QSpice C-Block Components

Using Bus Wires

Document Revisions:

2024.06.14 – Begin draft.

2024.06.22 – Early release on dev channel.

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 sixth 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.***

This document investigates QSpice schematic "bus wires" with a focus on using them for binary bus data. It builds on concepts and definitions in the prior documents so I strongly suggest that you review those first. See my GitHub repository here for the prior documents and code samples for this one.

Overview

Back in October of 2023, a QSpice forum user started this thread: Usage of bit vector in C-Block. He was trying to connect a QSpice schematic bus wire to a C-Block component Bit Vector port. That seemed useful. I hadn't investigated Bit Vector ports and hadn't played with bus wires.

Knowing nothing about Bit Vector ports or even bus wires, I joined in the discussion. As a group effort, we eventually sussed out a solution although it didn't use the Bit Vector port type.

This paper revisits connecting bus wires to components and, hopefully, adds a bit more detail. Along the way, we'll look into how QSpice sets up the passed `uData` array data and eventually provide code to simplify using boolean/digital bus connections.

Note: Apologies in advance for the length of this paper. You might think that I get paid by the word-count….

Bus Wire/Tap Creation

I found creating bus wires and connecting individual wires to buses a bit non-intuitive. The QSpice Help is rather light on this topic. So, against my better judgment, we'll start by walking through the steps to use bus wires in a QSpice schematic.

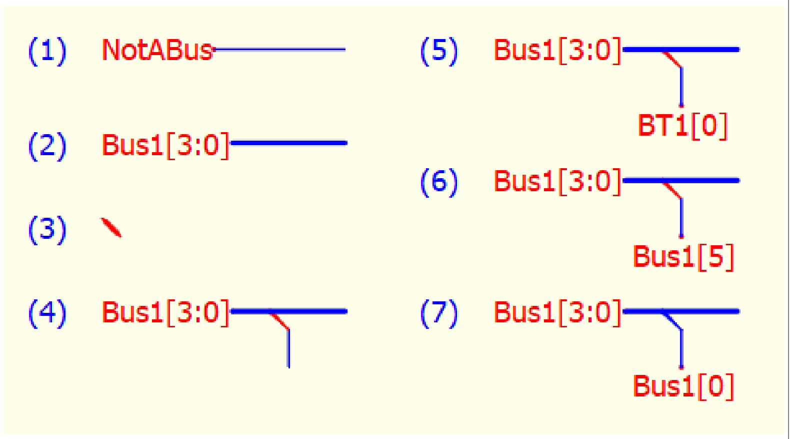

Figure 1: Adding Bus Wires In Schematic

I'm going to go fast here since you're likely quicker on the uptake than I was….

- When you drop a wire onto the schematic, it is displayed as a thin blue line. Give the wire a net name and the wire remains a thin blue wire. It's not a bus. [Figure 1 (1)]
- Give the wire a net name with a "range" and you get a bus. [Figure 1 (2)] `Bus1[3:0]` bundles four wires into a bus. Notice that the bus wire is a thicker blue line.

  It's possible that not all of the bundled wires are used/tapped. You can specify the range from low to high or high to low. The range also need not contain the zero index. That is, you could create a `StrangeBus[3:7]`.
- To connect a wire to the bus, use the QSpice Edit menu or right-click the schematic background and select "Place a Bus Tap." The mouse becomes a bus tap icon [Figure 1 (3)]. Drop one end on a bus. Add a wire to the other end. [Figure 1 (4)] Notice that the tap is red meaning that it's not properly connected to a specific wire within the bus.
- To make that wire a proper connection, you need to name the wire net with the indexed bus name. When properly connected, the tap changes from red to blue. Figure 1 (5) is invalid

because the wire net name is `BT1` and not `Bus1`. Figure 1 (6) is invalid because there is no bus wire at index 5. Figure 1 (7) properly connects to the 0-indexed bus wire (tap is blue).

Moving on….

Beneath The Covers

From QSpice's viewpoint, when you create a bus, you're really creating multiple individual wires, not really an array in any programming sense. If you open `CblockBasics6.qsch` and look at the netlist, you'll see net names like `B1[0], B1[1], … B1[7]`.

So, does the order of indexes matter? That is, is `Bus1[0:3]` different than `Bus1[3:0]`?

Well, if you're *not* connecting the bus to a C-Block, order doesn't matter – at least as far as I can tell. That is, whenever you tap the bus, you specify the index of the wire that you're connecting to so there's not really any confusion about what is connecting to what. That said, *pick an order and stick with it* – it matters when connecting to a C-Block as we'll see.

At the risk of stating the incredibly obvious, remember that wires have no direction and no data type. Component ports, on the other hand, have both direction and data type. We'll return to this in a bit.

Connecting To A C-Block Component

Open `CBlockBasics6.qsch` & `*.cpp` to follow along. It's a bit messy so let's break it down.

We have an 8-bit bus defined as `B1[7:0]`. We're using the lower four bits as inputs from a counter circuit (not shown, see schematic) and the high nibble as outputs. (Ignore `R1-R4 + GND`; they simply ensure that the bus wire names are available for plotting.)

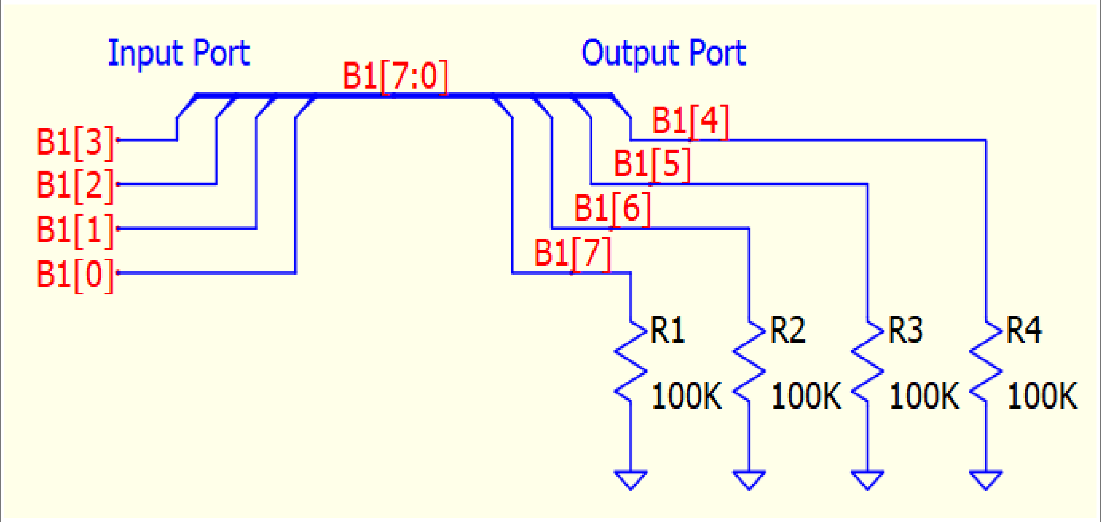

Figure 2: Bus B1

For convenience and testing, there are functions to convert the `B1[3:0]` and `B1[7:4]` wire groups into a voltage value to plot for quick testing/review.

Figure 3 shows the component block and bus connections. Simple, eh?

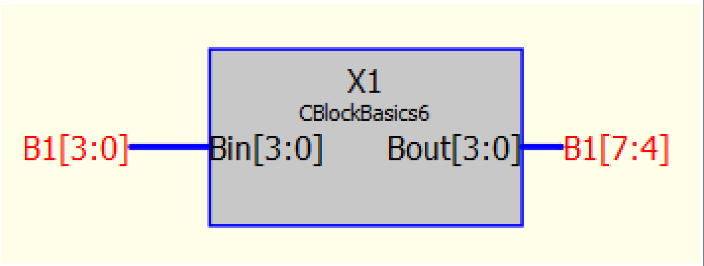

Figure 3: The C-Block Component

The input and output ports (`Bin` and `Bout`, respectively) are defined as type Boolean. Note that the ranges are both "[3:0]." In particular, note that `Bout[3:0]` doesn't match `B1[7:4]`. We'll be getting to this shortly.

Bus-To-Component Data Mapping

Now we get to the fun stuff….

Component-Side Mapping

If we generate a C-Block component, we get this `uData` array parameter definition in the evaluation function:

```
bool  Bin_3_  = data[0].b;   // input
bool  Bin_2_  = data[1].b;   // input
bool  Bin_1_  = data[2].b;   // input
bool  Bin_0_  = data[3].b;   // input
bool &Bout_3_ = data[4].b;   // output
bool &Bout_2_ = data[5].b;   // output
bool &Bout_1_ = data[6].b;   // output
bool &Bout_0_ = data[7].b;   // output
```

There are several things to note:

- The input/output port data is of type Boolean as specified in the component's port data types. If we change the input or output port data type, the above will change to match the specified types.
- Each bus wire arrives as an individual `uData` array element (`data[0]` to `data[7]` above).
- All of the input ports precede all of output ports in the array data. If we had more ports, all of the inputs would still precede output ports. If we also had passed attribute parameters, they would appear immediately after the input ports in the array.
- QSpice names each bus wire element based upon the port index range.

That last item probably needs clarification. The `Bin` port was defined as `Bin[3:0]` so we got four elements named Bin_#_ (`Bin_3_=data[0].b` to `Bin_0_=data[3].b`). If we had instead defined the port as `Bin[0:3]`, the parameter order of the names would have reversed (`Bin_0_=data[0].b` to `Bin_3_=data[3].b`).

Note carefully what's happening here: QSpice generates the code using names that are based on the order of the range indexes. In fact, we could have used entirely nonsensical indexes when defining the ports. For example, had we named the input port `Bin[17..20]`, we would have generated `Bin_17_ = data[0].b` to `Bin_20_=data[3].b` in that order. QSpice is simply generating component variable names starting with the first index and ending with the last.

So the port name (with index range) determines the number of and names of the generated `uData` elements and their order. How are they mapped to the connected bus in the schematic?

Schematic-Side Mapping

Looking at the netlist, we have this:

```
Ø´X1 «B1[3]´b B1[2]´b B1[1]´b B1[0]´b» «B1[7]´b B1[6]´b B1[5]´b B1[4]´b» «»  CBlockBasics6
```

Notice that the port names (`Bin` & `Bout`) don't appear in the netlist; only the bus names appear. So, for the above, `B1[3]` gets mapped to the component `uData data[0]` element. This continues until `B1[4]` is mapped to `data[7]`.

That's pretty straight-forward. If you run into problems, comparing the netlist to the `uData` parameter definitions may be useful.

Further Down The Rabbit Hole

Consider what might happen if we have a mismatch between the size of the attached bus wires and the port declaration. For example, I've changed the input bus from `B1[3:0]` to `B1[4:0]` in Figure 4.

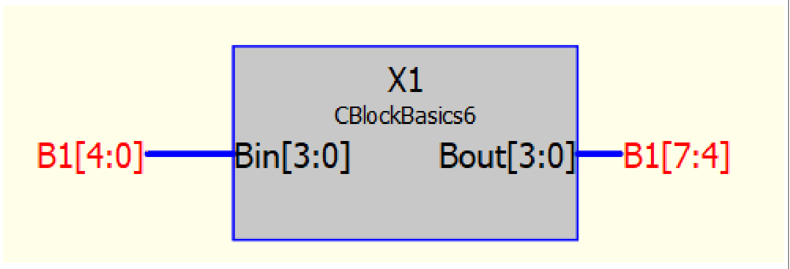

Figure 4: Mismatched Bus/Component

If we generate a new component template, the C++ code does not change. Also, QSpice does not complain about the connection of a 5-wire bus to a 4-wire port during code generation.

The netlist now has this component element:

```
Ø´X1 «B1[4]´b B1[3]´b B1[2]´b B1[1]´b» «B1[7]´b B1[6]´b B1[5]´b B1[4]´b» «»  CBlockBasics6
```

So QSpice is clever enough to pass only 4 wires into the component. QSpice doesn't generate any warning about the mismatch – it just "peels off" a subset of the wires in the bus. Of course, we now have a bug that will be hard to chase down since the four `Bin uData` array elements are now incorrectly mapped to `B1[4]..B1[1]`.

We can go down the "*how many ways can I screw this up rabbit hole*" all day and learn nothing beyond these simple rules:

- Always specify ranges in a consistent order, "[high:low]" or "[low:high]."
- Ensure that the number of schematic bus and component port elements match.

Obey the rules and you'll be fine. Moving on….

Bit Vectors

While we can attach a bus to a port of any data type, it seems most useful if the input/output port type is Boolean for digital signals as we've done in the sample code. It would be nice if the bits were packed into a "register," e.g., all 8 bits packed into a single unsigned char variable. The Bit Vector port data type seemed to be designed for this.

In Figure 5, I've changed the data type of `Bin` to Bit Vector and set the range to [3:0].

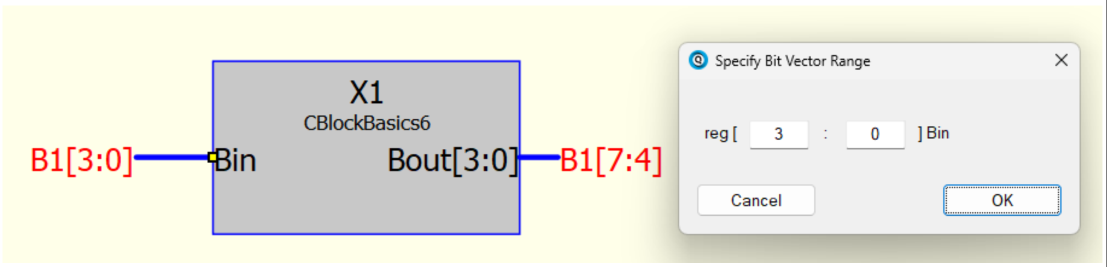

Figure 5: Bin As Bit Vector

Regenerating the C++ code template changes the `uData` array code to this:

```
unsigned char Bin     = data[0].uc; // input
bool         &Bout_3_ = data[1].b ; // output
bool         &Bout_2_ = data[2].b ; // output
bool         &Bout_1_ = data[3].b ; // output
bool         &Bout_0_ = data[4].b ; // output
```

That looks promising. The four input bus bits appear to be mapped into an unsigned char "register" (`data[0].uc`).

Here's the netlist:

```
Ø´X1 «B1[3]´uc» «B1[7]´b B1[6]´b B1[5]´b B1[4]´b» «»  CBlockBasics6
```

Hmm. Well, that isn't right. The `data[0].uc` element will be assigned the value of `B1[3]`, not the bit values of `B1[3:0]`.

I tried other things without success. Eventually, I broke down and bothered Mike Engelhardt (the author of QSpice). As I suspected, connecting buses to Bit Vectors is a Verilog-only thing.

Still, it would be useful for C++ code. What are our options?

Bit Vector Alternatives

At this point, we've seen that we can't use the Bit Vector port type directly in non-Verilog code. The best that we can do is deal with each bus line that's passed individually as they are passed to our code. The QSpice-generated template code was:

```
bool  Bin_3_  = data[0].b;   // input
bool  Bin_2_  = data[1].b;   // input
bool  Bin_1_  = data[2].b;   // input
bool  Bin_0_  = data[3].b;   // input
bool &Bout_3_ = data[4].b;   // output
bool &Bout_2_ = data[5].b;   // output
bool &Bout_1_ = data[6].b;   // output
bool &Bout_0_ = data[7].b;   // output
```

Note what is really happening here:

- The input port values are being copied to local variables. This ensures that we *don't* write to the actual input port values (`data[0..3]` elements) if we change the local values.
- The local variables for the output port values are *references* to the underlying `uData` array locations. Changing the local variable changes the output port values.

Personally, I'm not fond of the template. I would have done this instead:

```
const bool &Bin_3_  = data[0].b;   // input
const bool &Bin_2_  = data[1].b;   // input
const bool &Bin_1_  = data[2].b;   // input
const bool &Bin_0_  = data[3].b;   // input
bool       &Bout_3_ = data[4].b;   // output
bool       &Bout_2_ = data[5].b;   // output
bool       &Bout_1_ = data[6].b;   // output
bool       &Bout_0_ = data[7].b;   // output
```

My version prevents writing to the input ports and ensures that the compiler detects what is likely an error. My QSpice code generator does it my way (see the QCodeGen project here).

A last reminder before moving on: *If you edit the schematic component, be sure to regenerate the code*.

So, how can we "treat the input/output ports as registers?"

Option 1

We can, of course, write something like this to pack the port bits into and unpack the port bits from "bit register" variables:

```
// create digital registers
unsigned char binReg = ((((Bin_3_ << 1) | Bin_2_) << 1) | Bin_1_) << 1) | Bin_0_;
unsigned char boutReg = ((((Bout_3_ << 1) | Bout_2_) << 1) | Bout_1_) << 1) | Bout_0_;

/* do stuff with regs */

// write output reg values
Bout_0_ = boutReg & 0x01;
Bout_1_ = boutReg & 0x02;
Bout_2_ = boutReg & 0x04;
Bout_3_ = boutReg & 0x08;
```

Of course, this will work but it's tedious and, frankly, not at all elegant code. If we had, say, 16-bit buses, this will get annoying and error-prone very quickly. Surely we can do better, be more elegant.

Option 2

Each digital bus wire comes in as an individual `uData` array element and the elements have a predictable order. With this knowledge, we can write code that is more flexible and less error-prone.

The `CBlockBasics6_Alt1.cpp` code attempts to do this for registers up to 8 bits and is easily modified for larger registers.

I'm resisting the urge to provide a line-by-line explanation of the code and hope a general description and the code comments suffice. (To run the code in `CBlockBasics6.qsch`, change component block name attribute to "`CBlockBasics6_Alt1`".)

Here's what I've done at a high level:

- Defined an 8-bit register type ("`BitReg8`"). It's basically an `unsigned char` with some "syntactic sugar" to conveniently manipulate an entire 8-bit value, low and high nibbles, and individual bits.
- Implemented a function to load a bit register from `uData` array elements ("`getReg8()`").
- Implemented a function to save a bit register's bit values to `uData` array elements ("`setReg8()`").

The evaluation function code could be as simple as this:

```
extern "C" __declspec(dllexport) void cblockbasics6_altx(InstData **opaque, double t, uData
*data) {
  // set up uData array offset pointers to bus data
  const uData *pInBusData4  = &data[0];
  uData       *pOutBusData4 = &data[4];

  // local register variables initialized with bus bit values
  BitReg8 inReg  = getReg8(pInBusData4, 4);
  BitReg8 outReg = getReg8(pOutBusData4, 4);

  /*** do stuff with register data -- here we just copy in to out… ***/
  outReg = inReg;

  // finally, set the output port values from the per-instance data
  setReg8(pOutBusData4, 4, outReg);
}:
```

The `CBlockBasics_Alt1.cpp` code is a bit more complicated but only because it implements per-instance data.

Option 3

I suppose that we could come up with fancier solutions. Perhaps templates could provide more elegance and error-protection. But then we'd have to ditch the ancient DMC compiler (included with QSpice) for something more modern.

I'll leave that to you. If you come up with better solutions, please let me know.

Option 4

A potentially better solution would be to write a custom QSpice code generator. Since it would know the port bit-widths and I/O directions, it could produce code that does something similar to Option 2.

However, it could be even better: It could produce code that is specific to bit-widths and `uData` offsets to prevent coding errors.

Oh, gee, I've already written a code generator. I guess this is just a shameless plug.

I hope to add this feature to the project shortly so keep an eye on the QCodeGen project on my QSpice GitHub repository.

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