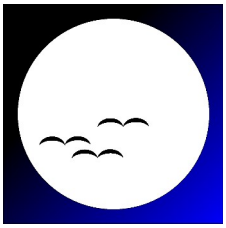

QSpice C-Block Components

Revisiting Trunc()

Document Revisions:

2024.05.27 – 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 fourth 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.***

This document will hopefully demystify the Trunc() function, what it does, and how to use it. It builds on the first paper so consider that "required reading" for terminology and overview of C-Block components. See my GitHub repository here for the prior documents and code samples for this one.

Overview

The C-Block `Trunc()` function has been the subject of several lengthy discussions on the QSpice forum. The "official example code" (the "ACME Semiconductor" demo program) is complicated and, frankly, doesn't really clearly explain how `Trunc()` works and why/when/how we'd want to use it.

Time has passed, stuff has been learned….

In this paper, I will:

- Demonstrate the problem that the `Trunc()` function is designed to solve.
- Detail the `Trunc()` function parameters.
- Describe the general QSpice simulation cycle as it relates to C-Block components.
- Present code for a "canonical" `Trunc()` function.

Be warned in advance: *This is a long and tedious paper.* And, worse, there will be a "Part 2" because, well, I've been working on a single comprehensive document for months. It's too complicated to cover all of the dark corners and use cases in one go.

If you're new to writing QSpice C-Block components, I strongly suggest that you start at the beginning of this series for basic concepts and terminology.

The Problem

To understand the problem that `Trunc()` addresses, let's start with the `CblockBasics.qsch` schematic and `CBlockBasics4.cpp` example code. The component is a simple comparator: `Vout` is set high if `Vin` > `Vref` and set low otherwise. The component's `TTOL` attribute sets the "*time tolerance*" used in the `Trunc()` function. If `TTOL` is zero, the `Trunc()` function is disabled.

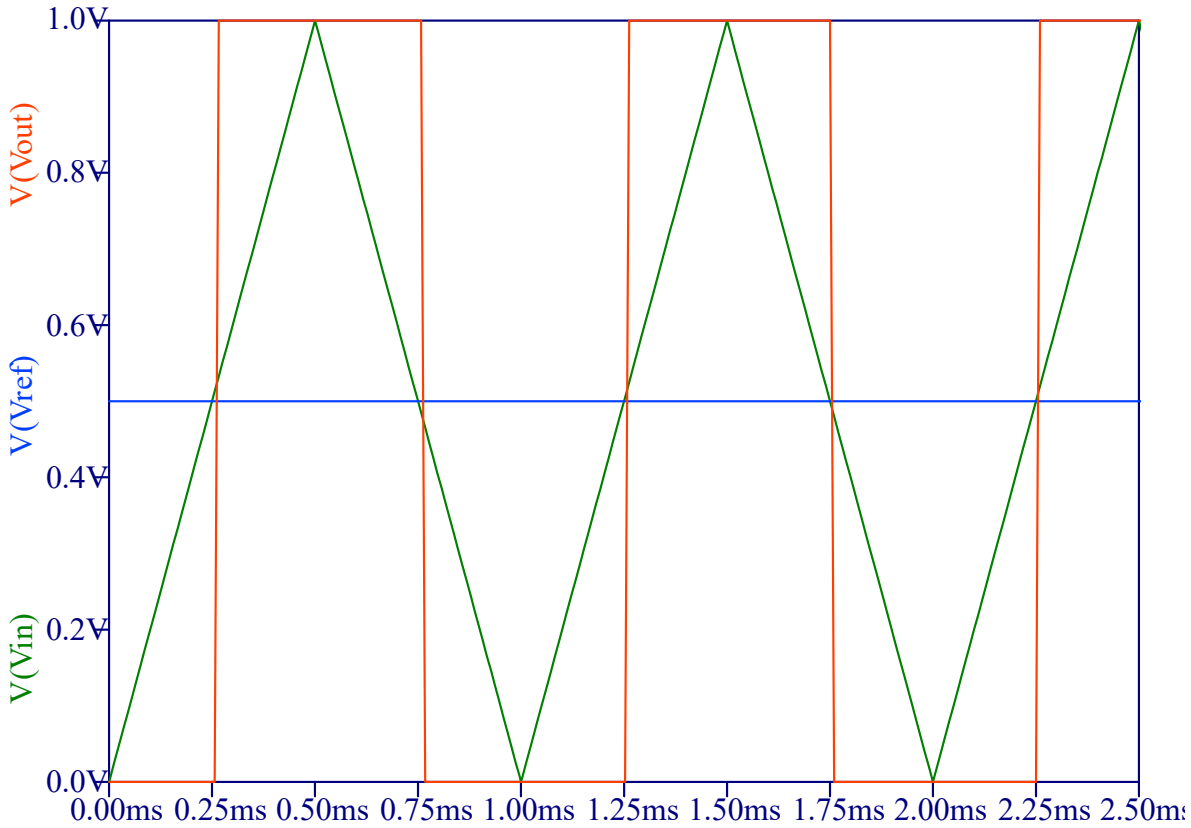

Figure 1 – MAXSTEP=10µS (Trunc() disabled)

We expect a comparator to produce something like Figure 1. Here, `Trunc()` is disabled and I've set the `MAXSTEP` option to 10µs in the schematic. (Only the first 2.5ms of simulation is shown.)

Setting `MAXSTEP` to a small value gives us a nice result but forces QSpice to take far more samples than are really needed. Presumably, there are some 250 sample points in the 2.5ms shown above. And, of course, more sample points == longer simulation run times.

Figure 2 shows the result without using `MAXSTEP` and with `Trunc()` disabled.

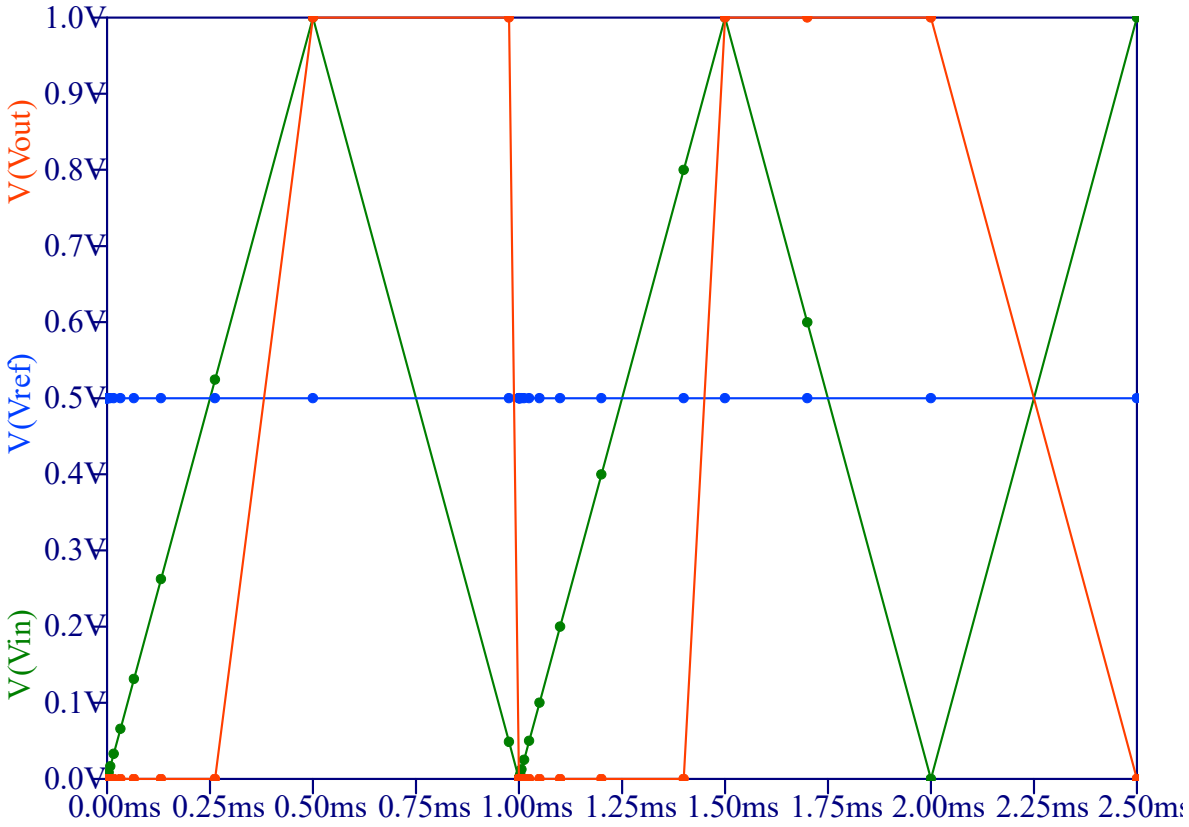

Figure 2 – Trunc() Disabled (TTOL=0)

Clearly, the result looks not at all like a proper comparator.

The Problem: QSpice uses a predictive/adaptive algorithm to select simulation time-steps. It attempts to increase the simulation sample point density around time-points where circuit values are changing rapidly.[1]

Because QSpice knows nothing about the internals of our component, the simulation points were selected based on only the schematic voltage sources. Above, we get only a couple of dozen sample points; clearly more are needed. (Note that, if we had a more complex schematic, QSpice would likely generate many more samples.)

To help QSpice do its predictive/adaptive time-stepping/sampling magic, we can use `Trunc()` to give QSpice some insight into our component.

Figure 3 shows a simulation using `Trunc()` with a `TTOL` of 10µs. This is a definite improvement. Now we get nice sharp edge transitions. While there are still many "extra" sample points, it's not as bad as setting MAXSTEP to 10µs.

1 See QSpice forum member @KSKelvin's excellent documentation for more details about QSpice's adaptive time-step algorithm: QSpice - How Time Step Works.pdf.

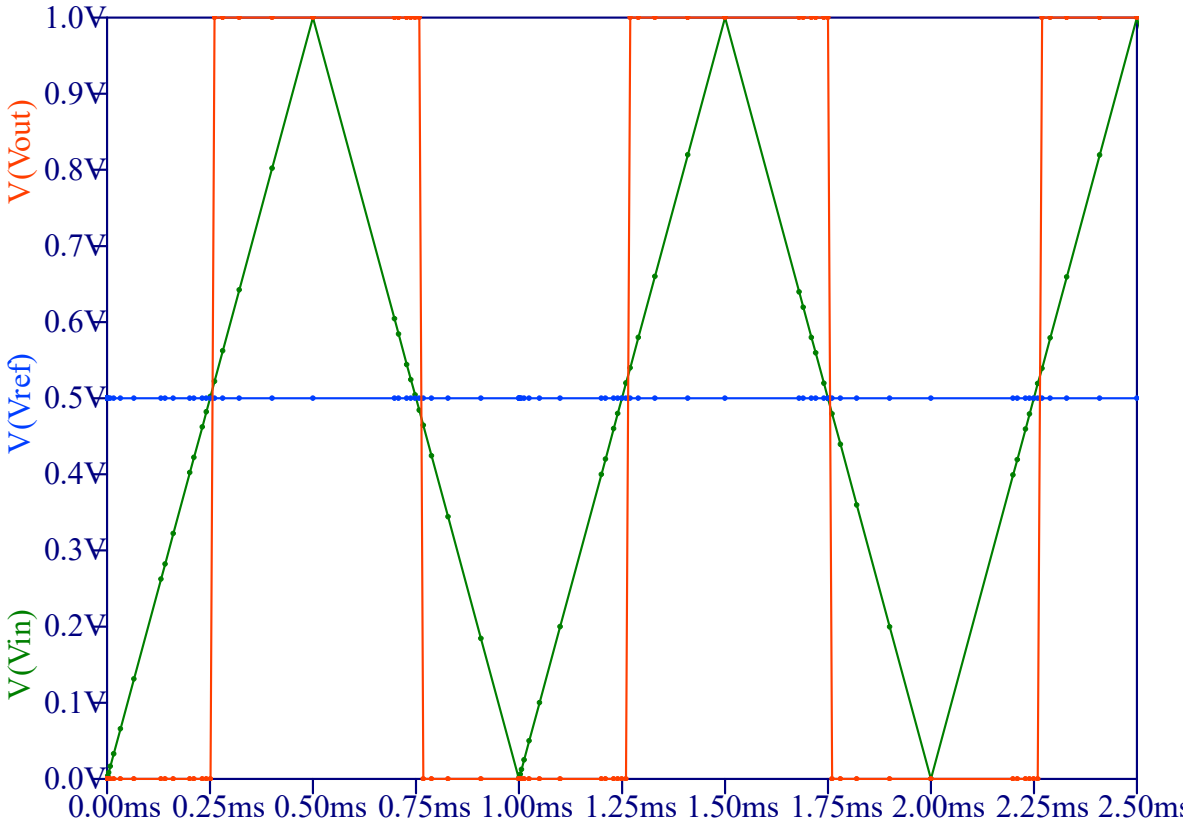

Figure 3 – TTOL=10µs

To sum it up: `Trunc()` provides a way to tighten up the simulation samples around our component's output state transitions.

Trunc() – Parameter Overview

If you compare the evaluation function and `Trunc()` function signatures in `CBlockBasics4`, they are quite similar:

```
void cblockbasics4(InstData **opaque, double t, uData *data)
void Trunc(InstData *inst, double t, uData *data, double *timestep)
```

In the evaluation function, the parameters are:

`opaque`	A pointer to a location to save a pointer to the per-instance data that we allocate.
`t`	The QSpice simulation clock time for the current simulation datapoint.
`data`	A pointer to the `uData` array of input port, output port, and component attribute values.

`Trunc()` is similar but adds a parameter:

`inst`	A pointer to the per-instance data that we allocated in the evaluation function. That is, it's equivalent to *opaque.
`t`	The *proposed next* simulation clock time for the next evaluation function call.
`data`	A pointer to the `uData` array of port and attribute values *that would be present* in the next evaluation function call if `Trunc()` returns *without* changing the value in `*timestep`.
`timestep`	A pointer to a time-step value.

The `timestep` parameter is a bit more complicated. I'll cover it in more detail later. For now, just think of it as a place to return a value to force QSpice to reduce the simulation time-step.

OK, that's probably confusing. Pushing on….

The QSpice Simulation Cycle

We'll need to understand the basic QSpice simulation cycle. During initialization, the evaluation function, `MaxExtStepSize()` (if present), and `Trunc()` (if present) are called a few times. After that, the simulation cycle is something like this:

1. QSpice calls the evaluation function. A simulation step is created, i.e., all circuit values are recorded and become part of the simulation output.
2. QSpice selects a "*proposed* next time-step/sample time-point."
3. QSpice calls `MaxExtStepSize()` (if present). If `MaxExtStepSize()` returns a value smaller than the time-step selected in #2, this becomes the next *proposed* time-step.
4. QSpice calculates circuit values for the *proposed* time-step. This includes all of the input data values for the component at the proposed time-step.
5. QSpice calls `Trunc()` (if present) with the calculated circuit values. The "`t`" parameter passed to `Trunc()` is the *proposed* simulation time-point. The `data` parameter array contains the calculated input values for time = `t`.
6. If `Trunc()` returns a value in `*timestep` that is smaller than the *proposed* time-step, QSpice reduces the *proposed* time-point and returns to #4 to recalculate inputs and call `Trunc()` again.
7. Otherwise, QSpice returns to #1 to process and commit the simulation sample for the proposed time-step.

If you prefer a flowchart, see Appendix A.

The above is somewhat simplified. The key takeaway is this: Each time that `Trunc()` is called, the circuit state (including the `data` array input values) is pre-calculated for a proposed next sample time-point.

Yeah, probably as clear as mud. Sorry. Hopefully, the code will bring all of the above together.

CBlockBasics4 Code – Trunc()

The evaluation function is straight-forward: It allocates a per-instance structure and saves the `TTOL` parameter in the structure.

Here is the `Trunc()` function implementation for reference:

```
 1.extern "C" __declspec(dllexport) void Trunc(
 2.    InstData *inst, double t, uData *data, double *timestep) {
 3.
 4.  if (!inst->ttol) return;   // do nothing if disabled
 5.
 6.  if (*timestep > inst->ttol) {
 7.    // Save a copy of the output vector
 8.    double      &Vout  = data[4].d;
 9.    const double _Vout = Vout;
10.
11.    // create temporary copy of instance data
12.    InstData  tmp  = *inst;
13.    InstData *pTmp = &tmp;
14.
15.    // call the evaluation function with the temporary instance data
16.    cblockbasics4(&pTmp, t, data);
17.
18.    // implement a meaningful way to detect if the state has changed...
19.    if (Vout != _Vout) { *timestep = inst->ttol; }
20.
21.    // Restore output vector
22.    Vout = _Vout;
23.  }
24.}
```

Line #4 is there only to make it easy to disable `Trunc()` in this demonstration code. It wouldn't usually be present in production code. (In fact, the time tolerance is often hard-coded in the component.)

The important code is in lines #6 to #23. It:

- Creates a temporary copy of the per-instance data.
- Saves a copy of the `Vout` state in `_Vout`.
- Calls the evaluation function with the temporary per-instance data, the proposed `t` time-point, and the pre-calculated `data` array values.
- If the evaluation function changes the `Vout` state, sets `*timestep` to return our desired TTOL
- Finally, restores the `data` array `Vout` value from the saved `_Vout` value.

This is the "canonical" technique for using `Trunc()` – call the evaluation function to determine if the state would change at time `t` and, if so, force QSpice to choose a smaller time-step by returning a small value (TTOL) in `*timestep`.

Note that the evaluation function "doesn't know" whether it's being called directly by QSpice (when a simulation sample would be recorded/committed) or by `Trunc()` (when a simulation sample is not being recorded/committed). That's why we must preserve/restore the `data` array contents in the `Trunc()` call to the evaluation function.

To confuse things further, we could have used a “non-canonical” technique in `Trunc()`. And the `*timestep` value might not be exactly what you expect.

We’ll cover those bits in yet another paper.

Conclusion

We’ve covered how `Trunc()` works in painful detail. We’ve outlined the general QSpice simulation cycle as it relates to C-Block components. We’ve seen how a “canonical” `Trunc()` function works.

In a future paper in this unplanned series, we’ll investigate the `Trunc()` `timestep` parameter’s unexpected behavior and explore “non-canonical” uses for `Trunc()`.

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

Appendix A

Trunc() Flowchart