear telling the user the reason for failure.

* System.PPPConnect now takes an optional parameter which gives you the choice to allow user intervention during the dial-up process, or whether you want it to run by itself. Some of your users may have more than one ISP they can dial, and the default may not always be who they want to call. If this is the case, call System.PPPConnect(true) to allow user intervention, when possible. This feature is not available on all systems, so please see the "Things to Watch Out For" section for more information.

* ServerSocket has been added. This is a totally new concept to REALbasic, and one that was previously very difficult to achieve using our Socket class. A server socket is a permanent socket that listens on a single port. When a connection attempt is made on that port, the server socket hands the connection off to another socket, and continues listening on the same port. Previously it was difficult to obtain this functionality due to the latency between a connection coming in, being handed off, creating a new listening socket, and beginning the listening process. If you had two connections coming in at relatively the same time, one of the connections would be dropped due to there not being a listening socket on that port.

* Sockets now appropriately handle multiple simultaneous incoming connection requests. In previous versions of REALbasic, this phenomenon would provide a range of results, including: crashes, hangs and dropping both connections. This bug has finally been fixed for the Mac platform. This was not an issue with Windows sockets, and continues to be a non-issue on that platform.

* An implementation of the UDP protocol has been added to REALbasic 5.0 in the form of the UDPSocket class. A description of the class is at the end of this document. Please also see the discussion of the differences between the TCP and UDP protocols.

* The UDPSocket class makes use of a new data structure, called Datagram. This class is used to store data, as well as an IP address, for purposes of sending and receiving data. When you call UDPSocket.Read, the Datagram will contain the originating machine's IP address, and the data that machine sent to you. When calling UDPSocket.Write (assuming you pass in a Datagram), it should contain the data you wish to send, and IP address of the remote machine. This IP address could also be a multicast address, or the broadcast address.

Things to Watch Out For:

* Circular references. It is often logical for you to have a socket associated with some other data type, such as an EditField subclass, or something along those lines. In this case, the EditField subclass has a reference to the socket. When the socket is done with whatever you set it out to do, it's often necessary to let the EditField subclass know that the socket is done. This might mean that your socket has a reference to your EditField subclass (if your socket is subclasses as well). This will cause a circular reference! You will leak memory if you have a circular reference! You can break the circular reference by setting one of the two objects to nil (most likely the socket since it is done with what the EditField had it doing).

* Orphaned sockets that never close. If you orphan a socket, you need to be certain that at some point, that socket will get a disconnect message from it's connection. For example, some web servers will not release a socket

once the data has been transferred to it (for efficiency purposes). If your application quits with your socket in this state, then that socket is lost to the system (and leaked) until the machine is rebooted (or the web server finally releases the socket). This could cause crashes with your application, and the system! If you think your socket could be in a state where it is left open, keep a reference to the socket around somewhere, and use the SocketCore.Close method to terminate the connection.

* Once your socket has received an Error event, it has been closed. The connection is torn down, and the internal buffers have been released. This means that once an error has occurred, and you have left the SocketCore.Error event, the socket will no longer be connected, has nothing in it's send or receive buffers, and is ready to be used again (without calling SocketCore.Close). The information that is retained is: the socket's port, address (in the case of TCPSockets) and LastErrorCode properties.

* Calling SocketCore.Close tears down the socket, and will close any connections the socket may have. The only information that is retained after calling SocketCore.Close is the socket's port, address (in the case of TCPSockets) and LastErrorCode properties. All other information is discarded.

* System.PPPConnect behaves slightly differently, depending on how you use it. If you pass in the value 'true', on the Mac, there will be no intervention. There are no standard dialogs provided for the user to choose which connection to use on Macintosh. If you pass in 'true' on Windows, and the user is running on NT 4 or later, then the standard RAS Manager dialogs will appear, and ask the user for connection information. If the user is running Windows 95/98/ME, or you pass in 'false' (or leave the parameter blank), then the system will attempt the connection using the first phonebook entry it can find.

Miscellaneous Tidbits of Information:

* TCPSocket.Address is used only to specify the address of where to connect to. REALbasic does not modify this property at all. This is in contrast to TCPSocket.RemoteAddress, which specifies the remote IP address of the machine you are trying to connect to. If you set TCPSocket.Address to "www.google.com", then TCPSocket.RemoteAddress, upon connection, will be set by RB to Google's IP address ("216.239.39.101"). Before the connection has occurred, and after the connection terminates, TCPSocket.RemoteAddress will be an empty string. What this means is, don't rely on TCPSocket.Address to give you any useful information. It is there strictly to tell REALbasic where to attempt a connection to. If you want information about the connection, use TCPSocket.RemoteAddress.

* Because REALbasic does not modify the TCPSocket.Address property, there can be situations where this property is an empty string. For example, when using a ServerSocket, the TCPSockets that have a connection handed off to them will not have their TCPSocket.Address property set.

Protocols Explained:

TCP:

The *Transmission Control Protocol*, or TCP, is the basis for most internet traffic. It is a connection-oriented protocol that provides a reliable way to transfer data across a network. Because of this principal, all TCP sockets follow a similar procedure for use. To establish a connection between two computers (to be able to send data back and forth), one computer must be set up to listen on a specific port. The other computer (called the client) then attempts to connect by specifying the network address (or IP address) of the remote machine and the port to attempt the connection on. This means that in order to send and receive data with a remote machine, both machines must have some indication that this connection will be established. That happens by either picking a well-defined port for the listener (or server) to listen on, or by some prior arrangement (eg. you are the author of both the server and the client program). When a server receives a connection attempt for the port it is listening on, it accepts the incoming connection, and sends an acknowledgement back to the remote machine. Once both machines have reached an agreement (or are "Connected"), then you can begin sending and receiving data. When you close your connection with the remote machine, there is a similar handshake process that goes on, so both computers know that the connection is being terminated.

Due to the amount of error checking, and handshakes, TCP is an extremely reliable protocol to use. When you send a packet of information out, it is guaranteed to make it to the remote machine (assuming you have not been disconnected, either abortive or orderly). But this feature comes at the cost of high overhead. A typical TCP packet that is sent over the network has around a 40-byte header that goes with it. This header is checked, and changed by all the various machines en route to it's destination. This overhead makes TCP a slower protocol to use. The

tradeoff is speed vs security, in favor of security. If it is speed you are looking for (like to write a networked game), then you should look into the UDP protocol.

UDP:

The *User Datagram Protocol*, or UDP, is the basis for most high speed, highly distributed network traffic. It is a connectionless protocol that has very low overhead, but is not as secure as TCP. When you wish to use a UDP socket, since there is no connection, you do not need to take nearly as many steps to prepare. In order to use a UDP socket, it must be bound to a specific port on your machine. Once the bind has occurred, the UDP socket is ready for use. It will immediately begin accepting any data that it sees on the port it has bound to. It also allows you to send data out, as well as set UDP socket options (which will be described later). In order to differentiate between which machine is sending you what data, a UDP socket receives a data structure known as a Datagram. A Datagram consists of two parts, the IP address of the remote machine which sent you the data, and the payload (or the actual data itself). When you attempt to send data out, you must also specify information in the form of a Datagram. This information is the remote address of the machine you want to receive your packet (this is not entirely true, so please read further), the port it should be sent to, and the data you wish to send the remote machine.

UDP sockets can operate in various modes, which are all very similar, but have vastly different uses. The mode which most resembles a TCP communication is called 'unicasting'. This occurs when the IP address you specify when you write data out is that of a single machine. An example would be sending data to www.google.com, or some network address. It is a Datagram which has one intended receiver.

The second mode of operation is called 'broadcasting'. Just as the name implies, this is a broadcasted write. It is akin to yelling into a megaphone. Everyone gets the messages, whether they want to or not. If the machine happens to be listening on the specific port you specified, then it will receive the data in. This type of send is very network intensive. As you can imagine, broadcasting can amount to huge amounts of network traffic. The good news is, when you broadcast data out, it does not leave your subnet. Basically, a broadcast send will not leave your network to travel out into the world. When you want to broadcast data, instead of sending the data to an IP address of a remote machine, you specify the broadcast address for your machine. This address changes from machine to machine, so RB provides a property on the UDPSocket class which tells you the correct broadcast address.

This brings us to the third mode of operation for UDP sockets: 'multicasting'. It is a combination of unicasting and broadcasting that proves to be very powerful and practical to use. Multicasting is a lot like a chat room: you enter the chatroom, and are able to hold conversations with everyone else in the chatroom. When you want to enter the chatroom, it is called JoinMulticastGroup, and you just need to specify the group you wish to join. The group parameter is a special kind of IP address, called Class D IPs. They range from 224.0.0.0 to 239.255.255.255. Think of the IP address as the name of the chatroom. If you want to start chatting with two other people, all three of you need to call JoinMulticastGroup with the same Class D IP address specified as the group. When you wish to leave the chatroom, you just need to call LeaveMulticastGroup, and again, specify the group you wish to leave. You can join as many multicast groups as you would like, you are not limited to just one at a time. When you wish to send data to the multicast group, you just need to specify the multicast group's IP address. All people who have joined the same group as you will receive the message.

Multicasting has some extra features which make it an even more powerful utility for network applications. If you wish to receive the data you sent out with a multicast send, you can set the SendToSelf (also known as loopback) property on the socket. If it is set to true, then when you do a send (to a multicast group) you will get that data back. You can also set the number of router hops a multicast datagram will take (also known as the Time to Live). When your datagram gets sent out, it runs thru a series of routers on the way to it's destinations. Every time the datagram hits a router, it's RouterHops property gets decremented. When that number reaches zero, the datagram is destroyed. This means you can control who gets your datagrams with a lot more precision. There are some "best guesses" as to what the value of RouterHops should be.

0	-- same host
1	-- same subnet
32	-- same site
64	-- same region
128	-- same continent
255	-- unrestricted

Note that if your Datagram runs through a router that does not support multicasting, it is killed immediately.

Due to the connectionless functionality of UDP, it does not make any guarantees that your data will reach

it's destination. You can work around this by creating your own protocol on top of the UDP protocol which acknowledges receives.

PPP:

The point-to-point protocol, or PPP, is the way to gain a connection to the Internet with a dial-up modem. It is a system-wide functionality that you can use to get the modem to dial out to an ISP, and upon successful connection, you TCPSocket and UDPSocket code will function. Due to the system-wide nature of PPP, the calls that were attached to Socket have been moved to the System class. Note that if you say System.PPPDisconnect, it will terminate the connection to the Internet. This means that you will kill other applications' connections as well as your own, so be sure to ask the user if they want the connection terminated before happily killing all connections to the Internet!

Socket Errors Explained:

The SocketCore class has a property called LastErrorCode, which is an integer value, specifying what the last error code is. These error codes provide you with key information about your socket, and it is not advisable to ignore them. Here is a list of the current errors, and what they mean to you.

- 100 -- There was an error opening and initializing the drivers. Generally, this means that either OpenTransport (on the Mac), or WinSock (on Windows) is not installed, or the version is too low.
- 101 -- This error code is no longer used. You will not see any 101 errors in RB 5.0
- 102 -- This is an error you will see more often than most. It means that you lost your connection. You will get this error if the remote side disconnects (whether it's forcibly, by pulling their ethernet cable out of their computer), or gracefully (by saying SocketCore.Close). This may or not be a true error situation. If the remote side closed the connection, then it's not truly an error, it's just a status indication. But if they pulled the ethernet cable out of their computer, then it really is an error; but the results are the same. The connection was lost. You will also get this error if you call TCPSocket.Disconnect.
- 103 -- You will get this error if we cannot resolve the address you specified. A prime example of this would be a mistyped IP address, or a domain name of an unreachable host.
- 104 -- This error code is no longer used. You will not see any 104 errors in RB 5.0
- 105 -- The address is currently in use. This error will occur if you attempt to bind to a port that you have already bound to. An example of this would be setting up two listening sockets to try to listen on the same port.
- 106 -- This is an invalid state error, which means that the socket is not in the proper state to be doing a certain operation. An example of this would be trying saying TCPSocket.Write before the socket is actually connected.
- 107 -- This error means that the port you specified is invalid. This could mean that you entered a port number less than 0, or greater than 65,535. It could also mean that you do not have enough privileges to bind to that port. This happens primarily under OS X if you are not running as root. You can only bind to ports less than 1024 if you have root privileges in OS X.

These are not the only errors that you can get from SocketCore.LastErrorCode. If REALbasic cannot adequately map the underlying provider's error code to one of the above codes, we will pass you the provider's error code. Traditionally, for the Mac, these error codes are negative numbers in the range [-3211, -3285]. For Windows, these error codes are usually positive numbers, in the range [10004, 11004]. For a description of the Macintosh error codes, find a copy of MacErrors.h, and for Windows error codes, find a copy of WinSock.h.

PPP Status Codes Explained:

When calling System.PPPStatus, there are a number of codes returned to you, that thus far have been confusing, not well documented, and sometimes incorrect. Here is the definitive explanation about the status values returned.

- 0 -- There is no connection present. This means that you haven't called System.PPPConnect, or that the connection process failed. It could also mean that you have called System.PPPDisconnect, and the connection has been disconnected.
- 1 -- Not used
- 3 -- The connection is being closed. This means that you have called Sytem.PPPDisconnect, and the

disconnection process has begun. It does not mean that the disconnect is complete.
- 4 -- The connection is being attempted. This does not mean that you have a valid internet connection, and so using your TCPSocket or UDPSockets may cause an error.
- 5 -- You have a valid connection. This means that you can use your TCPSocket and UDPSocket code because you are fully connected.

Socket Classes:

SocketCore:

Properties:
- Port as Integer -- specifies the port to bind on or connect to
- LocalAddress as String -- specifies the local IP address for the machine (*Read Only*)
- LastErrorCode as Integer -- specifies the last error for the socket (*Read Only*)
- IsConnected as Boolean -- specifies whether the socket is currently connected or not. For TCPSockets, this means you can send and receive data, and are connected to a remote machine. For UDPSockets, this means that you are bound to the port and able to send, receive, join/leave multicast groups, or set socket options. (*Read Only*)

Methods:
- Close() -- closes the socket's connection and resets the socket
- Poll() -- polls the socket manually (allows a socket to be used synchronously)
- Purge() -- removes all data from the socket's internal receive buffer
- Connect() -- attempts to connect. For TCPSockets, the address and port property must be set. For UDP sockets, the port property must be set.

Events:
- DataAvailable() -- occurs when some more data has come into the internal receive buffer
- Error() -- occurs when an error occurs with the socket
- SendComplete(userAborted as Boolean) -- occurs when a send has completed. userAborted will always be false for UDP sockets.

TCPSocket (inherits from SocketCore):

Properties:
- Address as String -- specifies the address to try to connect to
- RemoteAddress as String -- specifies the IP address of the remote machine you have a connection with. (*Read Only*)
- Lookahead as String -- shows the data that is available in the internal queue without removing it (*Read Only*)
- BytesAvailable as Integer -- tells you how many bytes of data are available in the internal receive buffer (*Read Only*)
- PPPStatus as Integer -- Deprecated. (*Read Only*)

Methods:
- Listen() -- attempts to listen on the currently specified port for incoming connections.
- Read(bytes as Integer) -- reads the amount of bytes specified from the internal receive buffer
- ReadAll() -- reads all the data from the internal buffer
- Write(data as String) -- writes out the string to the remote connection
- Disconnect() -- disconnects the socket, resets it, and fires a socket 102 error.
- PPPConnect() -- Deprecated.
- PPPDisconnect() -- Deprecated.

Events:
- Connected() -- occurs when a connection has been established.
- SendProgress(bytesSent as Integer, bytesLeft as Integer) as Boolean -- tells you that some progress has been made during the send. Returning true from this event causes the send to be aborted.

UDPSocket (inherits from SocketCore):

Properties:
- RouterHops as Integer -- specifies how many routers hops the data sent out can make. This is also

known as the Time To Live (or TTL). This property only applies to multicast sends.
- SendToSelf as Boolean -- specifies whether the data you send out will be sent to yourself as well. This is also known as loopback. This property only applies to multicast sends.
- PacketsAvailable as Integer -- tells you how many packets are available in the internal receive buffer (*Read Only*)
- BroadcastAddress as String -- specifies the machine's broadcast address. You can specify this address in a Write, and the data will be broadcast across the network (but you will not receive the data you sent back). (*Read Only*)

Methods:
- JoinMulticastGroup(group as String) as Boolean -- attempts to join the specified multicast group. If the socket successfully joined, it returns true. You can join as many multicast groups as you'd like. The group parameter specifies a Class D IP address, in the range: 224.0.0.0 thru 239.255.255.255
- LeaveMulticastGroup(group as String) -- leaves the specified multicast group
- Write(address as String, data as String) -- constructs a Datagram and sends the data to the specified address. If the address is that of a multicast group, all members of that group will get the data. If the address is the broadcast address, everyone on the network will get the data. If the address is an IP address, then the data is unicast to just that address.
- Write(data as Datagram) -- writes the data to the address specified.
- Read() as Datagram -- retrieves a datagram from the internal receive buffer. The address member of the datagram is the remote address from which the data was sent.

Events:
- None.

Datagram:

Properties:
- Address as String -- specifies an IP address. It is the address of the remote machine which you are sending data to, or receiving data from.
- Data as String -- specifies the data that you are sending out, or have received.

Methods:
- None

Events:
- None

ServerSocket:

Properties:
- Port as Integer -- specifies the port to bind to for listening
- MinimumSocketAvailable as Integer -- specifies the minimum number of sockets to keep around at any given point in time. If the server falls below this threshold of available sockets, it will call the AddSocket event to replenish it's supply of sockets.
- MaximumSocketsConnected as Integer -- specifies the maximum number of sockets that the server will allow to connect. When a socket disconnects (that was connected from the ServerSocket), the server will allow one more connection.
- LocalAddress as String -- specifies the local IP address of the machine. (*Read Only*)
- IsListening as Boolean -- specifies that the ServerSocket is listening for incoming connections. (*Read Only*)

Methods:
- Listen() -- starts the server listening.
- StopListening() -- closes the ServerSocket so it can no longer accept incoming connections.

Events:
- AddSocket() as TCPSocket -- occurs when the server socket needs a socket for it's internal pool. Will get called when the ServerSocket first begins listening.
- Error(errorCode as Integer) -- occurs when the server socket has an error

System.PPP* Functions

Methods:

- PPPConnect([userInteraction as Boolean]) -- attempts a dial-up connection, might be with user interaction
- PPPDisconnect() -- disconnects the dial-up connection

Properties:

- PPPStatus() as Integer -- retrieves the current application's PPP status (*Read Only*)

SSLSocket (inherits from TCPSocket):

Methods:

- None

Properties:

- Secure as Boolean -- Specifies whether you want the Connect method to obtain a secure connection. If this value is false, then you will have a regular TCP (non-secure) connection. You must set this to true before attempting the connection.
- SSLConnected as Boolean -- Returns true if the connection is currently secure, false if the connection was made insecurely. (*Read Only*)
- SSLConnecting as Boolean -- Returns true while attempting a secure connection, false once the connection has been made (or before attempting the connection) (*Read Only*)
- ConnectionType as Integer-- Sets the connection type to attempt when calling Connect. See the "New Features and Nifty Ideas" section for the various connection types REALbasic currently supports. This property must be set before attempting the connection.

Events:

- None

If you have questions, comments, additions or errata, please contact aaron@realsoftware.com or support@realsoftware.com

Version History:
Sep 30 2002 -- AJB -- Initial write
Oct 29 2002 -- AJB -- Added information about SocketCore and UDPSocket classes
Nov 06 2002 -- AJB -- Added information about the Datagram class
Nov 07 2002 -- AJB -- Added information about socket error codes
Nov 12 2002 -- AJB -- Added more information about our PPP implementation, updated socket error codes
Nov 19 2002 -- AJB -- Added some useful miscellaneous information about Address and RemoteAddress
Nov 19 2002 -- AJB -- Added the IsListening property to ServerSocket
Nov 20 2002 -- AJB -- Specified which properties were read only, added documentation for SSLSocket