REALbasic Socket ReadMe

Table of Contents:

Background:

This document describes the socket functionality in version 5.5 of REALbasic. It might not be correct for earlier versions, though we try to keep this document updated as of REALbasic version 5.0.

Prior to version 5.0, REALbasic supported only one socket protocol: TCP/IP. This is a connection-

oriented protocol that has been an Internet standard for years. Then, with version 5.0, many new features and changes were implemented (including adding support for new protocols such as SSL and UDP). For version 5.5, there are many improvements to the existing socket functionality, as well as new features and a better underlying implementation for Mac OS X.

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

Synchronous vs Asynchronous Operation:

In general, there are two modes of operation for a socket: synchronous and asynchronous. ***REALbasic sockets are always asynchronous*** (though synchronous operation can be simulated by the calling code).

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. Example: if you say Socket1.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, tend 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.

Asynchronous sockets 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 Socket1.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.

Asynchronous sockets tend to be faster than their synchronous counterparts because they give the calling program the control it needs to do other things, such as process user input, etc. This means your sockets will work faster. But there are some caveats you need to be aware of (see below).

Overhaul for 5.0:

With REALbasic 5.0, sockets have gone through a major overhaul. The first major change was the introduction of a new base class, SocketCore, from which other socket classes inherit. This class handles the core functionality for the sockets that RB provides. SocketCore is an abstract class; that is, it cannot be instantiated by itself.

From this new SocketCore base class, two new subclasses are derived: TCPSocket and UDPSocket. The old Socket class has been deprecated (though it is still functional), and you are encouraged to use the new TCPSocket class instead. TCPSocket has new functionality, and has been rewritten for Mac OS 9 and OS X.

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

Changes for 5.5:

With REALbasic 5.5, sockets have gone through another overhaul for Macintosh OS X. With Apple moving further away from supporting OpenTransport (in fact, they officially dropped support for OpenTransport when OS 10.3 was released), we decided to move the underlying implementation from OT to using straight BSD sockets. So again with version 5.5, sockets have been rewritten under the hood to use the best technology provided by the operating system. Consequently, with this change, we were able to make our sockets behave identical on three separate platforms. Since OS X, Linux and Windows all have a POSIX-compliant (well, mostly) implementation of BSD sockets, the code under the hood is identical for all three of those platforms. This should provide for an identical experience on all three of those platforms where sockets are concerned.

In addition to the under-the-hood changes we made, we added some new classes to help you write networking code without worrying about all the little details. We now provide three new helper classes for networking: EasyUDPSocket, EasyTCPSocket and AutoDiscovery.

Caveat One:

Just because you have called TCPSocket.Connect does not guarantee that you are connected to the remote side. All it means is that 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. Because of this functionality, you should not call TCPSocket.Write until you know you are connected.

The two ways to know that you are connected are to either wait until you receive a TCPSocket.Connected event, or to test the return value of SocketCore.IsConnected. If you fail to adhere to this, you either cause the connection process to halt, resulting in a lost connection error (102), or an out of state error (106).

Caveat Two:

When you say TCPSocket.Write, you are beginning the process of sending data across the wire. Certain low-level socket service providers have limits on the maximum amount of data the socket can send in one batch. This is dependent on a few factors; among them are which library is providing the TCP/IP services (such as OpenTransport on Mac Classic, WinSock on Windows, etc), 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. Never assume how much data REALbasic will send between calls to TCPSocket.SendProgress. It is natural to see this fluctuate. Each provider that RB uses specifies the max and min amount of data it will send.

If you are trying to send data larger than the max, it will not be sent all in one chunk. Instead, RB 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, which 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 that if you are sending one and two bytes at a time across the network, you are also attaching these 40 or so bytes of header to each send. This can bog down a network extremely easily, unless the Nagle algorithm is implemented. All network providers that we use 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 will have been completely sent. Rely on the SocketCore.SendComplete event to tell you when the send 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.

Caveat Three:

One of the new features of sockets in REALbasic 5.0 is the ability to orphan a socket. (This is a necessity because of the ServerSocket class.) In making this new functionality possible, we have introduced some new behavior to the socket class. When you call SocketCore.Connect, TCPSocket.Listen, or return a socket in ServerSocket.AddSocket (*see caveat 3a), your socket's reference count is 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. Example: You write 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 use the following snippet of code:

```
Dim s as MySpiffySocket

s = new MySpiffySocket
s.port = 7000
```

```
s.address = "somecool.server.com"
s.connect
```

The socket will continue to live and stay connected, even though there is nothing owning a reference to it (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 to a window, and then called SocketCore.Connect or TCPSocket.Listen before closing the window, there will be two references to the socket, whereas in previous versions of REALbasic there was only one reference (the window's reference). In this case, the socket will continue to function until its connection is terminated, *even after the window has been closed.*

The termination can be done either locally (by calling TCPSocket.Close) or remotely (with the remote host terminating the connection). If you have not called Connect or Listen for your socket, then there will be only one reference to it (the window's), and it will be destroyed appropriately when the window closes. In either case, once the application terminates, all sockets are released gracefully, and your app will not leak memory.

Caveat Three-A:

The reference count isn't incremented as soon as you return a socket in ServerSocket.AddSocket. Instead, the socket is pooled internally, and its reference count 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. Until that time, the ServerSocket is the parent of the TCPSocket, and so the TCPSocket will stick around. If a socket returned from the ServerSocket.AddSocket event has been handed a connection, and then the ServerSocket is destroyed, the socket will remain connected and continue to function.

Caveat Three-B:

The new functionality described above tells you that a socket can be orphaned. This does not hold true for a ServerSocket or a UDPSocket. Each of these sockets MUST have a reference holder. If it does not, then once the socket goes out of scope in your code, it is destroyed. However, when the ServerSocket is destroyed, it will not terminate any of your already-made connections (if there are any). It will only destroy TCPSockets that have not been connected.

Caveat Four:

A ServerSocket can only return a TCPSocket (or a subclass of TCPSocket) in its 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 UDPSockets, or any subclass of a UDPSocket.

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 sequential order, sometimes the writes will get strung together.

Example: 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 hasn'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 recommend you packetize your information. Include a header with your data. Example: instead of just "Hello World", send "<Msg>Hello World</Msg>". You have a beginning tag that your receiver 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 each packet.

Caveat Six:

Due to the nature of the UDP protocol, there are certain things you cannot take for granted with a UDPSocket that you can with a TCPSocket. UDP does not guarantee that your data will make it to its 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 of the APIs in the operating system. Because of this, if you call System.PPPConnect in non-user interactive mode (by passing false), RB will use the *first* entry it finds in the connection list.

Caveat Eight:

When using UDPSockets, most operations require that you be bound to a local port. For example, you cannot set the SendToSelf flag if you have not bound yourself (using the SocketCore.Connect method). This is also the case for setting RouterHops and Join/LeaveMulticastGroup. So before you make these calls, be sure to set the SocketCore.Port property and call SocketCore.Connect on the socket, otherwise you will get an out-of-state error (106).

Caveat Nine:

Due to the speed increases with REALbasic 5.0 sockets, calling TCPSocket.Connect from within the TCPSocket.Error event can cause stack overflow exceptions. This situation will occur if you are flooding a server with requests that it is denying. The connection will occur, and the error will fire immediately. Then, from within the error event, you try to connect again, which (in turn) fires another error event. Use caution with socket events so that you do not get yourself into this type of situation. One workaround for this behavior is to use a timer to cause the connection process to happen. Another work-around is to set a flag before calling the initial TCPSocket.Connect call and checking the state of that flag before starting a new connection attempt.

Caveat Ten:

Know your connection types when using SSLSockets. Not all servers will accept a connection with the default ConnectionType (SSLv23). This is server-specific, and generally not known beforehand. I suggest working around it by making multiple connection attempts to the server. If the initial attempt is rejected (causes a 102 error), then try again with a different ConnectionType property set. Just be sure to have a way to terminate this process if none of the connection types works, or if you get an error other than 102 (*see Caveat Nine).

Caveat Eleven:

When you call ServerSocket.Listen, this begins an extensive process. The first thing that occurs is that the ServerSocket needs to fill its internal pool of handoff sockets. It does so by calling the AddSocket event. This event will be called until we have enough sockets in the internal pool of available sockets. Currently (as of 5.5), we will fill up to the MinimumSocketsAvailable plus ten extras. Note that if you return nil from this event, it will throw a NilObjectException. Only after this process has completed is the ServerSocket ready to hand off connections. Connections that come in while the server is populating its pool are rejected. To know when the ServerSocket is ready to accept incoming connections, check the ServerSocket.IsListening property.

Caveat Twelve:

On Mac OS X and Linux, attempting to bind to a port < 1024 will cause a SocketCore.Error event to fire with an error 105 unless your application is running with root permissions. This is due to security issues that can arise from allowing sockets to listen on privileged ports. This is not a bug, but a security feature that is provided in some operating systems.

Caveat Thirteen:

There is a limit to the number of sockets your application can have opened concurrently on Mac OS X (prior to OS 10.3). This is because BSD sockets use a file descriptor for each open socket (one that is currently bound to any port on the machine). The standard limit on OS X is set to 256 file descriptors, but this limit can fluctuate based on the amount of RAM in your machine. This means that you can have, at most, 256 sockets connected at once per application. In practice, this number tends to be less than 256, because your application might have files open, or the underlying API calls might be using a file descriptor for their purposes. This is not an issue on Windows, Mac OS 9 or (to a certain extent) Linux, and it is not a bug in REALbasic. It is a caveat of the underlying BSD system. For more information, and a possible workaround, check out this web page:
http://support.realsoftware.com/feedback/index.php?p_public_id=uvadknxt
Note that you can run into this issue on Linux, but it tends to be far less likely. The same Mach-O call can be made on Linux as well to reset the file descriptor limit for the user.

Caveat Fourteen:

One caveat to the message-based system we have implemented is that REAL Software reserves all command IDs less than 0 for internal use. That means, when you are sending messages, you should not use a command ID < 0 as it may very well cause issues with other classes (such as the AutoDiscovery class).

Caveat Fifteen:

Even though you have access to the TCPSocket.Write, Read and ReadAll methods in the EasyTCPSocket class, you should _never_ call them. Doing so will cause a RuntimeException to be raised (with an appropriate message set). This is so the internal protocol is enforced. The same is true for the EasyUDPSocket and AutoDiscovery classes.

Caveat Sixteen:

The easy networking classes were created to help you communicate with other REALbasic applications easier. As such, you cannot use things like the EasyTCPSocket class to write your own HTTPSocket. This is because we enforce a protocol under the hood (on both the sending and the receiving ends) that does not just send out raw data (like a regular TCPSocket or UDPSocket would). So, if you need to communicate with another application over the network (like an FTP server, or some other protocol), chances are, you will not be able to use the easy networking classes provided. This also applies to the AutoDiscovery class. AutoDiscovery is not Rendezvous (or Zeroconf); it is a proprietary protocol under the hood. Because of this, you will not be able to auto-discover things like iChat over your network.

Caveat Seventeen:

The UDP protocol allows you to send small packets across the network in a low-latency fashion. But with this extra freedom comes a few problems you need to watch out for. In addition to the fact that packets may be dropped or rearranged before they reach their destination, there is a concept of a maximum transmission unit (also called MTU). This MTU limit can fluctuate from system to system (on some systems, it's a user-specific setting that can be changed), but is typically around 1500 bytes. To be safe, always try to keep the packets you send out to having less than 1024 bytes of payload data. Failure to do so can cause the packets to be dropped by routers or rejected by network transports.

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 might be added to REALbasic later.

Known Issue Two:

Setting the SendToSelf property might not work on all versions of Windows. MSDN states that the SendToSelf property on Windows 95 and NT 4 cannot be turned off. A multicasting socket will always receive the data it sends out on these systems.

Known Issue Three:

System.PPPConnect, System.PPPStatus and System.PPPDisconnect are currently not supported for Linux and Mach-O builds. This may change in the future.

Obsolete Workarounds No Longer Needed:

In previous versions of REALbasic (before 5.0), 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 its 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 are in a tighter send loop. Once the data has finished sending, the socket no longer takes up as much processor time.

In old applications that I have come across, I have seen calls to Socket.Poll strewn throughout Socket event handlers. These calls are not necessary. The Poll method is used to allow RB to fire events to you, as well as update internal data structures. Since you are already in an event, then calling Poll might cause reentrancy issues. You should only call Poll if you are in a tight loop and need to give RB some time to update its internals.

Deprecation One:

Socket.PPPConnect, Socket.PPPStatus and Socket.PPPDisconnect are now deprecated. This is because they don’t make much sense. A PPP connection (point to point protocol) is a dial-up connection that must occur before a TCP/IP 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 calling Socket1.PPPConnect or Socket2.PPPDisconnect will affect ALL sockets on the system. In fact, saying Socket.PPPDisconnect will affect 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 use 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, which can be used identically to the old Socket class. We strongly recommend that you use TCPSocket in newer versions of REALbasic. The only good reason to use Socket instead is if you need backwards compatibility with REALbasic 4.5 and earlier (though the new socket features will obviously not be available in the earlier versions).

New Features and Nifty Ideas for REALbasic 5.0:

* 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 is a Mac-only improvement (since on Windows and Linux, the send speeds have never been bound to the event handling loop).

* 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 is a Mac-only improvement; 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; it 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 final 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 to 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 has been fixed, and the socket will now return the correct IP address of the remote connection. Now, on either platform, you can 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 (REALbasic Pro only) 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 want 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 roll back to 2 if needed
2 - SSLv3	SSL version 3
3 - TLSv1	TLS (Transport Layer Security) version 1

The default protocol is SSLv23, which is compatible with most SSL servers. You must set this property BEFORE you call 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 users to choose which phonebook entry they want to connect with. On all other Windows platforms, these dialogs are not available. Therefore, REALbasic will attempt a dialup connection using the default phonebook entry, including the username and password supplied. If either of these fields is 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 that gives you the choice to allow user intervention during the dial-up process, or whether you want it to run by itself. Some users might have more than one ISP, and the default might not be the one 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; see "Things to Watch Out For" for more information.

* ServerSocket has been added (REALbasic Pro only). This is a totally new concept to REALbasic, and one that was impossible to achieve using the old 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 almost 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 dropped connections. This bug has been fixed for the Macintosh. It 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. See also 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 was sent to you. When calling UDPSocket.Write (assuming you pass in a Datagram), it should contain the data you want to send, and IP address of the remote machine. This IP address can also be a multicast or broadcast address.

New Features and Nifty Ideas for REALbasic 5.5:

* Getting the system to pick a port for you. There are some instances where you'd like the system to pick a port for you. These needs can range from needing a port for passive FTP file transfers, or perhaps you wrote a class to auto-discover other applications on the network and you'd like to negotiate a port to connect over TCP on. Well, now you can have REALbasic pick the port for you. If you specify a SocketCore.Port = 0 and then call TCPSocket.Listen or UDPSocket.Connect, we will pick a random, open port for you to connect on. Most often, these ports will be in the range 49512 to 65535 (inclusive).

* Determining what port you're really bound to. If you used the above method for obtaining an open port, then you'll probably need to use this method to determine which port you're actually bound to. After calling SocketCore.Connect or TCPSocket.Listen, you can check the SocketCore.Port property to see which port was assigned. This means that the port number will change from 0 to the port number we are actually bound to. There's another benefit to this duality to the Port property. There is a hacking technique called port hijacking where the hacker steals a port out from under you. If this is the case, checking the SocketCore.Port property will tell you if someone has hijacked the port out from under you. It can be a good idea (though paranoid) to periodically check to make sure the .Port property lists a port that you expect to see. For instance, if you were listening on port 80 for HTTP connections, but the .Port property says you're listening on port 2113, then something might be wrong.

* Multiple interface support. New with version 5.5 is multiple interface support for Mac OS X, Windows and Linux. This allows you to write applications that can bind to different NIC cards installed on a user's machine. You can use this to write tunneling applications, and various other things you couldn't previously do in REALbasic. To see what interfaces are installed on the user's machine, you can use the System.GetNetworkInterface method, and assign the obtained interface object to a SocketCore.NetworkInterface property. For more information on the type of information available, please see the NetworkInterface API section.

* New networking helper classes. Have you ever wanted to write your own networked application, but have just been stuck in how to get it started? How about trying to write applications that self-discover one another on a local network? All of this (and more) have been made trivially easy now. The basis behind the new networking classes in RB is to provide you with easy access to a basic protocol which allows you to send and receive arbitrary messages. This may sound complex at first, but it really isn't. First, let's define what a "message" is. A message is made up of two pieces of information. One is an integer representing a command, and the other is a string of data. This command could be anything you'd like it to be. It is there to allow you an easy way to identify what type of information is in the data string. For example, you could send a command ID of 100 to mean that the string data is actually a

memory block containing a FolderItem. Or you could have ID 101 mean that the string contains the username of a remote application. This message mode is enforced on you in that you cannot do an arbitrary .Write command. If you'd like to send arbitrary data, then you can just make up a "misc command ID" and send your arbitrary data. When you receive data in, you are no longer given a DataAvailable event. Instead, we only pass whole messages to you with the ReceivedMessage event. Because of this, we do not allow you to read in arbitrary data using the .Read or .ReadAll methods.

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 wanted it 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 its connection. For example, some web servers will not release a socket once the data has been transferred to it (for efficiency reasons). So once you are done with the socket, unless you explicitly call SocketCore.Close, it will remain active and connected. 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 SocketCore.Close to terminate the connection. (Note that calling TCPCore.Disconnect also applies instead of calls to SocketCore.Close).

* Once your socket has received an Error event, it has been closed. The connection has been torn down and the internal send 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 its send buffer, and is ready to be used again (without calling SocketCore.Close or TCPSocket.Disconnect). The information that is retained is: the socket's port, address (in the case of TCPSockets) and LastErrorCode properties, as well as any data left in the socket's receive buffer. If you attempt to call Connect or Listen on the socket, the internal receive buffer will then be destroyed, so your socket can start over afresh.

* Calling SocketCore.Close tears down the socket, and will close any connections the socket might have. The information that is retained is: the socket's port, address (in the case of TCPSockets) and LastErrorCode properties, as well as any data left in the socket's receive buffer. If you attempt to call Connect or Listen on the socket, the internal receive buffer will then be destroyed so your socket can start over afresh. This also applies to calls to TCPCore.Disconnect

* 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.

* Beware of endian-ness! There are two different endian standards. On Macintosh systems you have big endian-ness, and on Windows and x86 Linux you have little endian-ness. Sockets work with streams of data and do not muck with the endian-ness of the data you are transferring. This is fine in many cases if you are transferring strings from one platform to another (like from Mac OS X to Windows XP). But if you are transferring binary data (such as from a BinaryStream or a MemoryBlock), you will want to ensure that the endian-ness matches from server to client regardless of what platform you are on. You can do this by setting the "LittleEndian" flag on MemoryBlocks or BinaryStreams to the same value for your client and your server. Failure to ensure that the endian-ness is consistent will result in a possible byte-order conflict in your application.

* Setting the socket's port after connecting. Once you have called SocketCore.Connect or TCPSocket.Listen, setting the socket's port will do nothing. This isn't a problem unless you are trying to set then check the socket's port while you're connected. For example, let's pretend your socket is currently connected to www.google.com and you do the following:

```
TCPSocket1.Port = Val( EditField1.Text )
if TCPSocket1.Port = 0 then
  MsgBox "Please enter a valid port"
else
  MsgBox "Thanks!"
end
```

If the socket is currently bound when you do this, you will never see the "Thanks" message, even if the EditField contains a valid port. This is because of the new feature (in REALbasic 5.5) that allows you to check which port you're currently bound to. Since you are connected, the test for .Port = 0 will fail because you're already bound to some (non-zero) port. However, if your socket isn't bound to any local port (you haven't called .Connect or .Listen yet, or the socket is Closed), then it will work just fine because we will just report back whatever is stored there. This probably won't affect many of you, but it's certainly something to watch out for!

Miscellaneous Tidbits:

* TCPSocket.Address is used only to specify the address 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 REALbasic will, upon connection, set TCPSocket.RemoteAddress 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. 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.

* TCPSocket.DataAvailable events will only fire once control has been given back to REALbasic's internals. This means that if you have code executing in a tight loop somewhere, your TCPSocket's DataAvailable event won't get the chance to fire until your code is done executing, or you call SocketCore.Poll. The DataAvailable event will fire once per chunk of data received. This means that if you do not read all the data from the buffer, we won't fire another DataAvailable event until new data arrives. Also, the DataAvailable event is not reentrant, so you will never have to worry about getting a new DataAvailable event while processing the current one.

* UDPSocket.DataAvailable events will fire when at least one new packet has arrived in REALbasic's internal packet queue. It is possible (likely even), that more than one packet will have arrived. We only give you one DataAvailable event for a discrete set of packets. This means that you will need to loop over the PacketsAvailable property of the UDPSocket in order to read all of the packets in. If any packets come in while you are processing a DataAvailable event, we will fire the DataAvailable event for you again.

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 (e.g. 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.

Picking a port to listen on is not always well defined. If you are implementing a well-known protocol, such as writing an FTP program, then you know you will need to support listening on port 21. But if you are writing your own protocol for your application, then how do you know what port to use? I suggest picking a port at random (at design time, NOT at runtime), and then checking to see whether that port is registered by another application. You can check this at

http://www.iana.org/assignments/port-numbers

If the number you have chosen is registered by another application, you should choose a different number.

Due to the amount of error checking, and handshakes, TCP is very reliable. 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 its destination. This overhead makes TCP a slower protocol; it gives up speed to gain security. If it is speed you are looking for (e.g. 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 it is not as secure as TCP. To use a UDP socket, since there is no connection, you do not need to take nearly as many steps to prepare.

A UDP socket 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 is bound to.

It also allows you to send data out, as well as to set UDP socket options (which will be described later). To tell 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 that sent you the data, and the 'payload'-- 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; please read further), the port it should be sent to, and the data you want to send the remote machine.

UDP sockets can operate in various modes, which are all very similar, but have vastly different uses. The mode that 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 to some network address. It is a Datagram that has one intended receiver.

The second mode of operation is called 'broadcasting'. As the name implies, this is akin to yelling into a megaphone. Everyone gets the message, 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.

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 of the UDPSocket class that 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 chat room, and are able to hold conversations with everyone else in the chat room. When you want to enter the chat room, you call JoinMulticastGroup, and you specify the group you want to join. The group parameter is a special kind of IP address, called a 'Class D IP'. It can range from 224.0.0.0 to 239.255.255.255. Think of the IP address as the name of the chat room. 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 want to leave the chat room, you just need to call LeaveMulticastGroup, and again, specify the group you want to leave. You can join as many multicast groups as you like; you are not limited to just one at a time. When you want to send data to the multicast group, you just need to specify the multicast group's IP address. Everyone that has joined the same group as you will receive the message.

Multicasting has some extra features that make it an even more powerful utility for network applications. If you want to receive the multicast data you sent (known as "loopback"), set the SendToSelf property on the socket. If it is 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 (known as the "Time to Live", or TTL). When your datagram gets sent out, it runs thru a series of routers on the way to its destinations. Every time the datagram hits a router, its RouterHops property is 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. Most routers do not support multicast packet forwarding, and so, as a general rule, a multicast will never escape your local network. Therefore, multicasting can be great in a large internal network (that spans many routers and switches), but it probably will not work for you if you are trying to write Internet applications.

The connectionless functionality of UDP makes no guarantee that your data will reach its destination. You can work around this by creating your own protocol, on top of the UDP protocol, that acknowledges receives.

I want to go into a little more detail about Class D IP addresses, since they seem to confuse many users. A Class D IP address is a specially reserved IP that no "real" machine can have. So you do not have to worry that your local machine's IP address is not Class D, and similarly, you do not have to worry about IP collisions on your network. These IPs are used by the network transport layer to determine how to efficiently to send a packet. When broadcasting, the transport will simply blast the packet out to every computer on the network. But when multicasting, the transport will determine which machines are connected to the group, and it will only send the packets to those machines. This will cut down on network traffic for "chatty" protocols, to a large degree. As of this writing, there is no database of Class D IPs and which applications use them (like there are for registered ports), and so picking an IP can be somewhat hard to do. A general rule of thumb is to pick a random-looking IP in the Class D range. If you run into collisions, it's usually pretty trivial to change the address that the socket tries to multicast to, and collisions are fairly rare.

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 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!

Easy Networking Classes Explained:

EasyTCPSocket:

This class allows you to establish connections and communicate via the messaging protocol with a remote machine. The main difference between the EasyTCPSocket class and a regular TCPSocket class is the message-based aspects of the protocol. The connection process is identical to a regular TCPSocket. However, when you want to send data to a remote machine you must use the SendMessage method to do so. If you are on the receiving end of a message, you will get the ReceivedMessage event. For your synchronous listening needs, there is a WaitForConnection method which will synchronously wait for a predetermined amount of time for a connection to be established. If the connection is made, it returns true, otherwise, it returns false. Note that we call App.DoEvents internally so that your application's UI will stay responsive during this call. Also for your synchronous needs, we added the WaitForMessage method. This method will wait for a message to come in with the command ID you specify. Once that message comes in, we will return the string data portion of that message. If a message comes in with a command ID that is different from the one you are expecting, we will drop that message. Note that this method internally calls App.DoEvents, so your UI will stay responsive.

EasyUDPSocket:

This class allows you to easily communicate with the UDP protocol using unicasting, multicasting or broadcasting. Like the EasyTCPSocket class, this class is based around the simple messaging protocol described above. You can send message to either an individual, or to an entire group. One main benefit to the EasyUDPSocket class is the way we handle multicasting. Everyone has a hard time remembering what IP addresses can be used with multicasting. It's a bother to try to look up what a "Class D" IP address is. So we made it simple on you: you don't have to use one if you don't want to. You can pass any string you would like as a groupName parameter and we will change it into a proper multicasting IP address for you. This means that you can pass "My Awesome Application" in as the parameter to Register and use that same string for calls to SendMessageToGroup. You can still pass in a valid Class D IP address; we will honor them. One other helper function is the Bind function. This sets up the socket for your properly (so you don't have to set .RouterHops or .SendToSelf up yourself) and does the bind on the port specified. One thing to keep in mind is that we default to having .SendToSelf on. You are welcome to override this default yourself by setting SendToSelf to false after the making the .Bind call. While you can still call [Join/Leave]MulticastGroup, we suggest that you only use Register and Unregister for your application.

AutoDiscovery:

The AutoDiscovery class does exactly as the name implies; it lets you automatically discover other applications on the local network. It does so by checking to see what other applications are using the same group name that you pass in to the Register function. When a member joins (this includes your application when you first call .Register), you will get a MemberJoined event with the IP address of the member that joined. When a member leaves, then you get a MemberLeft event with their IP as well. If you would like a list of the currently connected members, you can get an array of their IPs back by calling GetMemberList. If you're worried about your member list getting stagnant (due to computer's crashing, etc), you can always call the UpdateMemberList method which will clear the internal list of connected members and re-query the network for members.

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.

- 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 or later.
- 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 forcibly, by a user pulling the Ethernet cable out of his or her computer), or gracefully (by calling SocketCore.Close). This might or might 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 the Ethernet cable got pulled, then it really is an error, but the result is the same: the connection was lost. You will also get this error if you call TCPSocket.Disconnect.

- 103 -- You will get this error if RB cannot resolve the address you specified. A prime example of this would be a mistyped IP address, or the 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 or later.
- 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. Example: calling 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.
- 108 -- This error means that you've run out of memory. We try to provide you with this error when the OS or underlying transport provider let us know that an operation could not be completed due to a lack of memory. Chances are, you will never encounter this error.

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 Classic, these error codes are negative numbers in the range [-3211, -3285]. For Windows, Mac OS X and Linux, these error codes are usually positive numbers in the range [1, 1000]. For a description of the Macintosh Classic error codes, find a copy of MacErrors.h, for Windows error codes, find a copy of WinSock.h (and add 10000 to the error code provided by REALbasic), and for Mac OS X or Linux, check out errno.h for error codes.

PPP Status Codes Explained:

When calling System.PPPStatus, there are a number of codes returned to you that have thus far 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 System.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 might 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)
- NetworkInterface as NetworkInterface -- specifies which network interface the socket should use when binding. Leaving this property as nil will use the currently selected interface.

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 properties 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*)
- BytesAvailable as Integer -- tells you how many bytes of data are available in the internal receive buffer (*Read Only*)
- BytesLeftToSend as Integer -- tells you how many bytes of data are left to send. This allows you to write a synchronous socket without implementing any events. (*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, [enc as TextEncoding]) -- reads the amount of bytes specified from the internal receive buffer. Optionally, if you pass in a non-nil enc, we will set the returned string's encoding property.
- ReadAll([enc as TextEncoding]) -- reads all the data from the internal buffer. Optionally, if you pass in a non-nil enc, we will set the returned string's encoding property.
- Write(data as String) -- writes out the string to the remote connection
- Disconnect() -- disconnects the socket, resets it, and fires a socket 102 error.
- Lookahead() as String -- shows the data that is available in the internal queue without removing it
- 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.

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

EasyTCPSocket (inherits from TCPSocket):

Methods:

- SendMessage(command as Integer, data as String) -- Constructs a message and sends it to the remotely connected side. Note that commands should be greater than (or equal to) 0.
- WaitForConnection(timeout as Integer) as Boolean -- Initiates a listen and waits for a connection to come in. If the connection is established before the operation times out, then this method returns true (otherwise it returns false).
- WaitForMessage(command as Integer) as String -- Waits indefinitely for the specified command to come in. When the command arrives, we return the data portion of the message. Note that if any commands come in that are not the ones we are waiting for, they are dropped.

Properties:

- None

Events:

- Error(code as Integer) -- Replaces the TCPSocket error event. Functions the same, but instead this event passes in the error code so you don't have to check the LastErrorCode property.
- ReceivedMessage(command as Integer, data as String) -- Fires when a new, complete message has arrived.

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([enc as TextEncoding]) 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. Optionally, if you pass in a non-nil enc, we will set the returned string's encoding property.

Events:
- None.

EasyUDPSocket (inherits from UDPSocket):
Properties:
- None

Methods:
- Bind(port as Integer) -- Binds the EasyUDPSocket to the port specified. This is a helper function so you don't have to set .Port and call .Connect.
- Register(groupName as String)-- Registers your class so it can receive multicasts. groupName can be any string. We will map it to a valid Class D IP address. For example, you can say .Register("Aaron's Cool App") and we will hash that to a valid Class D IP address. However, if you use a valid Class D IP address, we will use that without modifications.
- Unregister(groupName as String) -- Unregisters your class from the group which you had previously registered yourself with.
- SendMessageToGroup(groupName as String, command as Integer, data as String) -- Constructs and sends a message to the group you registered with.
- SendMessageToIndividual(ip as String, command as Integer, data as String) -- Constructs and sends a message to the individual you specify.
- WaitForMessage(command as Integer, ByRef fromIP as String) as String -- Waits for the command specified to be received. Will pass you back the IP address which sent the message by reference and return the message from the method call.

Events:
- Error(code as Integer) -- Replaces the TCPSocket error event. Functions the same, but instead this event passes in the error code so you don't have to check the LastErrorCode property.
- ReceivedMessage(fromIP as String, command as Integer, data as String) -- Fires when a new, complete message has arrived.

AutoDiscovery (inherits from EasyUDPSocket):
Properties:
- None

Methods:
- GetMemberList() as String() -- Gets an array of the members we currently know about in our group.
- SendMessageToGroup(command as Integer, data as String) -- Sends a message to the group we currently belong to. This is a helper function so you don't need to keep track of the group you belong to.
- UpdateMemberList() -- Removes all members from our cached list and re-queries the group to see who's still active.
-

Events:
- MemberJoined(ip as String) -- Fires when a member joins our group
- MemberLeft(ip as String) -- Fires when a member leaves our group

Datagram:
Properties:
- Address as String -- specifies an IP address. It is the address of the remote machine that 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 its 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 its internal pool. Will get called when the ServerSocket first begins listening, and be called as many times as it takes to entirely fill the internal socket pool up (up to ServerSocket.MaximumSocketsConnected times).
- 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*)

System.NetworkInterface Functions:
Methods:
- System.NetworkInterfaceCount() as Integer -- Returns a count of the number of interfaces currently installed on the system. Note that the loopback device may or may not be listed depending upon the version of your operating system. For example, on Windows, the loopback interface will not be listed, but on OS X, it may be listed.

- System.GetNetworkInterface(index as Integer) as NetworkInterface -- Returns a NetworkInterface object corresponding to the index you passed in. This index is 0-based.

NetworkInterface:
Properties:
- MACAddress as String -- Holds the MAC address for the NIC card
- IPAddress as String -- Holds the IP address currently assigned to that interface
- SubnetMask as String -- Holds the subnet mask (also called the netmask) for that interface

Methods:
- None

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
Jan 28 2003 -- AJB -- Updated some of the documentation to rectify confusing parts.
Apr 04 2003 -- AJB -- Updated information, and added new caveats.
May 02 2003 -- AJB -- Added DataAvailable information
Jun 17 2003 -- AJB -- Added warning about endian-ness
Dec 22 2003 -- AJB -- Updated to make the document 5.5 compliant
Jan 02 2004 -- AJB -- Added another "gotcha" to watch out for with getting the port.
Jan 23 2004 -- AJB -- Updated the NetworkInterface information