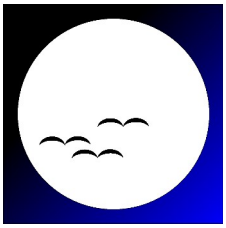

QSpice C-Block Components

Post-Processing Techniques

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 eighth 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()).***

This paper 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.

Preface / Author's Note

This project started as something small. And then, suddenly, it wasn't. The documentation is, therefore, somewhat disjointed even after many revisions. Sorry about that. "Perfect is the enemy of good enough" and it's time to move on….

Background

Writing robust C-Block components requires that our code handle:

- Multiple schematic component instances.
- Multi-step simulations.
- Shared resource management (if any).

These topics were covered in the C-Block Basics #2 (“Shared Resources & Reference Counting”) and #3 (“DllMain() & Simulation Start/End Tasks”) papers.

This document revises those techniques using new QSpice features. Consider those papers “minimum required reading” for understanding the issues at hand.

Overview

This paper provides a fairly complete example to “post-process” data in a QSpice C-Block component. The code demonstrates how to:

- Do one-time initialization/finalization when a simulation is started/completed, even when multiple components are in the schematic or running a multi-step simulation.
- Log simulation & component state information to a binary and text files.
- Generate log file and post-processing names & commands from user-supplied component attribute strings.
- Create a stand-alone post-processing *.exe to convert binary log data into a *.csv file.
- Launch an external program or command (*.exe, *.bat, *.cmd) from a component DLL.

Along the way, we’ll use some QSpice features added in recent releases which obviate the need for some workarounds in prior papers.

If you’re interested, buckle up – it’s going to be a fun ride. OK, maybe just a ride….

New QSpice Features

August and October 2024 QSpice releases added some new features that we will use. I’ll break them into two categories: State Information and Function Calls.

New State Information

If you generate a default C-Block template from a schematic component block with all options selected, you’ll find these new declarations/definitions (there are more):

```
extern "C" __declspec(dllexport) const int    *StepNumber              = 0;
extern "C" __declspec(dllexport) const int    *NumberSteps             = 0;
extern "C" __declspec(dllexport) const char  **InstanceName            = 0;
```

They are pretty much what you’d think:

- `StepNumber` – Pointer to the current processing step in a multi-step simulation. If not a multi-step simulation, `*StepNumber` is set to 1.

- `NumberSteps` – Pointer to the expected number of steps[1] to be processed in a simulation. If not a multi-step simulation, `*NumberSteps` is set to 1.
- `InstanceName` – Pointer to a pointer to the schematic name of the currently processing instance. `**InstanceName` points to a null-terminated ASCII string.

We can use these like this:

```
Display("The current step is %i of an expected %i steps for component name %s\n",
    *StepNumber, *NumberSteps, *InstanceName);
```

Easy enough as long as we get the pointer de-referencing correct.[2]

Of course, if you want to copy the instance name into a fixed-length array (e.g., to write to a file), you'll need to handle possible array overflow.[3]

New Function

If you generate a default C-Block template from a schematic component block with all options selected, you will not find this new function:

```
extern "C" __declspec(dllexport) void PostProcess(struct sMYDLL *inst) {
  // add code here
}
```

`PostProcess()` is called once for each schematic component instance prior to the `Destroy()` calls for each instance but only at the completion the last simulation step. When called, QSpice is done collecting simulation data and has closed all of its output files.

A Skeletal Example & Analysis

To demonstrate the subtleties, let's consider some code where we need to handle a resource shared between multiple schematic component instances. We'll also need to handle multiple simulation steps.

Let's assume that we have a log file of some sort (the shared resource). We might write some code to open the file as follows:

```
#include <stdio.h> // for file I/O
...

// global variables
FILE* gFile      = NULL;
bool  gFirstTime = true;
...

// evaluation function
extern "C" __declspec(dllexport) void mydll(struct sMYDLL **opaque, double t, union uData *data) {
    ...

    if (!*opaque) {
        // allocate instance data structure (not shown)
```

1 A simulation may be terminated "early" from the schematic by using the abortsim() function in a behavioral voltage/current source or from a component by returning -1e308 from MaxExtStepSize().

2 InstanceName is not valid inside the Trunc() function nor inside evaluation function when called from Trunc(). Display() calls are also ignored during Trunc() processing. If you need to log the instance name in Trunc() you can (1) add a "const char* instName" member to the per-instance structure and assign "myStruct->instName = *InstanceName" during initialization or (2) save a copy of the **InstanceName string in a character array in per-instance data.

3 Per the QSpice author (Mike Engelhardt) there's not a specific limit on the length of instance names.

```
        ...

        // if first time, open log file
        if (gFirstTime) {
            gFirstTime = false;
            gFile = fopen("logname.log", "w+"); // use "w+b" for binary files
            // error handling omitted
        }
        // other component instance initialization
        ...
    }

    // evaluation function code goes here; gFile is open for fwrite()/fprintf()/etc.
    ...
}
```

Above, we open the file for create/overwrite in the instance initialization portion of the evaluation function of the component instance called first by QSpice for the first simulation step. Sorry – long sentence. Short version: The file is opened exactly once regardless of the number of component instances or simulation steps.

Our `Destroy()` and `PostProcess()` functions might look like this:

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

extern "C" __declspec(dllexport) void PostProcess(struct sMYDLL *inst) {
    fclose(gFile);
}
```

The above will close the log file before the final `Destroy()` call at the end of the simulation. There are two possible issues with that:

- Maybe we want to write some final data for this instance and simulation step in the `Destroy()` function. Well, the file has already been closed.
- If there are multiple component instances, `fclose()` will be called multiple times.[4]

So, let's fix that and add one more complication: The component needs to do some final processing on the log data after all instances have written their data to the log file and the file is closed.

To do all of that, we'll need a few modifications to track the number of active component instances, recognize when all simulation steps have been completed, etc.

Here's some general code for that:

```
#include <stdio.h>
...

// global variables
FILE* gFile      = NULL;
bool  gFirstTime = true;
bool  gPostFlag  = false;
int   gRefCnt    = 0;
...

// evaluation function
extern "C" __declspec(dllexport) void mydll(struct sMYDLL **opaque, double t, union uData *data) {
    ...
```

4 Technically, it's not an error to call fclose() on a closed file. But it is, to my thinking, very bad coding form. Open a file once and close it once if you want to write code that won't break easily when changed by a future maintainer. Or when changed by future you.

```
    if (!*opaque) {
        // allocate instance data structure (not shown)
        ...

        // new instance is active, increment reference counter
        gRefCnt++;

        // if first time, open log file
        if (gFirstTime) {
            gFirstTime = false;
            gFile = fopen("logname.log", "w+"); // use "w+b" for binary files
            // error handling omitted
        }
        // other component instance initialization
        ...
    }

    // evaluation function code goes here; gFile is open for fwrite()/fprintf()/etc.
    ...
}

extern "C" __declspec(dllexport) void PostProcess(struct sMYDLL *inst) {
    gPostFlag = true;
}

extern "C" __declspec(dllexport) void Destroy(struct sMYDLL *inst) {
    if (gPostFlag) {
        // write final instance data record (if any) to file before freeing instance data
        // for this simulation step
        ...
    }

    // free instance data and decrement reference counter
    free(inst);
    gRefCnt--;

    // if reference counter is not zero or not in post processing, return
    if (gRefCnt || !gPostFlag) return;

    // write final simulation run data record (if any) and close log file
    ...
    fclose(gFile);

    // at this point, all instances are complete/freed and log file is closed
    // add code for final post-processing activities here
    ...
}
```

While that might seem complicated, it should be reliable and maintainable.

The example code implements, more or less, this approach.

A Complete Post-Processing Framework

The accompanying example files demonstrate a complete version of the above plus a few extras. It's a bit more complex but follows the same general approach.

At a high level, the code simply writes simulation data to a binary log file and then runs an external program or command to convert the binary file to a *.csv text file.[5] The file names and post processing command are constructed from user-supplied component attribute strings.

5 Yes, the *.csv could be written directly from the component but then we'd not learn about launching external commands, handling binary vs text files, and whatnot, would we?

The code is (hopefully) well-commented and largely self-explanatory so I'll spare you the line-by-line commentary. On the other hand, I'll cover a few key elements below.

Files

CBlockBasics8.qsch	QSpice demonstration schematic.
CBlockBasics8.cpp	The component DLL code.
CBlockBasics8_pp.cpp	The post-processing binary-to-CSV log converter program code.
CBlockBasics8_pp.cmd	An example post-processing batch command file.
CBlockBasics8.h	Common header file for the *.cpp files.
CBlockBasics8_pp_dmc.cmd	Batch command to compile CBlockBasics8_pp.cpp to *.exe using DMC.

Compiling The Code

The code uses pre-compiler conditional directives to support both the DMC compiler as well as more modern compilers[6]. If you're compiling with something other than DMC, I presume that you don't need detailed instructions. (See comments in the *.cpp files for the MSVC compilation that I used.)

For DMC, you can compile the CBlockBasics8.cpp component DLL from within QSpice as usual.

Assuming that you've installed QSpice to the default installation folder, you can use the included batch file to compile the CBlockBasics8_pp.exe standalone post-processing program:

- Open File Explorer and navigate to the CBlockBasics8 project folder.
- Double-click on the CBlockBasics8_pp_dmc.cmd batch file.

If you installed QSpice somewhere other than the default installation folder, you'll need to edit the *.cmd for the path to DMC.

Launching An External Program or Batch Command

For me, the most interesting part of this project was running an external program/command from within the component DLL. Windows contains quite a few ways to do this:

- `_spawn()` variants
- `_exec()` variants
- `system()`
- `ShellExecute()/ShellExecuteEx()`
- `CreateProcess()`

6 See "About The QSpice Compiler (DMC)" and "VSCode Configuration" topics on the GitHub repository for more information.

The code uses `CreateProcess()`. Note that the component (and, therefore, QSpice) is suspended until the launched process completes.[7] See the `runExtCmd()` function in CBlockBasics8.cpp for the implementation.

Since you can run a batch command file (*.bat or *.cmd), you can easily construct a post-processing activity that performs a sequence of processing steps without altering the component code. The possibilities are endless – you could run PowerShell or even Bash commands if WSL is installed. The only limitation is that the component (and QSpice execution) is suspended until the command completes.

Note: Anything written to `stdout` by a launched process is captured in the QSpice Output window. The text writes are not serialized and will likely corrupt the Output text so it's best to avoid writing to `stdout` if possible.

About C++ Code

A quick note for folk with no C++ programming experience: QSpice-generated C-Block code and demonstration samples look like straight C code. Technically, they are C-style code compiled with a C++ compiler.

The code for this project is mostly C-style code with one notable exception: The `BinData` structure in CBlockBasics8.h uses C++ constructors to initialize the structure data values.

Hopefully this won't deter non-C++ programmers too much.

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

7 It would be nice if the code could launch an external program that "lived" beyond the end of the simulation. I tried a number of the above Windows techniques (system(), ShellExecute(), etc.) and variations of CreateProcess() but was unable to achieve that. It seems that, when the component DLL is unloaded, all processes started by the DLL are killed. If you know of a solution, please let me know.