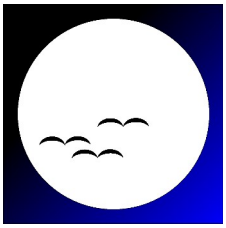


QSpice C-Block Components

The Absolute Basics

Document Revisions:

2023.10.27 – Initial version.
2024.01.03 – Added Update Notes section.

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.")

Overview

For those relatively new to C/C++ coding and unfamiliar with DLLs, getting started writing QSpice C-Block components may be a bit intimidating. That's a shame – the ability to write DLL code is pretty powerful stuff. Once you understand what's going on, it's really not that hard. So, let's see if we can get over the conceptual hurdles.

To be clear, this document is intended to explain the bits and pieces of C-Block code. I'll use a small demonstration (EvilAmp) but it is merely to get code in front of us to discuss, not to develop anything particularly useful.

C-Block Component Use Cases

Before digging in, let's clear up a possible misconception: C-Block components are intended for simulating digital stuff. You can do analog-like things for sure. But, if it contains feedback loops, it may not work as you plan.

Here's why: For each simulation point, QSpice will call the DLL with input voltages. The DLL then sets its output voltage(s). The simulator does not use the outputs to re-calculate inputs and call the DLL a second time for that simulation point.

Bottom line: Digital only. Don't feed outputs back into inputs and expect no surprises.

Code Templates

Qspice makes generating a basic component code template super-simple. See the QSpice tutorial videos for step-by-step instructions:

> The QSpice C++/Verilog starter video (Mike Engelhardt)
>
> The Qorvo YouTube channel

For demonstration purposes, I've already:

- Created a schematic containing the hierarchical block for the component (EvilAmp.qsch).
- Set the Symbol Type to "ϕ(.dll)" in the Symbols Properties.
- Added input and output ports of data type double.
- Added an attribute, Gain, of type integer and value of 666.
- Added a voltage source and connected it to the input port on the EvilAmp block.

If you've not already done so, get the associated demo files. Take a few minutes to look at the schematic and generated code before we step through it bit by excruciating bit.

The Simplest Version

Generate a code template and select none of the checkboxes.

We get this:

```
// Automatically generated C++ file on Tue Oct 10 14:02:20 2023
//
// To build with Digital Mars C++ Compiler:
//
//    dmc -mn -WD evilamp.cpp kernel32.lib

union uData {
  bool                  b;
  char                  c;
  unsigned char         uc;
  short                 s;
  unsigned short        us;
  int                   i;
  unsigned int          ui;
  float                 f;
  double                d;
  long long int         i64;
  unsigned long long int ui64;
  char                 *str;
  unsigned char        *bytes;
};

// int DllMain() must exist and return 1 for a process to load the .DLL
// See https://docs.microsoft.com/en-us/windows/win32/dlls/dllmain for more
// information.
int __stdcall DllMain(void *module, unsigned int reason, void *reserved) {
  return 1;
}

// #undef pin names lest they collide with names in any header file(s) you might
// include.
#undef In
#undef Out

extern "C" __declspec(dllexport) void evilamp(
    void **opaque, double t, union uData *data) {
```

```
  double  In   = data[0].d;   // input
  int     Gain = data[1].i;   // input parameter
  double &Out  = data[2].d;   // output

  // Implement module evaluation code here:
}
```

We can ignore the `DllMain()` function. Don't change it, don't delete it.

The magical incantation preceding some functions – `extern "C" __declspec(dllexport)` – simply tells the compiler/linker to make the function callable from QSpice. Again, don't worry about it.

That leaves the evilamp() function. This is the "*evaluation function*" called by QSpice whenever it reaches a simulation point and needs our code to do whatever it does. Note that the name of this function must match the DLL name (without the ".dll") and is in lower case. It also matches the name in the schematic hierarchical block although the case may be different.

The parameters to the evaluation function are:

- `void **opaque` – A pointer to a pointer. We'll discuss this later.
- `double t` – The simulation time in seconds.
- `union uData *data` – A pointer an *array* of port and attribute parameters passed from QSpice.

The uData array elements are defined in the template as:

```
  double  In   = data[0].d;   // input
  int     Gain = data[1].i;   // input parameter
  double &Out  = data[2].d;   // output
```

These are the ports and attributes passed from QSpice. The order (and types) of these values may change if you alter the hierarchical block. If you do make changes, regenerate the C-Block code to get the proper offsets (and data types).

Because the first two are inputs, they are copied to local variables. That is, you aren't supposed to change the values directly in the uData array.

The third parameter is the value on the output port of the component. We will change this value so the `Out` variable is declared as a reference.

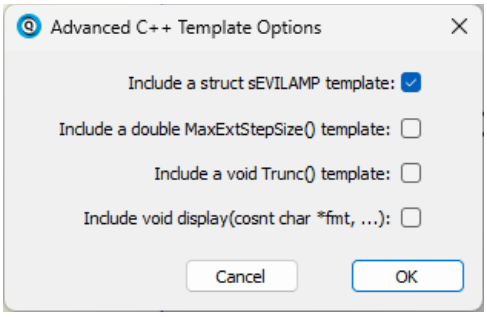
Now we can make our amp component evil. We add the following line:

```
Out = In * Gain;
```

Whatever voltage is applied at the component's `In` port will be multiplied by the `Gain` attribute and appear on the `Out` port.

Per-Instance Data

OK, what if we need to save some data between QSpice evaluation function calls? A common case is an input port being driven by a clock signal; we want to save the clock state (high/low) and, at the next call, compare the last and current state to determine if there was a rising/falling clock edge.



We could add a "`lastClock`" variable above the evaluation function (i.e., global scope). We'd then compare `In` to `lastClock`. If `lastClock != In`, we had an edge.

This would work but there's a problem: There is one DLL with one copy of the `lastClock` variable. If we place a second copy of our component on the schematic, both component "***instances"*** would share the same `lastClock` variable. That might be undesirable. It would be better if each instance of the component had its own copy of the `lastClock` variable (i.e., "***per-instance data***").

To solve this, let's generate a new C-Block and enable the "Include a struct…" option.

After we generate the file, we get some additional code:

```
#include <malloc.h>

void bzero(void *ptr, unsigned int count)
{
   unsigned char *first = (unsigned char *) ptr;
   unsigned char *last  = first + count;
   while(first < last)
      *first++ = '\0';
}

struct sEVILAMP
{
  // declare the structure here
  double lastClock;
};

extern "C" __declspec(dllexport) void evilamp(struct sEVILAMP **opaque, double t, union uData
*data)
{
   double  In  = data[0].d; // input
   double &Out = data[1].d; // output

   if(!*opaque)
   {
      *opaque = (struct sEVILAMP *) malloc(sizeof(struct sEVILAMP));
      bzero(*opaque, sizeof(struct sEVILAMP));
   }
   struct sEVILAMP *inst = *opaque;

// Implement module evaluation code here:
   if (inst->lastClock != t) {
      /* do stuff... */
      inst->lastClock = t;
   }
}

extern "C" __declspec(dllexport) void Destroy(struct sEVILAMP *inst)
{
   free(inst);
}
```

(Code in bold italics added to template after generation.)

The template now declares a structure, sEVILAMP, and we can add our "per-instance" `lastClock` variable there. Note that this is only a declaration – no storage for this data has been allocated.

The evaluation function signature has changed:

```
void evilamp(struct sEVILAMP **opaque, double t, union uData *data)
```

The first parameter, `opaque`, changed from a `void**` to a `struct sEVILAMP**`. That is, the passed parameter is ***the address of a pointer*** to an `sEVILAMP` structure.

Looking at the next few lines of code, we see how this works:

- If `opaque*` points to a null pointer, memory for an `sEVILAMP` structure is allocated.
- The address of the allocated structure is stored in QSpice at the address pointed to by *`opaque`. (Now `opaque*` no longer points to a null pointer.)
- The `bzero()` function is called to fill the newly allocated `sEVILAMP` structure with zero bytes.

Note that QSpice doesn't know anything at all about the `sEVILAMP` structure or the data that we put there. All QSpice does is give us a place to store a pointer to a block of data (the structure) that we allocate.

Now, each time that the evaluation function is called, the pointer to the `sEVILAMP` structure is saved in the `inst` variable simply for convenience. We'll use `inst->` to access the per-instance data. The added code demonstrates using the `lastClock` per-instance data.

Note: Pointer stuff is tough for beginning C/C++ programmers so, if the above wasn't clear, well, blame me for explaining it poorly. But do take time to get a handle on it (sorry, bad pun) because it will get worse when we get to the `Trunc()` function.

The last thing of note is the new `Destroy()` method. QSpice calls this at the end of the simulation and we free the allocated per-instance data here.

The display() Template Option

When generating the code template, checking this option adds a display() function. It's simply a wrapper for `printf()` and writes the data to `stdout` which the GUI displays in the Output window.

This is particularly useful for debugging C-Block code. However, each call adds a 60ms delay to the simulation execution. That's *clock time*, not simulation time, so don't put `display()` calls where they will get executed on every evaluation function call unless you need a coffee break.

The MaxExtStepSize() Template Option

If we enable the MaxExtStepSize() option when generating the code template, we get this additional code:

```
extern "C" __declspec(dllexport) double MaxExtStepSize(struct sEVILAMP *inst)
{
   return 1e308; // implement a good choice of max timestep size that depends on struct sEVILAMP
}
```

And we now reach the limits of my confidence. "Implement a good choice of max timestep size that depends on [the per-instance data]" is frustratingly vague.

I do know a couple of things:

- This function is called between the evaluation function calls (after several initial evaluation function calls).
- We get a pointer to our per-instance data.

- And 1e308 seconds is longer than the heat-death of the universe. This default return value clearly means “I’m in no hurry.”

I’ve used it where I’m trying to create a component that internally clocks at a certain frequency. For example, if I have a component with a “clock frequency” attribute of 1KHz, I might return 0.00100. It seemed to help when the simulation points weren’t hitting on my clock frequency reliably.

Do let me know if you have better information.

The Trunc() Template Option

If we enable the Trunc() option, we get the following additional template code:

```
extern "C" __declspec(dllexport) void Trunc(struct sEVILAMP *inst, double t, union uData *data,
double *timestep)
{ // limit the timestep to a tolerance if the circuit causes a change in struct sEVILAMP
   const double ttol = 1e-9;
   if(*timestep > ttol)
   {
      double &Out = data[1].d; // output

      // Save output vector
      const double _Out = Out;

      struct sEVILAMP tmp = *inst;
      evilamp(&(&tmp), t, data);
   // if(tmp != *inst) // implement a meaningful way to detect if the state has changed
   //    *timestep = ttol;

      // Restore output vector
      Out = _Out;
   }
}
```

Again, we’ve reached the limits of my knowledge. “Implement a meaningful way…” isn’t very, uh, meaningful because I don’t know exactly how QSpice works. The PracticalSMPS example didn’t help me much.

But I do know some things. Or can, at least, explain what’s happening if not why.

Here’s the breakdown of the above template:

- The function receives a proposed next sample time in the `t` parameter along with the port/attribute data in the `uData* data` parameter.
- Then, a *copy* of the passed per-instance data (`inst`) is made (`tmp`).
- A copy of the `uData` output port is saved (`_Out`).
- The evaluation function is called. The temporary copy of the per-instance data (`tmp`) is passed along with the `uData` data. Note that the evaluation function is allowed to change the passed per-instance data (`tmp`) and output port values (in `uData`).
- After the call returns, the commented out code does a simple compare of the original per-instance data (`*inst`) to the temporary data (`tmp`) to determine if the evaluation function changed the data – any of it. (I suppose that a more nuanced case would compare individual

per-instance data elements.) If anything changed, the `timestep` value is set to a “reasonably small” value to tell QSpice to make the next call to the evaluation function “soon.”

- The original incoming per-instance data wasn’t changed – remember, we passed a temporary copy. However, the `uData` output port was passed directly to the evaluation function and may have been changed. So, the original `Out` value is restored from `_Out`.

Now, you might wonder why all of the sleight-of-hand, the call to the evaluation function from `Trunc()`? Here’s my best guess: It’s best to have a single place in the code that does the calculations. Calling the evaluation function ensures that you don’t duplicate (or fail to faithfully duplicate) the actual functional state code.

Me? I’d have a function that’s called from both places and does the heavy lifting. But that’s simply my preference.

So, before explaining how I’ve used this, here’s what I’m assuming (all of which might be wrong):

- QSpice isn’t simulating at constant intervals. That is, it intelligently (magically?) decides that, after a simulation at time T, it doesn’t need to simulate again until time T + [some time]. That [some time] part isn’t a constant.
- The Trunc() function is called with the predicted input port values for T + [some time] passed in the uData parameter as well as the prior T per-instance data. Our code is simply supposed to decide if T + [some time] is too long, that our component’s state should change before that.
- If so, we return a smaller [some time] increment and QSpice resets the next simulation time-point to reflect this change.

So much for guessing. Here’s how I’ve used it:

- My component is supposed to “clock” at a frequency passed as a component attribute. Let’s say that’s 1KHz. The clock period is 0.001 seconds.
- Each time that the evaluation function is called, the next desired simulation clock tick is calculated and stored in the per-instance data.
- If the simulation time in `Trunc()` (the `t` parameter) is past the desired tick time, return a small value in `*timestep`. Otherwise, return the difference between the desired and proposed (`Trunc()` `t` parameter) times.

This seems to have worked so far. But I’m in deep water here and, as mentioned several times, I may be very, very wrong.

Update Notes

I’m too lazy to completely revise all of the above so check here for important revision notes.

2024.01.03

- As of sometime in November 2023, returning -1e308 from MaxExtStepTime() (exactly this value) aborts the simulation cleanly. This is useful if the component detects a condition where continued processing isn't desired.
- The QSpice 2023.12.31 release changed the way that DLLs are handled in stepped simulations. Previously, each step value executed as an independent simulation. That is, from the DLL's viewpoint, it was being loaded, initialized, and run independently for each step value in the QSpice simulation. After this update, the DLL is loaded/initialized once at the beginning of the simulation and not unloaded until after the full simulation (for all step values) is complete. This has multiple implications that I'll address in a separate document.

I'm stopping here. Let me know where I've gone far afield.

--robert