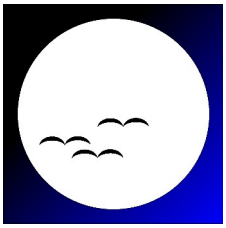

QSpice C-Block Components

Extending The “Berkeley Sockets API”

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.***
- ***The tenth C-Block Basics document was a technical reference for the QSpice “Berkeley Sockets API.”***

This paper builds on the prior document and provides example code to overcome limitations in the QSpice “Berkeley Sockets API” 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.

Overview

The prior document in this series was a “technical reference” for the QSpice “Berkeley Sockets API” framework. It provided some terminology, API function details, message data packet descriptions, etc. The final section, “Functional Limitations of Template Code,” listed things that you can do fairly easily

in “monolithic” C++ DLL components but that aren’t supported in the code templates for client/server components.

This paper starts with a monolithic component to demonstrate the limitations. Then it provides sample client code (C++ only) and server code (C++, Python, and Java) to demonstrate techniques to work around the framework limitations.

The code examples are heavily commented and will serve as the primary documentation. The remainder of this paper will merely explain problems, describe solutions, and point you to relevant code.

The Monolithic Component

We’ll start with the basic non-client/server component to demonstrate the QSocks framework limitations: CB11_Monolithic.qsch & CB11_Monolithic.cpp.

The component code isn’t complicated. The evaluation function:

- Allocates per-instance memory.
- Sets a per-instance end-of-simulation flag (`InstData.postProcess`) to false.
- Opens a user-supplied logfile name using the `*StepNumber` value to determine whether to overwrite or append to the logfile.
- Stores a pointer to the component instance name (`*InstanceName`) in per-instance data for use in logging.[1]
- Calculates a new `OUT` port value from input ports `EN` (enabled) and `IN` (a sine signal) times the input parameter `Gain`.
- Writes log entries. It uses `*ForKeeps` to determine whether `EvalFunc()` was called from `Trunc()` or from QSpice directly.

Elsewhere, the code:

- Writes log entries.
- Sets an “end of simulation flag” in the per-instance `endSim` member (`PostProcess()`).
- Closes the logfile at the end of the simulation step (`Destroy()`).

In the logfile entries, we include the step number (`*StepNumber`) and instance name (from the per-instance `*instName` or `*InstanceName`).

Those seem like common/useful things to do and not really complicated, right? It should be easy to convert this to a client/server component, right? Yeah, I thought so, too. We’re both wrong. We’re going to need more code….

1 QSpice sets the *InstanceName value to point to an empty string when Trunc() is called. We save the “real” instance name pointer for logging during the Trunc() code execution. We could, of course, copy the instance name to per-instance data but this is less complicated.

The Client/Server Component

I've created a client/server versions of the schematic – CB11_SocketAPI_Cpp, CB11_SocketAPI_Py.qsch, and CB11_SocketAPI_Java.qsch – with necessary changes:

- Changed `EN` port type from `boolean` to `float (64-bit double)` since that's the only supported port type.
- Changed the component DLL name attribute to CB11_Client.
- Added the required server attribute (`server="localhost:1024/CB11_Server.exe", server="localhost:1024/CB11_Server.py", and server="localhost:1024/CB11_Server.java"`).

I then generated the client/server code (CB11_Client.cpp, CB11_Server.cpp, CB11_Server.py, CB11_Server.java) and added the necessary code to support the extended functionality.

Custom Messages

To add the needed features, we need a few categories of custom messages.

Set Value Messages

These messages are used to pass values to the server in the initialization portion of the client evaluation function.

Message ID	**Used To**
PORT_LOGNAME	Set the logfile name (a string value) for the component instance.
PORT_INSTNAME	Set the component instance name (a string value).
PORT_GAIN	Pass the component `Gain` attribute.
PORT_STEPNBR	Pass the `*StepNumber` extern variable.

These demonstrate passing ASCIIZ strings and integer values. Other types could be passed.

The server merely stores the passed values in "per-instance data." See PORT_INITIALIZE below.

Event Notification Messages

These messages notify the server that a client event occurred.

Message ID	**Used To**
PORT_DESTROY	The client `Destroy()` function was called (end of step).
PORT_POSTPROCESS	The client `PostProcess()` function was called (end of simulation, immediately before `Destroy()` is called).

The server contains `Destroy()` and `PostProcess()` handlers for user-defined processing of these events.

Server Command Messages

These messages instruct the server to perform a function.

Message ID	Used To
PORT_INITIALIZE	Instructs the server to complete any required initialization. Sent after the client evaluation function one-time initialization values have been sent.
PORT_CLOSESERVER	Instructs the server to stop listening for messages and terminate. Sent immediately before the client closes the socket connection in the client `Destroy()` function. Note: The server does not send a response to the client.

The server contains `Initialize()` and `CloseServer()` handlers for user-defined processing of these events.

Notes About Messages

"Handshakes"

Messages use a "send/receive handshake" to avoid message corruption. That is:

- The server sends a message only in response to receiving a message from the client.
- The server always sends a response when a message is received (except for the PORT_CLOSESERVER message).
- After sending a message, the client always waits to receive a server response (except for the PORT_CLOSESERVER message).

The template-generated server code does not contain a default case in the main loop switch so it will not detect failures to honor the above "handshake rule." The custom server code adds a default case to save you headaches if you fail to follow the rule or send an invalid message code.

Message Buffer

The message buffer must, of course, be large enough to contain the largest message to be sent/received.

If you generate the client code from the project example, you won't see a fixed buffer size. The buffer is allocated by the `BerkeleySocket()` call in the client initialization code based on the number of input and output ports. A pointer to the allocated buffer is stored in the per-instance data buffer member.

We could calculate the buffer size but it's easier to look at the template-generated server code. Our project example creates a 36-byte buffer. The first four bytes of a message are the message ID. That leaves 32 bytes for data.

Our project needs to pass user-specified strings of arbitrary length. Specifically, the logfile name/path might be longer than 32-bytes. I've arbitrarily decided to use a buffer size of 1024 bytes; this allows a name/path of up to 1019 characters[2]

Anyway, `BerkeleySocket()` doesn't make it easy to change the buffer size to a specific value and, because it starts up the server and establishes the socket connection, we can't just not call it. So, the

2 1024 less 4-byte message ID less 1-byte ASCIIZ terminator.

client code simply frees the allocated buffer, allocates a new buffer of the desired size, and replaces the buffer pointer in the per-instance buffer member.

ConfigureBuffer() vs Structs

The QSocks framework client code uses `ConfigureBuffer()` to initialize the message buffer with the message ID and return a pointer to the byte following the message ID. This pointer is returned as a "pointer to a double" or, equivalently, a pointer to an array of doubles. This is all that is needed by the framework because all message data elements are doubles.

We could recast the pointer to other types like this:

```
int *pInt = (int*) ConfigureBuffer(instData.buffer, msgID);
*pInt = 5; // set the data element to 5
```

That works but it's a bit opaque and, on the server side, we have don't have `ConfigureBuffer()` hiding the underlying buffer. So, I use structures for different message types. For example:

```
struct IntBuf {
  int msgID;
  int intVal;
};
struct StrBuf {
  int  msgID;
  char strVal[BUFFER_SIZE - sizeof(StrBuf::msgID)];
};
```

Now the client and server code simply casts the buffer to the appropriate type and accesses the message ID and data portion directly.

Conclusion

I've probably missed some important points but the code is heavily commented.

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