Embedded.com

Hardware-in-the-Loop Simulation

By Martin Gomez, Courtesy of Embedded Systems Design
Nov 30 2001 (8:21 AM)
URL: http://www.embedded.com/showArticle.jhtml?articleID=15201692

Expensive, fragile, and unique systems are hard to test. You know the first releases of the software embedded in them will fail, but how? Hardware-in-the-loop simulation can substantially lower the cost of finding out.

Perhaps you've encountered a situation similar to this: you've created a system using an embedded computer, and you've started testing it by feeding it artificial inputs. If the inputs were analog signals, perhaps you wired a potentiometer, or a power supply, into the inputs to allow you to enter any value you like. While varying the inputs, you measured the outputs, to determine if they were appropriate for the present value of the input.

So far, this is not a particularly challenging scenario for an embedded software engineer, even if armed only with a voltmeter and a power supply. Suppose, however, that the embedded system you're testing is a bit more sophisticated. There are plenty of systems, after all, in which the output is not simply a function of the present inputs, but is instead a function of the present inputs and some combination of past inputs. Equally common are systems in which the outputs are both numerous and each a function of several inputs.

How do you conduct meaningful tests of such a system? In many cases, you can put breakpoints into the software so that it pauses after each cycle through the calculations. You can compare the actual outputs against the value you expected, given all the present and past inputs. You're still faced with the challenge, however, of giving it combinations of inputs that make sense, both relative to one another, and relative to their past values. After all, of all the possible combinations of inputs, only some are "legal." How do you generate those legal test vectors?

You may have additional challenges. How do the digital delays in your software affect the operation of the actual system? You can't easily measure such effects if you insert breakpoints with a debugger. To further complicate things, perhaps your system includes a peripheral, sensor, or actuator, which itself has an embedded computer in it. Can you stop it to measure the output while it's stationary? Probably not. It's part of the "world" as far as your embedded system is concerned, and you'll have to deal with it in real time.

An example

Many airplanes, missiles, and unmanned aerial vehicles have sophisticated autopilots. In most modern applications, autopilots are implemented in software. The inputs to a typical autopilot might be the airplane's current airspeed, the desired or commanded airspeed, the pitch angle (the angle of the nose above or below the horizontal), normal acceleration (commonly known as "g-force"),

and pitch rate (the angular velocity at which the nose is rising or falling). The output (simplistically) might be the deflection of the elevator (the flap-like surface that pushes up or down on the tail). The control law written into the software is designed (in this example) to hold the airspeed at a desired value. In other words, if the airplane is flying too slowly, the autopilot will try to lower the nose by deflecting the elevator downwards, and vice versa. Note that the laws of physics will enforce certain relationships between airspeed, pitch rate, and so on. For example, pitch rate is the time derivative of pitch angle, pitch rate times airspeed equals normal acceleration, and so on.

The relationship between these five inputs and the output is non-trivial. For one thing, the controls engineer is very likely to include an integrator into the control law. This means that if the actual airspeed differs from the desired airspeed, the difference is integrated over time, and a deflection proportional to the integral is applied to the elevator output. This complicates testing. If you turn the system on in the lab with any fixed value of the airspeed other than the commanded one, the elevator will slowly ramp up or down until it reaches the end of its travel. Therefore, you cannot easily measure the elevator deflection and compare it to what you thought it ought to be, because it's constantly changing.

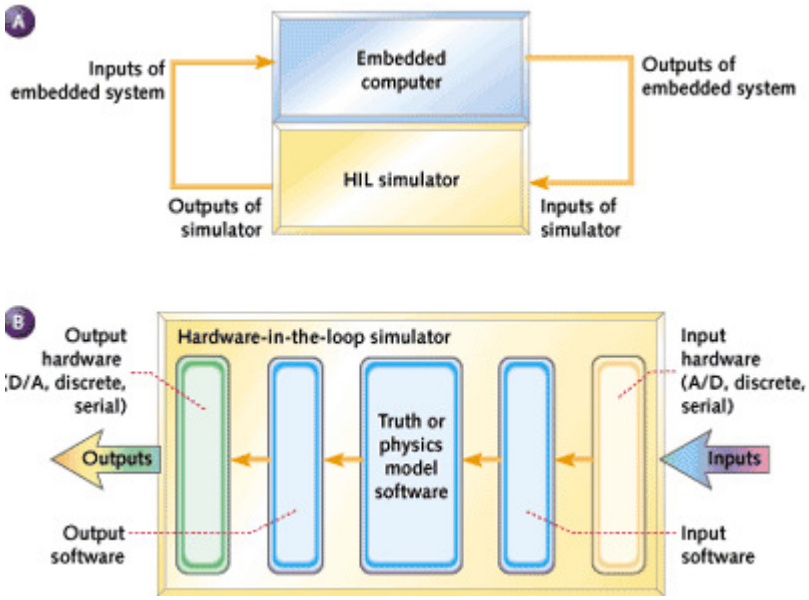

Figure 1
a) Block diagram of embedded system connected to a hardware- in-the-loop simulator;
b) Components of a simple hardware-in-the-loop simulator

A powerful tool often used in this situation is a hardware-in-the-loop simulator (HILS). A HILS is a device that fools your embedded system into thinking that it's operating with real-world inputs and outputs, in real-time. In the autopilot example, it fools the aircraft into thinking it's flying. Figure 1a shows a simple block diagram of an embedded system being tested using a HILS. Figure 1b shows the components of a simple HILS.

The outputs of the embedded system-the elevator deflection, to continue the previous example-are measured by the simulator's electronics. Let's postpone discussion of how that's done, because it's

a major architectural decision in the design of a HILS. For now, let's assume that the simulator simply measures the voltage that the autopilot computer would send to the elevator servo, if it were in actual operation.

The software running on the HILS calculates what the airplane's reaction to that elevator deflection would be. It must therefore include a physics model (also known as a "truth model"), which "knows" the airplane's mass, moment of inertia, and aerodynamic characteristics, as well as the equations of motion. The results of that calculation-the new value of the airspeed, pitch, pitch rate, and normal acceleration-are turned into analog signals, and fed back to the embedded computer. I'm using analog signals in this example, but there is no reason why they can't be serial data streams. Indeed, many aerospace applications use MIL-1553, ARINC-429, RS-422, or other protocols for box-to-box connections. Figure 2 shows the block diagram of the resulting system. Note that the commanded airspeed is a user input; it is not driven by the simulator. When testing the autopilot, the test engineer would drive this input to "tell the autopilot what to do," and use the simulator's data logging features to determine how well it did it.

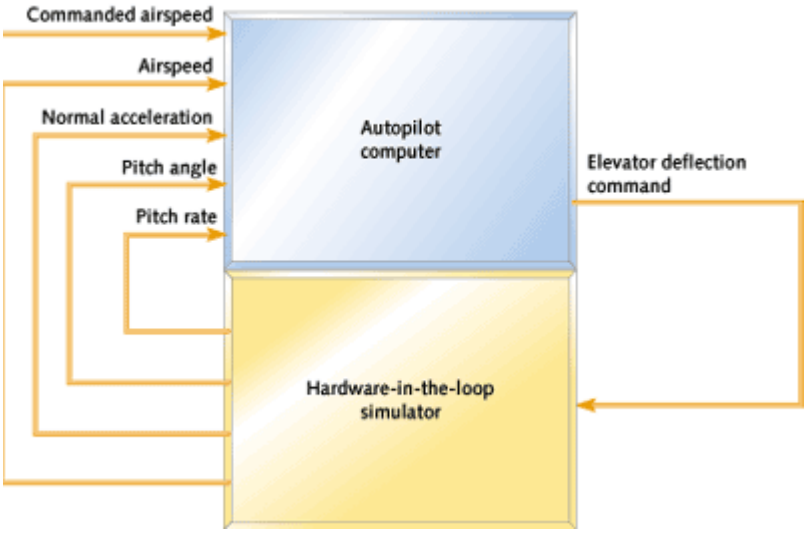

Figure 2 Autopilot being tested using a HILS

Why "hardware-in-the-loop"?

Now that we have PCs on every engineer's desktop, simulation has become a common tool. Many engineers will either write a custom program to simulate the behavior of their product, or use an off-the-shelf tool. For instance, the controls engineer who designs the autopilot's control algorithm will very likely simulate it in MATLAB, or with a custom C program running much faster than the real system on his PC. Similarly, many embedded software engineers first test the embedded code by running a reasonable facsimile of it on a desktop PC before porting it to the final 8-bit microcontroller (or whatever). They "fake" the embedded system's I/O and environment with custom code executing on their PC. There are three key differences between such a simulation and a HILS. First, the output of simulations are just squiggly lines plotted on a graph, not hardware signals. Second, the HILS runs in real time, and third, in a HILS the embedded software runs on the "real" hardware that you will eventually build into your product, not a workstation.

Digitizing and iteration rate

An experienced embedded software engineer, upon looking at Figure 2, will immediately jump to the correct conclusion: the HILS software has to iterate faster than the system under test, and the analog signals have to be digitized with a higher resolution than the system under test uses. How much faster? Hint: it's not just twice as fast. That myth is due to an incomplete reading of the sampling theorem. The "twice the rate of the highest frequency component" rule requires that you take each sample and multiply it by a scaled and shifted sin(wt)/wt function.1 In practice, try iterating five to 10 times as fast as the embedded system iterates. Given the fast computers you can buy nowadays, it's seldom worth doing all the extra math required by the sampling theorem. There are exceptions to this, but fortunately, I've never had to deal with one. Your controls engineer will doubtless express strong opinions on the subject.

Implementation

Now that we understand what a HILS is, and why you might need one, how do you make one? Unfortunately, "make" is the key word-I've never seen an off-the-shelf HILS for sale, although as we'll see, a couple of products come close. In the mid '90s, I led a team that designed a HILS from scratch for our company's internal use. It cost slightly over $100,000 to design and build the first unit. It had approximately 100 inputs and outputs and could iterate a fairly sophisticated model at 70Hz. The second identical unit cost about $25,000 to build. This was considered a bargain compared to the multi-million dollar unmanned aerial vehicles (UAVs) we were developing-if the HILS prevented the crash of just one UAV, the company would get its money's worth. There was another, even more valuable benefit: a HILS allows software to be developed and tested without waiting for all of the actual hardware to be built (or in this case, built and flown).

Start with the signal list of your embedded product. This is a list that includes all the inputs and outputs that the embedded system has and describes their range, resolution, and sample rate. (As an aside, a signal list is not a bad place to start any embedded project, since it forces the software and hardware folks to have a long, sincere chat.) Table 1 is a typical example.

Table 1 Signal list for the autopilot example

Signal Name	Range (V)	Range (physical units)	Resolution (units)	Sample Rate (Hz)	Input or Output
Airspeed	0-10	0-200 (m/s)	0.1 m/s	5	Input
Commanded_airspeed	0-10	0-250 (m/s)	0.25 m/s	10	Input
Pitch_rate	0-10	-2-2 (rad/s)	0.004 rad/s	10	Input
Pitch	0-10	0-2p (rad)	0.006 rad	10	Input
Norm_accel	0-10	-10-10 (g)	0.02 g	10	Input
Elev_cmd	0-5	-25-25 (deg)	0.05 deg	10	Output

Table 2 Signal list for a HILS to test the autopilot

Signal Name	Range (V)	Range (physical units)	Resolution (units)	Sample Rate (Hz)	Input or Output
Airspeed	0-10	0-250 (m/s)	0.06 m/s	50	Output
Commanded_airspeed	0-10	0-300 (m/s)	0.08 m/s	50	Output
Pitch_rate	0-10	-2.5-2.5 (rad/s)	0.001 rad/s	50	Output
Pitch	0-10	0-2p (rad)	0.002 rad	50	Output
Norm_accel	0-10	-15-15 (g)	0.008 g	50	Output
Elev_cmd	0-5	-30-30 (deg)	0.02 deg	50	Input

The HILS's signal list should end up being a "mirror image" of our embedded system's signal list. Table 2 shows that. Note that the simulator's inputs and outputs have a slightly wider range than the autopilot's outputs and inputs, and that they have better resolution.

The choice of hardware platform is dictated by a host of factors. Is there legacy software you want to reuse? For instance, do you already have a simulation up and running that simply lacks the hardware I/O? Are there ruggedness or environmental issues? How much I/O is available? By and large, a HILS is a laboratory device, so environmentals are often not a driving factor. These tools are usually operated by knowledgeable staff-the developers of the embedded system's hardware and software-so ease of use is also a secondary concern.

I've seen simulators built on PC, VMEBus, and proprietary platforms. Each has its advantages; to recommend one here would be meaningless. As with many such decisions, logistical, political, and administrative factors carry at least as much weight as the technical issues, and these vary from one organization to the next.

Two somewhat off-the-shelf platforms are worth mentioning: dSpace and National Instruments' LabView. dSpace (www.dSpace.de) provides what is essentially a PC chassis with one or more of Texas Instruments' DSPs plugged into the backplane. The PC runs the user interface and data logging code, and the DSP runs your simulation and the analog I/O. If you need to run a very fast hardware-in-the-loop simulation, the number-crunching power of a DSP will fit the bill. There are plenty of applications that have to iterate at a kilohertz or more. A Pentium, PowerPC, or other general-purpose processor is hard-pressed to compete with a dedicated DSP at such rates. dSpace has connections to MATLAB and Simulink, which your controls engineer will appreciate. These connections allow a simulation to be written in MATLAB on the PC and then run in real time on the DSP with much less hand-coding than if you wrote it from scratch in C.

LabView (www.ni.com) is a popular front-end for a large family of analog and digital I/O boards. Using a graphical user interface, you can build virtual instruments, and then connect them to simulate the world that your device-under-test lives in.

I refer to dSpace and LabView as "somewhat off-the-shelf" because you still have to customize them to your embedded system's I/O, and you still have to write the simulation code, albeit at a higher level than if you were doing it strictly in C.

Usability

One issue to consider, regardless of the platform, is reusability. Yours may not be the last project to use a HILS. Perhaps many of the requirements are common across projects. All will require analog I/O, discrete I/O, a fast processor, and so on. Why not try to make the common parts, well, common? In my HILS, this was taken a step further. The I/O portions of the code, as well as the interprocess communication mechanism, were common across all versions of the simulator, and were borrowed from the UAV flight computer's software, which was, in turn, shared by the ground station software. The same HILS was used to test four vastly different UAVs, with only minor changes to its software.

The details of reusability are best left to another article, but I must stress one key element: I/O drives a simulator design. No matter what you're simulating and testing, you will have to read the

device-under-test's outputs and drive its inputs. They won't be the same from project to project, so your simulator's design should be reconfigurable. Consider designing the simulator such that the mapping of hardware channels to internal software variables is configurable. For example, on our current project, channel 3 of the HILS's D/A converter might drive the embedded system's airspeed input. On the next project, it might drive a pressure input, or a temperature input, and so on. It would be nice, therefore, if the I/O portions of the simulator's code were table-driven, so that you didn't have to recode them. To extend this wish list, remember that the calibration of the HILS's I/O will vary from project to project too. On today's project, the airspeed signal might be mapped as follows: 1V = 0 knots and 8V = 150 knots. Next time, you may have to test a system with a very different calibration. Again, this can be table-driven. If you do this thoroughly, all you need to do to adapt the HILS's I/O to the next project is change the table. If your simulator hardware also allows itself to be reconfigured (SBS/GreenSpring's IndustryPacks work wonders for this), your simulator will earn its keep across many projects.

Architecture

When designing a hardware-in-the-loop simulator, an important question is "how much hardware do we put in the loop?" At a minimum, the embedded system's computer has to be in the loop, since its software is being tested. In many cases, however, that will not be enough. As with many design decisions in our profession, this one requires hardware versus software trade-offs. If you want the embedded software to see real inputs, to "think" that it's operating in the real world, and to have the world react properly to its outputs, you have to either include or simulate any hardware that stands between the embedded system and that world.

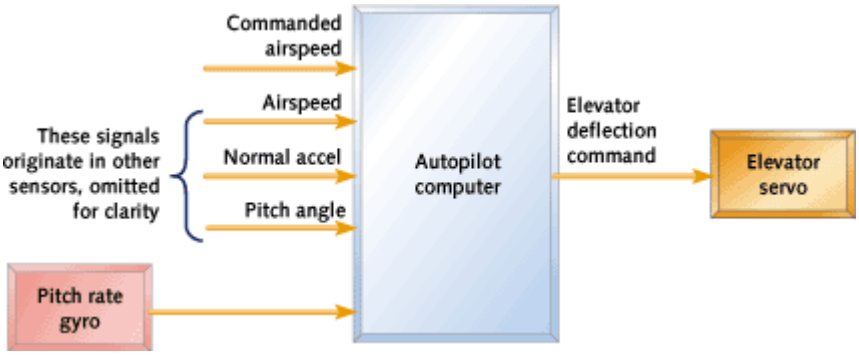

Figure 3 Block diagram of simple autopilot under test

This is best illustrated by continuing with our autopilot example. Let us consider what other components are built into our autopilot system in addition to the computer that runs our embedded software. In Figure 3, we've expanded the previous block diagram by including a sensor (the pitch rate gyro) and an actuator (the elevator servo). We'll omit the other sensors for simplicity.

The pitch rate gyro consists of a sensor that measures the aircraft's rate of rotation about one axis, in this case the pitch axis. How it works is not relevant to this discussion, but let's assume that the gyro's behavior can be characterized as follows:

- It has no noise in the frequency band of interest.
- Its output is mostly linear-it produces a voltage proportional to the pitch rate.

- Its only nonlinearity is a small bias-the output is a small non-zero number when the actual pitch rate is zero.
- To compensate for the bias (which would cause an autopilot to perpetually try to correct a non-existent rotation), the manufacturer has included a high-pass filter. This allows the pitch rate due to turbulence or maneuvers to pass, but blocks the DC bias.
- Its bandwidth is more than enough to sense the rotation of our hypothetical airplane.

The elevator servo is a bit more complicated. It consists of a motor, which uses cables and pulleys to deflect the elevator up and down. It also has a feedback mechanism, consisting of a potentiometer that tells a servo amplifier the elevator's current position. The servo amplifier will command the motor to deflect the elevator until it's at the position the autopilot commands. Let's assume that the elevator servo can be characterized as follows (and yes, all of these assumptions may stretch your imagination a bit, but bear with me-this article is about HILS design, not autopilot design):

- It is mostly linear-it produces a deflection proportional to the input voltage.
- It has a bit of a deadband-in other words, it doesn't actually move until the commanded deflection differs from the present deflection by a small amount.
- It has a finite slew rate-it can only move so many degrees per second.
- It has a finite small-signal bandwidth-if asked to move back and forth a degree or so, it can only do it so fast.
- It has a great deal of torque, so much so that there's no question of it being capable of deflecting the elevator under any conceivable load.

Our hypothetical controls engineer does not think he can write a model of the elevator servo with enough fidelity to do the job. Even if he could, it has enough non-linear behavior (finite slew rate, deadband) that it would take quite a bit of effort to model it in software. He does think, however, that the gyro can be adequately modeled. Depending on how much the servo costs, it might then make sense to include an actual servo in the hardware-in-the-loop simulation, and to model the gyro in software. Figure 4 shows the block diagram of this configuration.

Figure 4 Block diagram of a mix of real and simulated components

View full size image

Before you scoff at the notion of including expensive hardware in a software simulation, consider the relative costs. We're assuming that this HILS is a tool, not a product, so we're only going to make a handful of them-we're not considering mass production. Thus the labor to implement a servo model in software might cost more than simply including the servo hardware in the simulation. (This philosophy is admittedly biased by my career in the U.S. When I worked in Argentina, salaries were very low and hardware was frightfully expensive, so the decision there might have been different.)

As shown in Figure 4, the HILS must now read the deflection of the servo from the feedback pot, rather than from the autopilot computer's output, as it did in Figure 2. This allows the servo's hardware to display all of the characteristics we're interested in: slew rate, bandwidth, deadband, and so on.

However, because we chose not to incorporate an actual gyro into the simulation, we must include software to model one. Not only is this a fairly easy device to model (given the simplifying assumptions we made about its behavior), but it's a lot cheaper than the alternative. Remember, the gyro senses rotation, so to physically stimulate it would require that you purchase or build a device that will rotate the gyro under command from your HILS. You can buy such a machine-known as a rate table or gimbal-from a number of companies, for a small fortune.

A portion of the software, therefore, will be devoted to turning the simulated airplane's pitch rate (the "truth model") into a sensor output, given the behavior we've assumed for the gyro. It may be wise to make this a modular block of code, since you can then use the simulator to test various sensors, without changing the simulation code itself. The HILS would then have a block of software for every sensor (and, if we choose to model the actuators, for every actuator).

Both of these decisions-to model the sensor in software, and to incorporate the actuator hardware-involve trade-offs. For one thing, no software model is perfect. The equations that govern even a simple device can be very complex, and there will always be effects-hopefully small-that the model either doesn't incorporate or calculates inaccurately. On the other hand, including the hardware in the simulation is no panacea either. Does the device work the same on your lab bench as it will in real life? We assumed our servo would have a huge amount of torque, but in real life the torque is finite and the load is large, because the actuator has to move a massive object and overcome aerodynamic forces. Thus it will not move the same in flight as it does in the lab.

Data logging

The purpose of the HILS is to test an embedded system. The proof that the embedded system passed its test is that its outputs were correct for the inputs that it was given. The HILS ought, therefore, to provide data logging capabilities. One easy way to do that, if the required throughput is not excessive compared to the computer's throughput, is the following: at the end of every cycle, write the simulation's state to a file. By "state" I mean all the inputs to your simulation, its outputs, and any internal values it generates that are used to calculate the next cycle's outputs. If a hard disk is too slow to keep up with your real-time requirements, a RAM disk may be large enough to do the job. In this case, you might want to incorporate a means of starting and stopping data logging (much like the trigger mechanism on a scope or logic analyzer) so that you don't fill the data log with meaningless data. At the end of the test, transfer the file to a program such as Excel or MATLAB and analyze the results. You did build Ethernet into your simulator, didn't you? The volume of data generated can easily exceed what you'd happily transfer with floppy disks. This is an area where an off-the-shelf environment like dSpace or LabView can save you some development effort.

Limitations of a HILS

Lest you leave with the impression that a HILS is a silver bullet, let's list what it can't do. It cannot easily stop-if you pause the hardware-in-the-loop simulator, all the components that it's attached to, including the embedded program in your system-under-test, keep running. We could, of course, put breakpoints or a pause-on-command feature in our embedded software, but that's rather intrusive. If we did that, we would not be testing what we intend to ship. Even then, in the autopilot example above, the elevator servo has a dumb analog control loop built in. It won't stop just because the simulator or our embedded software stops.

A HILS cannot tell you what's going on inside your embedded software; it's not a replacement for an in-circuit emulator, a logic analyzer, or a software debugger. It can only read the embedded system's outputs. When the embedded software goes awry, there may or may not be enough information contained in those few outputs to determine what portion of the software was executing, or what the values of the internal variables were.

Other applications

The vast majority of embedded software engineers work in fields other than aerospace. However, with a little imagination, you can see how a HILS can be applied in other areas. For instance, machine control and motion control are two areas where it's hard to completely test the software before the expensive, fragile, and often unique machine is built.

Before I joined the aerospace field, I designed software for vacuum equipment used in the production of semiconductors. These were million-dollar machines equipped with pumps, valves, robot arms, and vacuum chambers. It would have been relatively straightforward, had I known about hardware-in-the-loop simulation back then, to sense the state of the pump and valve commands coming out of the embedded computer, and then calculate what the pressures in the various chambers would do. We had no such capability, however. Instead, I wrote the software, and tested it as best I could using more manual methods. When the machine was finally coming together mechanically on the shop floor, the schedule was starting to get tight. There were many hardware tests that needed to be performed. Problems with the hardware had to be fixed. Everybody knows "software is much easier than hardware," so it's okay to leave software testing until the very end, right? There ensued a few weeks of late nights, three shifts per day, twice-daily Gantt charts, hourly visits to the shop floor by anxious managers, and all-around misery.

Avoiding the all-too-frequent integration crunch is reason enough to invest in a hardware-in-the-loop simulator. esp

Martin Gomez is a software engineer at Johns Hopkins University's Applied Physics Lab, where he is presently developing flight software for the STEREO spacecraft. He has been working in the field of embedded software development for 17 years. Martin has a BS in aerospace engineering, an M.Eng. in electrical engineering, and is a part time graduate student in Applied Physics at JHU. He may be reached at martin.gomez@jhuapl.edu.

References

1. Proakis, John G. and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications, 2nd Edition.* New York: MacMillan, 1992.

Return to December 2001 Table of Contents

Copyright 2005 © CMP Media LLC