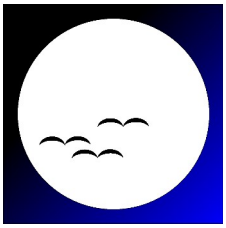

QSpice C-Block Components

Implementing An Internal Clock Source

Document Revisions:

2024.06.04 – Initial published version.

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 fifth 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.***

This document demonstrates a method to implement an "internal clock source" and examines a "peculiarity" with the QSpice adaptive time stepping algorithm. 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

A common C-Block component requirement is a clock source to trigger state changes. I thought that a QSpice pulsed voltage source as an input would be an ideal clock trigger for such components.

I was wrong.

This paper describes the problem and implements a solution using `MaxExtStepSize()`. I'll digress a bit and explain a key difference between `MaxExtStepSize()` and `Trunc()` along the way.

Buckle up….

The Problem

One of the first QSpice C-Block components that I developed and shared was an audio WAV file generator. It would take two input signals from the schematic, sample, and write them to a stereo *.WAV file. To provide a clock source to trigger sampling at the desired WAV frequency/sample rate, I did something like Figure 1. Simple. Straight-forward. And, to my surprise, unreliable.

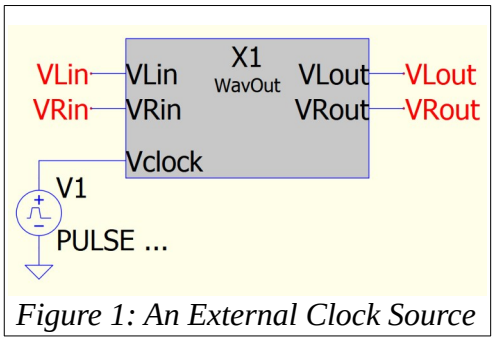

Figure 1: An External Clock Source

It worked just fine in my limited testing. Eventually, a user complained that the WAV file was dropping samples when longer simulations were run. This took a while to chase down because it really was related to the longer simulation run times. The V1 clock source really was dropping pulses. Really.

But why? Well, as best I can tell, it's a "feature" of the QSpice "magical speed sauce."

The QSpice Adaptive Time-Step Algorithm

Note: I don't have any inside information so the following explanation is based solely on personal observation and conjecture.

QSpice is fast. Part of the magic sauce is "adaptive simulation step selection." That is, QSpice analyzes the circuit behavior and adjusts the sample/time density dynamically. Simulation sample rate is increased around rapidly changing voltages or currents. Where voltages or currents change slowly, the sample rate is decreased.

I suspect that QSpice also gauges the impact of changing input voltages to output changes of schematic elements, a sort of sensitivity analysis for each component. Where changing inputs don't change outputs much, QSpice may reduce the weight of the element in the adaptive step selection algorithm.

All of that makes sense for well-defined circuit elements like native symbols and models. C-Block component behavior, on the other hand, isn't well-defined. In fact, to QSpice, they are black boxes.

Presumably, the pulse signal in Figure 1 doesn't do anything that QSpice can clearly associate with circuit activity (especially if the outputs are floating). As a result, QSpice would occasionally simply drop clock transitions from V1.

I needed a more reliable solution. The component needs to be able to force QSpice to take samples exactly when needed.

Note: The QSpice author has suggested that adding a capacitor across Vclock to ground would work around this behavior. I've not analyzed that solution.

Trunc() vs MaxExtStepSize()

Both `Trunc()` and `MaxExtStepSize()` allow us to influence QSpice sample timing. In theory, we could use either one or a combination of both. Let's take a moment to consider the differences between the two functions.

After some initialization, the QSpice simulation cycle works like this:

1. QSpice calls the evaluation function to take a simulation sample.
2. It then calls `MaxExtStepSize()` to get a maximum sample time increment for the next simulation step.
3. It then selects a next simulation time-point for the next sample using the magical adaptive time step algorithm. This time increment is, of course, not greater than the value returned by `MaxExtStepSize()` but may be less.
4. Then QSpice calls `Trunc()` with pre-calculated values for this next sample point. `Trunc()` can force a smaller time-step by returning a value in `*timestep`. If `Trunc()` reduces the step, QSpice loops back to #3 with a new simulation time-point for the next sample.
5. Otherwise, the simulation sample is taken by returning to #1.

Note the key difference:

- When QSpice calls `MaxExtStepSize()`, it has not pre-calculated anything. The return value simply limits the simulation time increment for the next time-step.
- When QSpice calls `Trunc()`, it pre-calculates the circuit state that would be present at a future simulation time-point. If `Trunc()` reduces the time-step by returning a value in `*timestep`, QSpice pre-calculates new circuit values for the reduced step and calls `Trunc()` again.

The `Trunc()` pre-calculations are computationally expensive. For our current purpose, we should avoid `Trunc()` if possible for speed. We'll try to make do with `MaxExtStepSize()`.

The Self-Clocking Component

The `CBlockBasics5.cpp` code demonstrates one way to generate a clock internally. It's pretty straight-forward.

The evaluation function does all of the heavy lifting. After some one-time initialization, it calculates and saves the next clock tick time and time increment in per-instance data (`next_t` and `incr_t`, respectively). An instance variable (`tickCnt`) keeps up with the click tick count – `calcTickTime()` uses it to calculate the next clock tick time with minimal rounding error for minimal clock jitter.

When a clock tick occurs, the time increment (`incr_t`) is set to `TTOL`. Otherwise, the increment is the time until the next clock tick.

`MaxExtStepSize()` simply returns the next step increment (`incr_t`) as set by the evaluation function. (The test for `incr_t <= 0` handles the several calls that QSpice makes during initialization.)

Now, about `TTOL`: This sets the *maximum* rise/fall time for clock transitions. The actual transition time may be less. In fact, the edge transition time may vary from tick to tick. (I suppose that we could force an exact edge transition time/slope but it would involve additional code to implement interpolations. I'll leave that as an "exercise for the student.")

So, that's it. It's pretty easy and, in my limited testing, more reliable than a schematic clock input.

Conclusion

If you made it this far, well, congratulations and thanks for sharing the ride. I hope that you find the information useful.

Please let me know if you find problems or suggestions for improving this documentation.

--robert