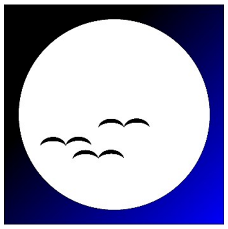

QSpice C-Block Components

“Berkeley Sockets API” Multi-Client Servers

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.”***
- ***The eleventh C-Block Basics document extended the “Berkeley Sockets API” framework to overcome the limitations of template-generated code.***

This paper builds on the prior two documents and provides basic multi-client host servers for C++, Python, & Java. It assumes that you have followed along with the earlier Berkeley Sockets API papers. See my GitHub repository here for prior documents in this series.

Overview

In the most recent two papers, we explored the QSpice “Berkeley Sockets API” and ways to extend the client/server template code to do, well, all of the stuff that a basic C/C++ DLL component can do.

If you want to create a server that runs on a separate host machine, there are some remaining issues:

- QSpice can launch servers only on the local machine.
- Each component instance creates a server process which opens a connection on a unique port.
- Each server process allows only a single client connection.
- Each server process terminates when the client disconnects.

What we need is a server that:

- Opens a single host port for client connections.
- Allows multiple clients to connect on that port.
- Launches a unique server thread for each client connection.
- Terminates only the client thread when the client disconnects.

This project implements such a client/server framework.

I'm not providing detailed explanations of the code. Other than the multi-client/multi-threading aspect, it's functionally quite similar to the code presented in C-Block Basics #11. I'll simply give some pointers and suggest potential improvements.

Compiling The Code

The client code can be compiled to a DLL using the DMC compiler included with QSpice. (See instructions in the code or open/compile from within QSpice.)

The C++ server code requires C++20 or better[1] so DMC won't work. I used Microsoft Visual Studio 2022 Community Edition (free). (See compile instructions in the code.)

The Python server code doesn't require compilation. The Java server has more options – run from *.java (which compiles on the fly), compile to class files (and run those), or run from a *.jar file. (See instructions in the code.)

Running The Examples

To run the examples, first start the server on the host machine with a port number argument. For example, start the C++ server with:

```
>CB12_HostServer.exe 666
[Server] Listening on port 666...
[Server] Press Ctrl+C to stop.
```

Port 666 is the assigned Doom multi-player server port, used here because I had to choose a port for example purposes. You can choose a different privileged port (i.e., < 1024).[2]

Open the schematic. If you're running the server on the local machine on port 666, it should work as is. For a different host or port, you'll need to change the server=”localhost:666/” component attributes.

1 If you remove the std::osyncstream code, you should be able to compile the server with C++11. It's there merely to prevent corruption/interleaving of std::cout messages from client threads. DMC does not support C++11 so you'll still need to use a modern compiler for the server code.

2 QSpice will not allow two component instances to connect to the same port unless the port # is less than 1024.

Change “localhost” to the IP address of the host. In theory, you can also use your host’s registered domain name (e.g., YourHost.com) but I’ve not verified that it works as advertised.[3] Change the port to whatever port you used to launch the server.

Run a simulation. Server messages are written to the host command prompt window. Client-side messages are written to the QSpice Output window.

Potential Improvements

I can think of a few improvements and enhancements….

Non-Windows Host Platforms

The example server code uses the Windows Sockets API. It would not be hard to port the C++ server code to other platforms such as Linux and macOS.

~~Alternatively, the server code could be rewritten in Java or Python. That would be pretty cool.~~

Python and Java server code has been added.

Multiple Components/Single Host

OK, this will probably be confusing….

The example code implements a single *component*. That is, there are two schematic *instances* (X1 & X2) of a hierarchical component block that load the same DLL and connect to the server on the same port to execute identical component logic. Taken together, that’s a *component* for this discussion.

Now assume that we’ve created a new component. It has a different hierarchical component block with whatever ports and attributes are required. The DLL and server code are modified for the ports and attributes and component logic. They are compiled to binaries with different names than the old version.

To use instances of both the old and new components in the same schematic, we could run the two servers with two different port numbers. That would, of course, work just fine.

However, maybe we want to run a single server that can handle both components. It should be possible to have the first message from a client connection pass a code to identify which component is connecting. The server would then launch a thread for the appropriate component logic before handling the next connection request.

For a hobbyist that has only a handful of components, this might be overkill. In fact, the Berkeley Sockets API is probably overkill for a hobbyist and inherently slower than a monolithic C++ DLL.

But if you had dreams of supporting a library of components – maybe micro-controller simulation tool[4] – it would be worth considering….

3 For what it’s worth, I tried adding a hosts file entry to test using a DNS name. As of this writing, QSpice bypasses the hosts file so that trick doesn’t work.

4 My QSpice Microchip micro-controller simulator project could be a candidate.

Conclusion

I hope that you find the information useful. If you implement any of the enhancements, please do share with the community.

And, of course, let me know if you find problems or suggestions for improving the code or this documentation.

--robert

Postscript: The Python Server

After the initial release of this project, I added code for a Python server. It uses the same schematic and client DLL as above so simply start the server with...

```
python cb12_hostserver.py 666
```

… before running the CB12_SocketAPI.qsch simulation.

But let’s be clear: I’m not a Python guy. I used heinous, evil, and untrustworthy AI tools to develop the code. I fixed those faults that my uneducated (un-Pythonated?) eyes could detect. As such, the code almost certainly doesn’t follow Python conventions, idioms, and styles. It’s probably totally rubbish and an embarrassment to experienced Python developers worldwide.

Notably, you should carefully review the code and, in particular, the error handling.

Bottom line: Consider the Python server code a starting point for development.

Postscript: The Java Server

After the initial release of this project, I added code for a Java server. It uses the same schematic and client DLL as above so simply start the server with...

```
java CB12_HostServer.java 666
```

… before running the CB12_SocketAPI.qsch simulation.

See notes in the *.java code for minimum Java version requirements and other ways to compile/launch the Java server.

Unlike Python, I have some experience with Java. However, I again employed heinous, evil, and untrustworthy AI tools to develop the code. As with the Python version, the code is likely rubbish even after my rework efforts. (Honestly, the AI stuff was something of an experiment – I’m curious how things will work out when inexperience meets the shiny new AI tools. Frankly, it’s going to be bad.)

That said, you should carefully review the code. In particular, I omitted most of the error handling code.

Bottom line: Consider the Java server code a starting point for development.