Data I/O Simulator Commentary and Applications Notes

Introduction

While creating and testing the DevSim data I/O simulator it occurred to me that it probably deserved some additional notes and comments beyond what was already in the book. Not so much because it's something amazing, it's really not, but because there are so many possibilities for ways in which it can be used. This document is intended to provide additional information regarding the possible applications of the DevSim simulator, and hopefully it will answer questions that aren't covered in the book.

The source code itself is, of course, the ultimate arbiter. I've attempted to be somewhat verbose with comments, because although DevSim looks simple, subtle things are going on that might otherwise be overlooked. As I state in the text, this is provided as a starting point for your own simulators, if you choose to use it.

The DevSim model

Looking at the diagram in Figure 1 (which is the same as in the book) we can see that everything on the left-hand side is an input of some sort, and everything on the right-hand is an output. In between we have data routing, user-defined function processing, noise injection and some scaling functions.

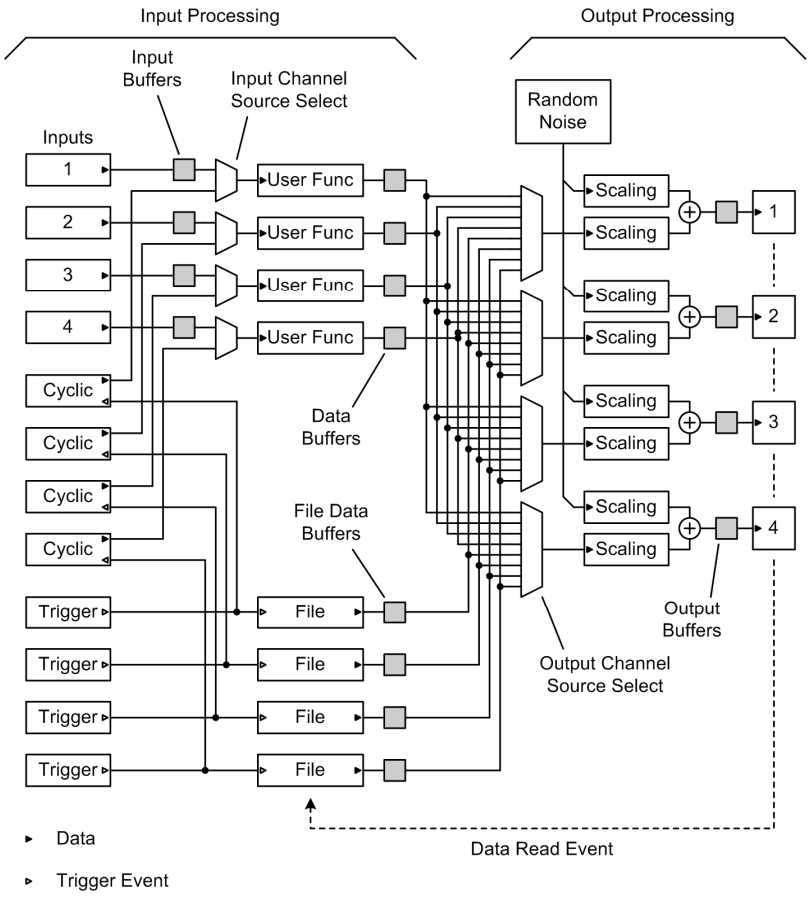

Figure 1 - DevSim Internal Architecture

An input may be one of (a) external data, (b) cyclic data, or (c) data from a file. Data from an input will appear at an output when DevSim completes a processing cycle.

DevSim is based on a model of data objects with sequential synchronous data transfer relationships between them. Data moves from the inputs (external, cyclic, or file) to the outputs, through a series of simulated MUXs (multiplexers). Along the way it may be modified by a user-defined function, scaled prior to output, and have random data values (noise) injected into it.

The basis of DevSim is its data objects. These are initialized in DevSim's `__init()__` method. You might want to take a moment or two and read through the code, which is heavily commented. In DevSim list objects are used as both single-variable buffers and as MUX simulations. The buffers hold data as it moves between the various sections of the simulator, and the MUXs determine how the data is routed between the inputs and the output.

Figure 2 illustrates the binding of input and output in DevSim.

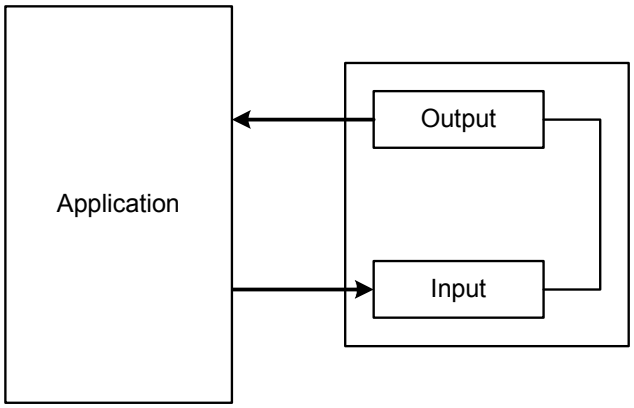

Figure 2 - DevSim Input-Output

But, a real I/O device will have an architecture more like the one shown in Figure 3. Here we have a clear distinction between the software API and the interface hardware it is controlling.

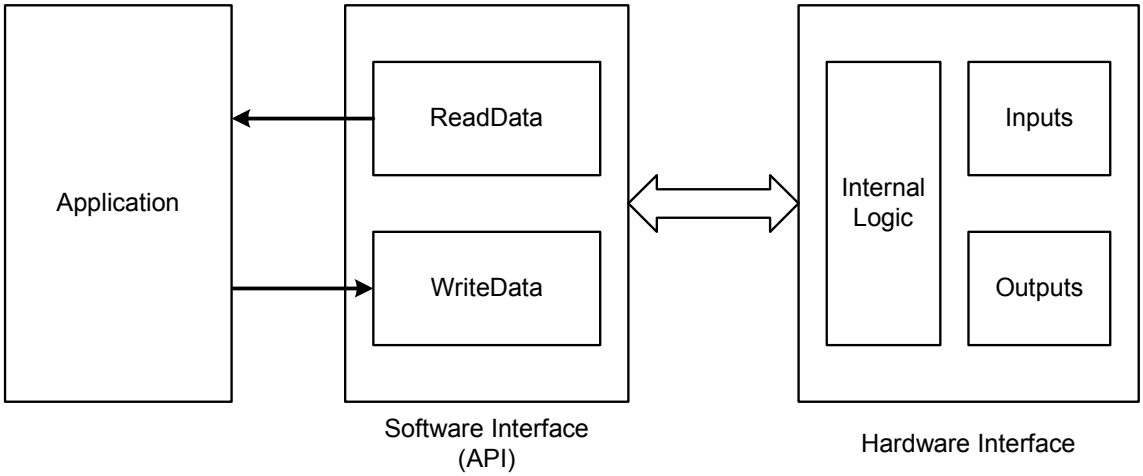

Figure 3 - Real I/O Device Architecture

In order to make DevSim emulate the architecture shown in Figure 3 we need to use a second input-output channel, and add some code to simulate the hardware. It doesn't have to simulate the hardware interface, just the data that we would expect to see move between the API and our application. Figure 4 shows such an arrangement.

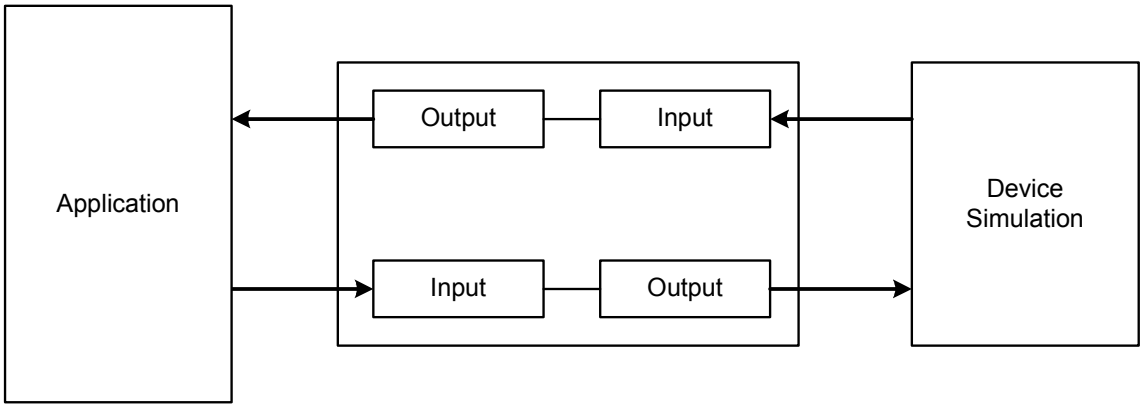

Figure 4 - DevSim with external simulation

This scheme allows for two channels of I/O with some external simulation code. That may not seem like a lot, but because DevSim is easily expandable there's no reason you couldn't have 8, 16, 32 or more input-output channels. I originally kept it to just four to keep things simple and easy to deal with.

Cyclic Data Generators

DevSim has four cyclic data generators, each of which is capable of generating a pulse, ramp, triangle, sine, or constant output. These can be used to drive the input of a function, such as, say, a filter, in order to examine its response. Figure 5 shows an arrangement where DevSim has been configured to generate cyclic data, process it with a user-supplied function, and then drive an external algorithm. The output from the algorithm is sent back through DevSim, where another user-defined function might be applied, and then the result is displayed in real time using a tool like gnuplot. One could also use matplotlib for this, and there are add-on graph widgets available for the Tkinter GUI toolkit as well.

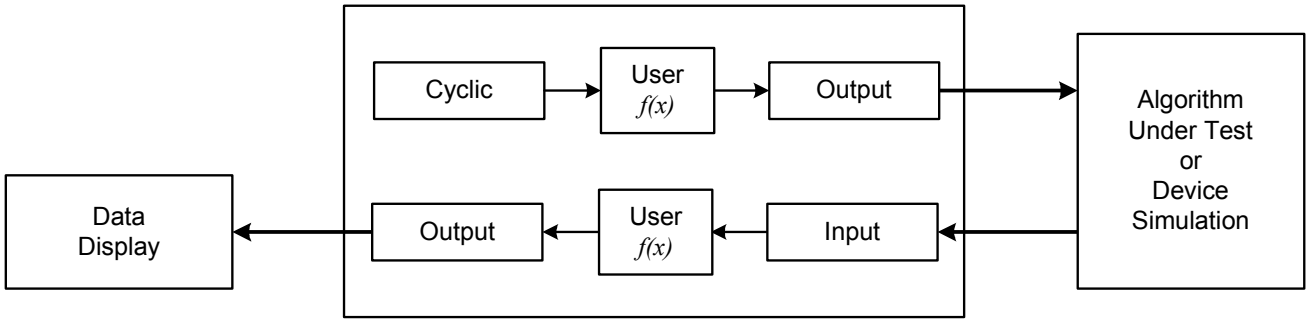

Figure 5 - Cyclic Data Source

The main loop of the simulator, the code for which is described in Chapter 10, controls the step-wise movement of data from an input to an output. The loop diagram shown in Figure 6 shows how the cyclic generator runs asynchronously as a separate thread, with the only point of interaction with the main loop being the output buffer of the cyclic generator. T_s is the main simulator loop time, and T_c is the cyclic generator time.

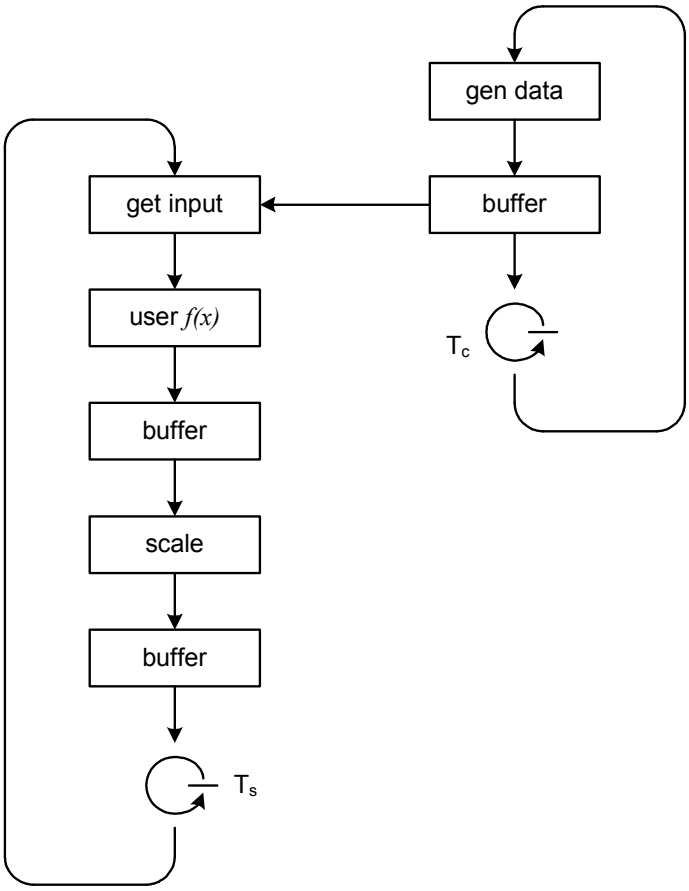

Figure 6 - DevSim Execution Loops with Cyclic Data Generation

Setting the cyclic time will sometimes require some fiddling to get it right. In the test scripts you can see how I let the cyclic generators run at a much faster rate (100 ms) than the main simulator cycle time (250 ms), and ended up with some decent waveforms.

DevSim Speed

DevSim isn't the speediest thing, mainly because there are a lot of `time.sleep()` calls in the code to give other threads a chance to run, and to help keep things in sync. The times were selected on a more-or-less arbitrary basis. It should be possible to speed up the simulator by adjusting the variable delay times.

Starting DevSim (or, how it should have been done to start with)

The original plan for DevSim was to give it the ability to wait for a start signal before releasing the main loop. The idea here was to allow the various internal parameters to be set before starting the main loop. Unfortunately, the code to do that didn't make it into the book. Achieve this I've modified things slightly so that the main thread, __simloop(), contains a while loop to wait for the startSim flag to go true before release the main loop. If you examine the test scripts you will see how this is used.

Demonstration Scripts

There are seven demonstration scripts provided with DevSim. These scripts exercise most of the functionality in DevSim, and will hopefully serve as useful examples.

TestDevSim1.py – External Data Input

This is a simple script that defines the input-output relationship for a single channel and then pushes data into the channel. The main things to note here are how the simulator is configured prior to the release of the main thread, and how it is stopped when the test is complete.

TestDevSim2.py – ASCII Data File Input

Reads data from an ASCII file and sends it to an output channel.

TestDevSim3.py – User-Defined Function

This script is identical to TestDevSim1.py, except that now a user-defined function has been inserted into the data path. The function itself doesn't do a whole lot, but it does demonstrate how this feature is supposed to work.

TestDevSim4.py – Random Noise Injection

The same as TestDevSim2.py except that noise, in the form of random numbers, is applied to the data.

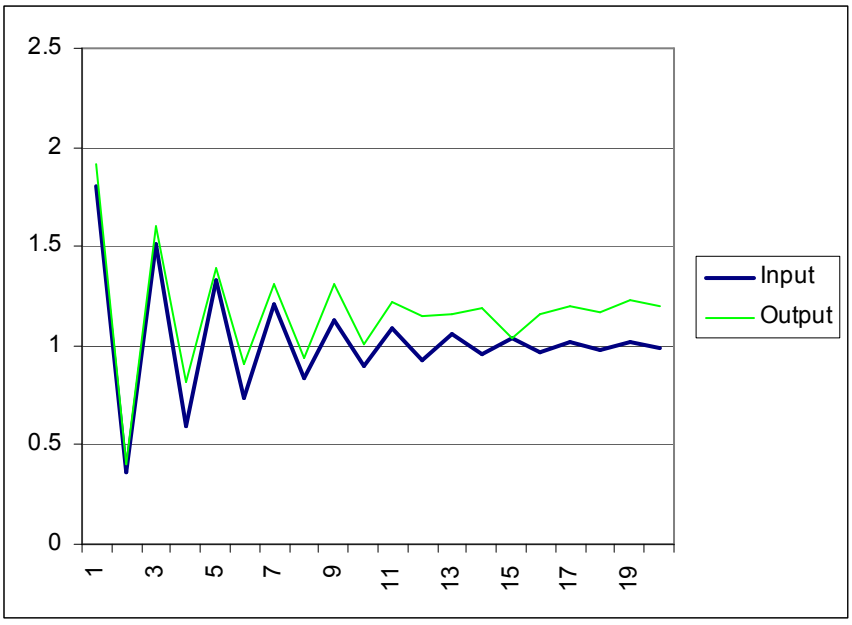

Figure 7 - Random "Noise" Injection

TestDevSim5.py – Multi-channel Operation

This script exercises all four channels. The first channel simply echoes input data to a specified output. The second channel applies a user-defined function to the input data. The third channel scales data read from a file. The fourth channel applies random data to simulate noise.

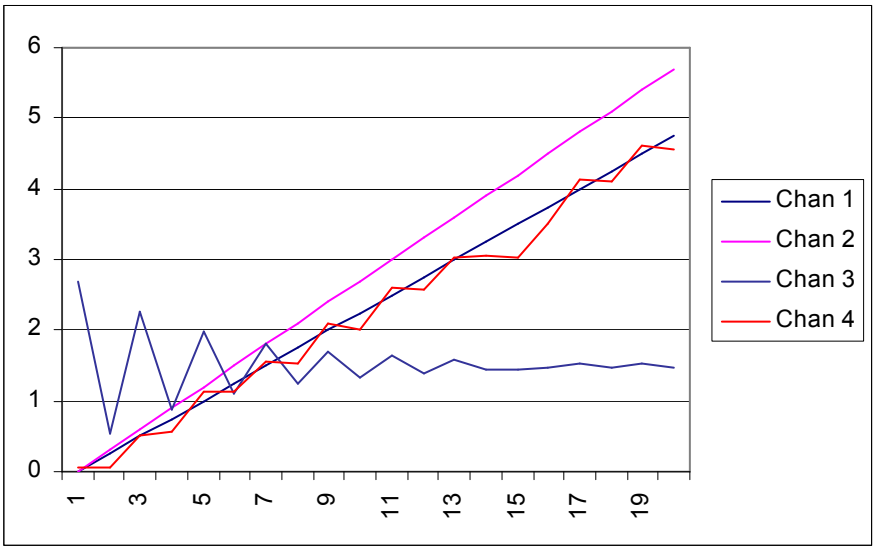

Figure 8 - Multi-channel Inputs

TestDevSim6.py – Cyclic Data Source

Demonstrates generation of a sine wave output using a cyclic data generator.

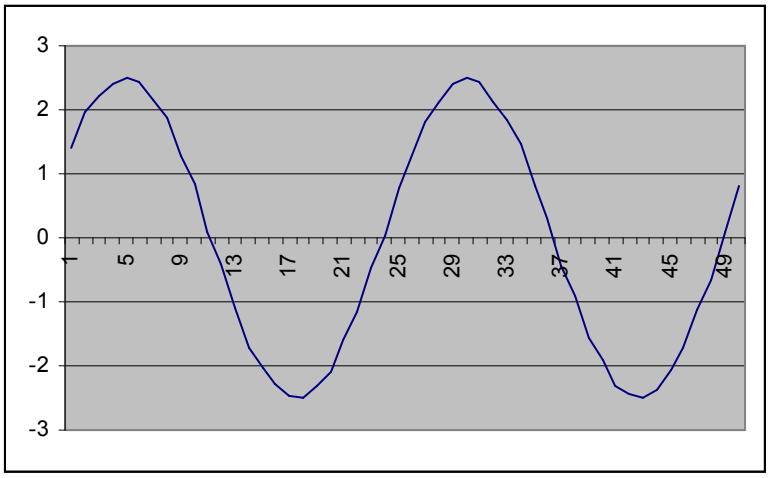

Figure 9 - Cyclic Sine Wave Generation

TestDevSim7.py - Four Simultaneous Cyclic Sources

The script TestDevSim7.py activates all four of the cyclic data sources at one time. The output, when captured and plotted, looks like Figure 10.

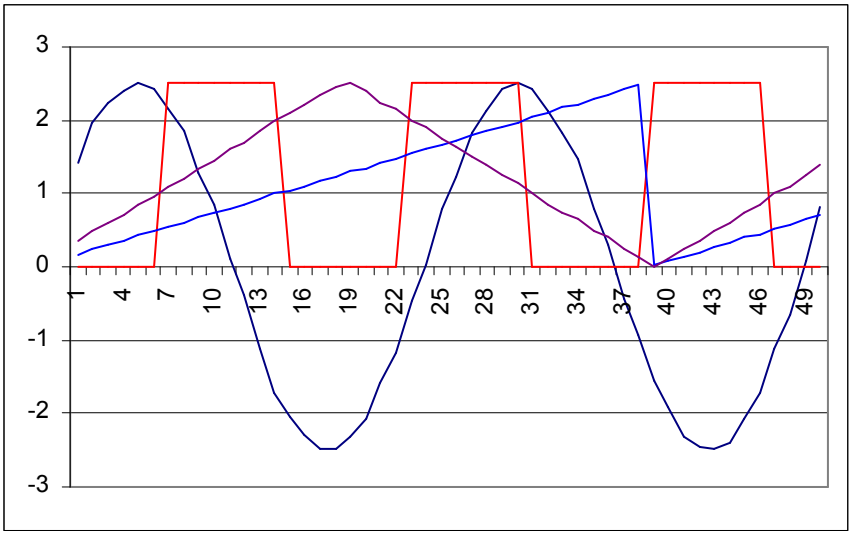

Figure 10 - Four Simultaneous Cyclic Sources