QSpice C-Block Components

A “Berkeley Sockets API” Reference

Caveats & Cautions

First and foremost: Do not assume that anything I say is true until you test it yourself. In particular, I have no insider information about QSpice. I’ve not seen the source code, don’t pretend to understand all of the intricacies of the simulator, or, well, anything. Trust me at your own risk.

If I’m wrong, please let me know. If you think this is valuable information, let me know. (As Dave Plummer, ex-Microsoft developer and YouTube celebrity (Dave's Garage) says, “I’m just in this for the likes and subs.”)

Note: This is the tenth document in a clearly unplanned series:

- ***The first C-Block Basics document covered how the QSpice DLL interface works.***
- ***The second C-Block Basics document demonstrated techniques to manage multiple schematic component instances, multi-step simulations, and shared resources.***
- ***The third C-Block Basics document built on the second to demonstrate a technique to execute component code at the beginning and end of simulations.***
- ***The fourth C-Block Basics document revisited the Trunc() function and provided a basic “canonical” use example.***
- ***The fifth C-Block Basics document revisited the MaxExtStepSize() function with a focus on generating reliable clock signals from within the component.***
- ***The sixth C-Block Basics document examined connecting and using QSpice bus wires with components.***
- ***The seventh C-Block Basics document described recent QSpice changes (a new Display() function changes to Trunc()).***
- ***The eighth C-Block Basics document detailed post-processing techniques and much more in painful detail.***
- ***The ninth C-Block Basics document explained how to use the QSpice EngAtof() function.***

This paper provides information about QSpice “Berkeley Sockets” client/server component framework. It assumes that you already have a good understanding of creating C-Block Components. See my GitHub repository here for prior documents in this series.

Background

In August of 2025, QSpice added a client/server component framework that it calls “The Berkeley Sockets API.” At a high level, you can now easily create components that use a separate server process to do simulation computations. The template generator produces a client component (C++ code) and server code (C++ or Python). You can even write your own server code in other languages that support sockets. (Java should be easy.)

Note: The framework went through a number of revisions over a month or so. This document is based on the 2025.10.02 release. It may not perfectly match any subsequent QSpice revisions.

Overview

This document contains reference information for the new QSpice client/server features. It covers:

- How the client/server code relates to a basic "monolithic" component (i.e., non-client/server C++ DLL component).
- Notes for creating basic client/server code using the template generator.
- Details of the QSpice "Berkeley Sockets API" (which I'll call the "QSocks API").
- Client & server data packet message contents.
- Functional limitations of template-generated client/server vs monolithic components.

This is *not* a primmer about writing components. If you're not already familiar with QSpice C++ DLL components, you should review prior documents in the C-Block Basics series first.

Also, this is not a primmer on client/server concepts. While you don't need an in-depth knowledge of Berkeley Sockets or networking principles, I assume that you have basic concepts of clients and servers and the idea of sending/receiving messages between client/server processes.

Finally, this document doesn't specifically address Python servers. The concepts will apply but you'll have to write your own code.

Basic Concepts & Terminology

The ***Berkeley Sockets API*** is a formal standard that allows two application programs to communicate across a network. Windows implements the Berkeley API as "*Windows Sockets*" or "*Winsocks*." The QSpice template ***server code*** (C++ version) uses Winsocks directly so you'll see raw Winsocks API calls there. (You can find Microsoft's Windows Sockets documentation here.)

If you search QSpice Help for "Berkeley Sockets API," you'll find documentation for five new callback functions. These functions are *QSpice-specific* wrappers around Winsocks API calls which add useful functionality for *client DLL code*. To make a clear distinction, I'll call these five functions the "*QSocks API*."

Further, the code generated by the template generator passes data between the client and server in a *QSpice-specific* way. I'll refer to the combination of QSocks API and data packet contents convention as the "QSpice Client/Server Framework" or "the QSocks framework" or similar.

Monolithic vs Client/Server Components

The QSpice simulator (`QSpice64.exe`/`QSpice80.exe`) doesn't "know" whether a schematic component is a traditional monolithic component or a client/server version. In both cases, the simulator loads a DLL and calls exactly the same DLL component entry points:

- The required evaluation function. (I'll call this "`EvalFunc()`" going forward.)
- An optional `MaxExtStepSize()` function.
- An optional `Trunc()` function.
- Optional "ancillary functions" such as `Destroy()` and `PostProcess()`.

In the client/server framework, the DLL contains the client code. For the *first three* of the above entry points, the client packages some of the function parameters and sends them to the server program. The server returns data to the client DLL which then returns values to the simulator. The ancillary functions are not supported in the template-generated client/server code.[1]

In simple client/server components, all of the component logic – including any required per-instance data – is contained in the server code. In theory, no modifications of the client code are required unless you need unsupported functionality. (See Functional Limitations of Template Code below.)

To get a feel for how the client/server framework works, spend a few minutes reviewing the QSpice `SocketAPI.qsch` demo and the associated client/server `*.cpp` files.

Creating Client/Server Components

Here are very abbreviated instructions for creating a client/server project or modifying an existing component.

Create A Basic Component

The first steps to create client/server code are the same as for a basic C++ DLL component:

- Create a hierarchical component block in the schematic.
- In Symbol Properties, change the Symbol Type to "`Ø(.dll)`".
- Edit the first string attribute to the desired name of the client DLL.
- Add and configure input and output ports. For client/server code, these must be of data type `float` (64-bit double).
- Add any required string attributes. (The framework doesn't support passing these additional attributes to the server. See Functional Limitations of Template Code below.)

Add Client/Server Elements

Now things change. Before generating the client/server templates, we need to add a special string attribute of the following form:

```
char *server=Host_Address:Port_Number/Server_Name
```

For now, just use one of the following and replace "`MyServer`" with the desired server file name:

```
char *server=localhost:1024/MyServer.exe          (to generate myserver.cpp)
char* server=localhost:1024/MyServer.py           (to generate myserver.py)
```

1 Technically, the Destroy() function is implemented in the client code to shut down the connection. However, no message is sent to the server so the server sees only that the connection was dropped.

Note that the value of `server` is not quoted. The template generator doesn't like quotes. You can add quotes after generating the initial code if, say, you need the server program name to contain a path with spaces. (I've not tested paths/spaces yet. Just keeping it simple for now.)

Now we can generate client and server template code. Right-click the component block to bring up the code generator options. The QSpice code template generator now has "Berkeley Socket API" options to create:

- C++ Client DLL
- C++ Server Template
- Python Server Template

The C++ Client DLL is required and is identical for both C++ and Python servers. Create the client code.

Then create the desired server template code. (You can generate both server versions if you choose.)

Compile the C++ client and C++ server code in the usual way (right-click, Compile in the code editor).

The Python server runs under the Python interpreter and does not require a separate compilation step. (You must have Python installed and in the program search path.)

At that point, you should be able to run a simulation without errors. However, the Waveform Viewer window contents will display rubbish until you add some code in the `Evaluate()` server function.

The QSocks API

The QSocks API contains five functions with external linkage (`extern "C" declspec(dllexport)`). From the client code's standpoint, we can simply think of them as:

```
•   unsigned int BerkeleySocket(const char *hostPortServiceString, int Ninputs, int Noutputs,
                   unsigned char **buffer);
•   double      *ConfigureBuffer(unsigned char *buffer, int code);
•   int          SocketSend(unsigned int socket, unsigned char *buffer, int count);
•   int          SocketRecv(unsigned int socket, unsigned char *buffer, int count);
•   void         SocketClose(unsigned int socket);
```

Note: These calls are used only in the client DLL code. They call code in the simulator (QSpice64.exe or QSpice80.exe) and are not available in server program code.

BerkeleySocket()

The `BerkeleySocket()` function is the most complicated of the QSocks API calls. It wraps a number of Winsocks and other WinAPI calls into a single function which:

- Allocates a packet buffer of "appropriate size" depending upon the number of component input and output ports. ("Input attributes" are not included in the calculation.)
- Optionally attempts to start a specified server on a specified port.
- Attempts to connect to the server.

Function Signature:

```
unsigned int BerkeleySocket(const char *hostPortServiceString, int Ninputs, int Noutputs,
   unsigned char **buffer);
```

Parameters:

Parameter	**Data Type**	**Description**
`hostPortServiceString`	`const char*`	Pointer to a string specifying the host address or name, port number (optional), and server program name (optional). If the port number is omitted, QSpice starts with 1024 and increments the port number for each additional component instance. If server program name is omitted, QSpice does not attempt to start the server – it must be running on the host/port before starting the simlation or the connection will fail.
`Ninputs`	`int`	Number of component input ports.
`Noutputs`	`int`	Number of component output ports.
`buffer`	`unsigned char**`	A pointer to receive the address of an allocated packet buffer.

Returns:

On Success	A socket number for the server connection.
On Failure	No return on failure; QSpice terminates with an error message in the simulation Output window.

Notes:

In the QSpice generated client code, `BerkeleySocket()` is called in the initialization section of `EvalFunc()`. `Ninputs/Noutputs` are set by the template generator to match the number of component inputs/outputs. The pointer to the allocated buffer is stored in the per-instance `buffer` member. The socket number is stored in the per-instance `ConnectSocket` member.

Client-side per-instance data is used by the QSpice framework (as above). Your component's per-instance data should be stored in the server-side code.

The size of the allocated buffer is computed as `MAX(4 + 8 * (Ninputs + 2), 8 * Noutputs)` as required for the largest message type (`PORT_TRUNCATE`).

The client code doesn't have an opportunity to assess error conditions.

ConfigureBuffer()

The `ConfigureBuffer()` function initializes a client outgoing packet buffer with a ***packet type***. It returns a pointer to the buffer position where additional data can be added. The pointer is of type `*double` or, equivalently, `double[]`.

Function Signature:

```
double *ConfigureBuffer(unsigned char *buffer, int code);
```

Parameters:

Parameter	**Data Type**	**Description**
`buffer`	`unsigned char*`	Pointer to a data packet buffer.
`code`	`int`	An integer representing the ***packet type***.

Returns:

On Success	A pointer to the position in the buffer following the packet type data element (*buffer + 4). That is, the pointer is the start of the array of component input port values.
On Failure	Writes a message to the QSpice Output window and aborts the simulation.

Notes:

See Data Packet Contents for valid pre-defined packet types.

SocketSend()

The `SocketSend()` function is a thin wrapper for the Winsocks `send()` function.

Function Signature:

```
int SocketSend(unsigned int socket, unsigned char *buffer, int count);
```

Parameters:

Parameter	**Data Type**	**Description**
`socket`	`unsigned int`	A socket ID for the server connection. The connection must have been successfully opened prior to this call.
`buffer`	`unsigned char*`	Pointer to the first byte of the buffer containing data to send, i.e., the first byte of the packet type.
`count`	`int`	Number of bytes to send in the data packet.

Returns:

On Success	The number of bytes sent. This can be less than `count`.
On Failure	Writes a message to the QSpice Output window and aborts the simulation.

Notes:

This is a non-blocking call (returns immediately).

The client code doesn’t have an opportunity to assess error conditions.

SocketRecv()

The `SocketRecv()` function is a thin wrapper for the Winsocks `recv()` function.

Function Signature:

```
int SocketRecv(unsigned int socket, unsigned char *buffer, int count);
```

Parameters:

Parameter	**Data Type**	**Description**
`socket`	`unsigned int`	A socket ID for the server connection. The connection must have been successfully opened prior to this call.
`buffer`	`unsigned char*`	Pointer to the first byte of the buffer to receive the packet data.
`count`	`int`	Maximum number of bytes to receive. Must not exceed the size of `buffer`.

Returns:

On Success	The number of bytes received. This can be less than `count`.
On Failure	Writes a message to the QSpice Output window and aborts the simulation.

Notes:

This is a blocking call, i.e., it does not return until the server responds (or the connection is dropped or some other failure is detected).

The client code doesn't have an opportunity to assess error conditions.

SocketClose()

The `SocketClose()` function is a thin wrapper for the Winsocks `closesocket()` function.

Function Signature:

```
void SocketClose(unsigned int socket);
```

Parameters:

Parameter	**Data Type**	**Description**
`socket`	`unsigned int`	A socket ID for the server connection. The connection must have been successfully opened prior to this call.

Returns:

N/A. If data cannot be received (e.g., invalid socket, no active connection to server), QSpice writes a message to the simulation Output window and terminates the simulation.

Notes:

`SocketClose()` closes the socket connection and frees resources. The QSpice framework appears to rely on the server closing/exiting when the client connection is dropped.

Unfortunately, the server code cannot distinguish between a dropped connection and a deliberate end-of-simulation-step event. (See Functional Limitations of Template Code below.)

Data Packet Contents

The QSpice framework defines four types of outgoing data packets (sent from client to server): `PORT_INFORMATION`, `PORT_EVALUATE`, `PORT_MAX_STEPSIZE`, and `PORT_TRUNCATE`. These packets contain a 4-byte integer (the packet message type) followed by message-specific data.

Incoming data packets (from server to client) contain one or more 8-byte double values. There is no packet type identifier in the server response packets.

Here is a summary of the packet message types based on examination of the generated C++ client/server code. See The QSocks API for QSpice framework functions that use these data packets.

Note: None of the client-side functions pass per-instance data to the server. The client-side per-instance data is used solely by the QSpice framework so don't mess with that unless you understand what you're doing. Put all per-instance data in the server code as global data variables. (Each component instance is a separate server process so the global data is unique to the instance.)

PORT_EVALUATE (Type ID = 2)

Description:

This packet type is used in the client code evaluation function (`EvalFunc()`). It sends the "`t`" (time-point) parameter and the input port values.

Packet Contents:

1. Packet Type ID: 4-bytes integer (value = 2)
2. Time-point ("`t`"): 8-byte double
3. Input port values: An array of one or more 8-byte double(s)

Server Returns:

An array of one or more calculated output port values (8-byte doubles).

Notes:

The template client code does not send "input attribute" values, only input port values.

PORT_MAX_STEPSIZE (Type ID = 3)

Description:

This packet type is used in the client code `MaxExtStepSize()` function. It sends the "`t`" parameter (time-point) value.

Packet Contents:

1. Packet Type ID: 4-bytes integer (value = 3)
2. Time-point ("`t`"): 8-byte double

Server Returns:

The desired maximum step size (a positive 8-byte double). A return value of `1e308` indicates that no time-step reduction is desired.

Notes:

None.

PORT_TRUNCATE (Type ID = 4)

Description:

This packet type is used in the client code `Trunc()` function. It sends the "`t`" parameter (hypothetical time-point) and input port values.

Packet Contents:

1. Packet Type ID: 4-bytes integer (value = 4)
2. Time-point ("`t`"): 8-byte double
3. Input port values: An array of one or more 8-byte double(s)
4. Followed by the `Trunc(…, *timestep)` parameter (the "otherwise planned time-step")

Server Returns:

The desired time-step value (a positive 8-byte double). A return value of `1e308` indicates that no time-step reduction is desired.

Notes:

The template client code does not send "input attribute" values, only input port values.

Functional Limitations of Template Code

The client/server code created with the QSpice Berkeley Sockets API template generator does not support the full functionality of monolithic C++ DLL components. The limitations include:

- Component input/output port data types must be of type 64-bit double. (This isn't really a problem; just change port types to double.)
- Component String Attributes ("input attributes") are not passed to the server and passing arbitrary data to the server isn't supported.
- Most API "`extern "C" __declspec(export)`" callbacks and variables available to C++ DLL components (e.g., `*StepNumber`, `*NumberSteps`, `*InstanceName`) are not generated in the client code. These can be added manually but, again, there's no provision for sending arbitrary data to the server.
- The `Destroy()` function (called at the end of each simulation step) simply drops the connection to the server. The server closes "ungracefully" without an opportunity to perform any needed end-of-step cleanup (e.g., writing final entries to and properly closing log files).

- The `PostProcess()` function (called at the end of simulation immediately before `Destroy()`) is not included.
- The template code generator requires at least one input and one output port. This requirement means that you cannot create a “timer component” that has no input port nor a “logging component” that has no output port.

All of the above can be done but require modification to both client and server code. That will be the subject of a future paper.

Conclusion

I hope that you find the information useful. Please let me know if you find problems or suggestions for improving the code or this documentation.

--robert