Automated Testing of Embedded Software

Lessons Learned from a Successful Implementation

By Jim Kandler

I've tested several embedded applications both manually and automatically. Right off the bat I can tell you that there are no silver bullets. There is no guaranteed way of doing automation correctly so that you guarantee success. Based on your systems, there are many different ways to succeed or potentially fail. I'm going present to you some of the lessons I've learned. I'm going to give an overview of what needs to be done for Embedded Systems, the similarities between the GUI testing and the Embedded Systems, and a successful approach we've used for more than one application. I've developed my approach from what I've observed from some of the big guys that are making the automation for embedded systems a turnkey tool for people. You can use the same kind of approaches for implementing your own system. If you understand this approach, projects will go that much easier and have less risk and probably save you a lot of time and money.

GUI and Embedded Differences

We're all used to seeing GUI test automation. A number of vendors have tools out that do that. They have a number of things in common. These things are talked about in Elizabeth Hendrickson's paper "Making the Right Choices" published in the STQE May/June edition of 1999. All the GUI systems have a test language. They call it TSL or some other name. Most of them are C-like. Along with the language, obviously, is some kind of an editor. Many times the editors are pretty fundamental. Then there's an engine that drives the testing scripts. There's a debugger that's needed for debugging your scripts. There is a script recorder that is used for recording scripts off the actions that you generate in the GUI interface. There's also a result recorder, obviously we need to capture the results of our testing in a log. Lastly, and the difference between Embedded Systems and GUI, is the interface to the target. Usually Microsoft supplies the interface by way of the operating system for most of the applications that people are talking about here. In an Embedded application there is no usual OS, there is some kind of an RTOS or such, but the interfaces and all those are different and dependent on the hardware. For the GUI applications, we're very used to the OS providing the hooks and links we need for testing. The GUI tools provide hooks into input devices and captures output to the display. You pass the data back and forth through the API's that the will capture the responses from the target. Today this makes GUI testing of applications very easy. They run in most of the operating systems and all the browsers.

Interface to the Target

The Embedded Systems today are similar to the PC based systems of years ago. In the days of DOS, the PC based systems didn't have an API that could be used for transferring data back and forth. We had some very early systems that were not very standardized, common or high tech. They were all custom implementations of some kind of an interface and you needed to provide an interface on your application. The embedded interfaces of today are typically like that. They are very customized and hardware dependent. You will actually end up having to hook into the hardware at some point. Many times this point is a standard connector. Whether it's at an analog signal that's coming in, a serial line or 8 bit bus, or a it may even be a pod that they can put on the processor. You'll need to make the interface in an unobtrusive manner. In other words, you don't want to be corrupting signals as you are monitoring them.

Define the Interface

One of the most important things with the automated embedded testing is going to be the interface that you choose or pick. The interface will define what testing you are able to do. So wherever you put this interface and how you configure it will be very important. It's going to limit and focus the amount of testing or the direction for your automation. Defining the interface is very important to the testing effort. It is so important that we have given the types of interfaces names so that we can understand which is the focus we have for our testing. We determine this focus relative to where the microprocessor and other hardware. With embedded systems, this is important because if the focus is wrong, you will start with a wrong view of the system and the testing that you try to implement will be very unproductive. You probably will not be able to accomplish what you wanted to do and you will spend a lot of time instrumenting the wrong test and hardware.

Type of Interface

The Type 0 interface is only the microprocessor board. The outside hardware does not exist. There are reasons you may want to do this and we'll get into this in a bit. The Type I is where the interface boards beyond the microprocessor exist, but all the outside signals are simulated, so you may have your whole package or your whole system, but the signals to the outside world are all simulated. In a Type II system there is a single board in the system that is missing or is simulated.

To describe these different types for interface systems, we will use a mockup of a real system. We have a robot arm for instance with a microprocessor board that generates output to the robot arm. Causing it to articulate and move. There is an input board that takes the data from the robot arm and tells the microprocessor where the arm is and how much of a load it's experiencing and so on.   It provides feedback to the microprocessor.

Type 0

In a Type 0 implementation, there is no robot arm, there is no input board, there is no output board, all the signals would need to be simulated and captured coming on to and off of the microprocessor board. What we're typically interested in here is verifying the actions and the responses on the board and to the software running on the board. So, we simulate these other inputs and outputs and we are able to actually get this running totally independent of the rest of this system. Also, because we are simulating all the incoming signals, we are able to generate all the error conditions that you won't be able to generate when the other boards are used.   The input and output boards are going to limit what you simulate and we can generate all sorts of erroneous signals and conditions that would not be possible otherwise.

Type I

The Type I system is another one is used when only the use environment is simulated. We may have our input/output boards in the controller, but we don't have a robot arm. There may be quite a number of reasons why you may want to do this. One is, you may have an item on your project that is late, but you may also have a need to simulate this environment because this environment is too expensive to build and run onsite for yourself. You cannot physically run it onsite because it may be too dangerous. You may want to simulate a missile. Obviously we won't be launching missiles around the building here. This may be an application for a nuclear power plant or in some other such plant like a steel or paper mill plant. These systems are many million dollars worth of machinery and we aren't about to turn that into a test bed for ourselves.

Type II

The other cases where you may have a single board in the system, not necessarily a microprocessor board or your main processor, but your input, for instance, may have processors or controllers running on it. You may want to simulate that board as being in the system, but it is not. So, it will allow you to do all sorts of exception handling and generate all kinds of errors here that you couldn't otherwise.

Signal Management

The whole objective here is to put a fence around the area of interest. I've shown you in the diagrams here, how to block off the areas that you are not interested in. Then you start focusing on the signals that transition back and forth across an interface. It is very important that we understand the interface and the signals moving across it.

Now that we've got an idea where the interface is, we can talk about managing the signals. We're going to find that in most systems, there are many signals. There are many input and output signals that are being generated on all of these devices. You will need to design this interface. Making a list as shown here is a great way to start to catalog all the types of signals and characterizes them. Then you can start to actually develop the hardware interface and to figure out what kind of boards and hardware you'll need to provide them. A spreadsheet is a useful tool to do this. It's easy to edit and easy to manage. Once you start to look at input or output cards, you may find that you need six signals and the cards provide eight. I would suggest that you leave extra signals as spares. I would not determine that I need eight signals, then buy a card that has only eight. You will need to have a few extra ports available. If you have made an error in signal allocation, you can then utilize some of the spares.

It's also common to have extra signals that are identified late in the testing effort. For instance, in the case of the Type 0, you may later find you want to provide microprocessor reset lines, other resets, other hardware resets, other power disable lines or power reset lines. So you can end up with a more automated system that is actually able to go in and reset itself and recover from a hang-up, then you can go back and start testing. Those kind of things will lead in the implementation of stand alone testing, so you don't need to have someone sit there in the middle of it while it's running.

Full Simulation

Once you catalog your signals, you need to start thinking about what's actually happening with the signals and the details of the signal going onto the target. Those signals going onto the target must be simulated and those coming off the target must be read and/or captured. In the diagram, a signal is being provided through the driver as an input and an output is coming off the board and through the driver and some decision being made on that signal and whether it's as expected or not. We have found it useful to use small development boards. These development boards are very fast and nice for handling fast and time dependent signals. These boards allow you to handle, through a slow serial link, the faster handshaking that you need or some special signal handling or protocol. This allows you time to provide handshaking and preprocessing on a board that does not overload the rest of your system.

Switched Simulation

I talked about simulated input signals. You may want to think about not fully simulating a signal. The test bed can be configured to interrupt the actual signal and provide the ability to alter or modify it before sending it on to the processor. This is similar to using an API. It would give you the ability to simulate errors and to actually corrupt some of the real time signals that are actually coming through the boards. If the signals you have and are trying to manage are very complex and very hard to simulate, this actually may be a better way for providing that simulation for error generation purposes.

For instance, this may be a very complex analog signal (an ECG) that needs to change over time or some other precious signal that needs to change over time. This input driver could provide us an offset to that ECG as we need it. Otherwise the signal would feed straight through the driver on a microprocessor board and we would not have the signal altered. We have the ability, as we need to generate our exceptions and error conditions that challenge the system.

If you are dealing with a system that has many analog signals that are unique, this is another means of dealing with trying to simulate a very large number of signals. You can use the existing signal and alter it as needed. Otherwise, you may bog down your test bed and you may not actually get it to run effectively. So, this would allow you to only break into those signals that need to be simulated or need to be changed for time. Another consideration is because you have all the rest of your system up and running, you don't need to generate all these other simulated signals to get the base system up and running. You can quickly get all your input cards in place and the rest of your system in place and only break into those signals that you need as your able to develop these tests. You can incrementally build on your test cases and leverage the existing system without having to spend much time generating infrastructure.

This method is also more likely to be used on a more mature where I may have a number of microprocessor boards and input boards where I would have the rest of the environment available to me. I would not necessarily go off and develop these signals from scratch. This would allow me to get started and running very quickly.

Leverage Other's Work

Constructing this kind of a system from scratch is a large task, and is very risky. I would not suggest that you attempt this yourself. Fortunately you don't need to. There are several different options that you have to leverage the prior work of others. There are a number of component suppliers and there are also system suppliers. There are trade-offs to using either of them.

Systems suppliers will do all the system integration and developments work for you and train you on how to use their system. But, you are using their system which leads to limited options for changes later if your supplier is not willing to do it for you if you move off their base system or move onto some special aspects of their base system.

Component suppliers are interested in selling components, so you must actually develop the system and build it; they will not do that for you. However, in the case of National Instruments, which we have used extensively, we found there were many competent consultants available to assist you. These people will do system integration with you. They are very adept and proficient at using these systems.

National Instruments

I mentioned National Instruments several times here, not because I work for them, but because I have been very impressed what they do as a component supplier. Their systems are very well integrated and very mature. For instance, LabView is a quite a mature language. It is on version 5 right now and is a very rich toolset with many icons and pre-built modules that are available for you. You can quickly implement differentiation, high pass, or notch filters. All of which is already built for you and you just need to hook it up to plug and play. This makes it very transportable and usable. It makes it easy for you to bring in a consultant and work on that to supplement and offset your own knowledge in-house.

There is a Test Stand or test executive. This is the engine, the debugger, report generation capabilities and is allows you to do batch processing. It's a high level tool, a mature tool and very robust.

There are also off-the-shelf data cards and analog signal cards. The value here is low cost relative to customs systems. These are available in less than 2 weeks through normal delivery channels with no special orders, no surcharges. Simply call up and give them a credit card # and you have them on your desk in a few days. This makes these cards very portable and robust and standard. These tools from National are very well integrated. As I have said here already, they are real value added tools. As you start to use these tools and build on them, it is simple to make progress.

National is a leading supplier for software in the automated test industry. We are leveraging their prior work for our own software testing purposes. If you invest in National, you aren't going to a fly-by-night company; they have been around for a long time. It is one of the largest and one of the most commonly used suppliers in the automated testing industry. In fact, the others on the list here (as shown) are all resellers of LabView. It interfaces very well with their cards and their tools, also.

LabView is the leading language for programming for the automated test industry. It is also the premier language for data acquisition simulation. This language is the lead for many reasons. You can take advantage of the very good technical support from National and other consultants that are available. The fact that is very well known and used often makes it very transportable. The last thing you want to do is have some of your engineers learning a language that is not transportable and that they'll never use again.

Features

We have seen systems built for software testing that have been redeployed in other areas of the business. Software development is using them to do debugging and troubleshooting. The board manufacturers are using them for manufacturing and testing. I've even seen these systems being used on the burning racks for final system testing. Some of the features of the toolset that we have are with LabView and Test Executive.

You can actually share data between your sequences and your tests so you can define global variable that can be used and shared between them.

There are expressions that can be defined so that you can have logic decisions that are embedded in this. If you want to watch for the watchdog on the microprocessor if it's not being reset, you could actually have it trigger the system to have a microprocessor reset for you.

You can also access variable information and run time from other programming environments to pull in data from other areas. If you want to do some dumb monkey testing, tables of variables can be accessed and run through batches of test scripts. Perimeters can be loaded from a database and this allows you a lot of flexibility for configure your test scripts on the fly and set different perimeters as they run.

Labview is a high-speed parallel sequence engine that is multi-threaded. It is not a single thread tool so that we get very high input and therefore advanced sequencing and looping and branching. So we are not sitting around waiting for one of the sequences to finish before the other one's can start.

We can also call tests that are written in other languages, such as Visual Basic and C. This makes it very transportable and flexible for using other tools that you've already developed. This is also modular and customizable.

I mentioned earlier that it interfaces to a number of the other simulation and data acquisition cards. This is all done very easily through the DLL and API's that are provided.

If we don't log any results, there is very little valued added in testing since we need to capture our results. The test will automatically log the results and determines whether you want to log them as a text file or as an HTML.

Programming Environment

The programming environment of LabView is very rich and we see only a small portion of it here. This is a for loop structure with a timing function enclosed and collecting temperature. The loop runs for 132 readings and gathers data from a knob that is multiplied by a factor of 1000. Again, it's a rich function here and it's very usable because all you need to do is connect the icons.

An Implementation

I've talked about principles behind embedded system automation. Here is one of our systems up and running. In the center of the picture you will see a chassis with a number of cables plugged into a chassis. That is the computer or, as National calls it, the PXI chassis. There is an emulator up on the shelf that is connected through a serial cable to the PXI chassis. On the left is our hardware that is the actual target and the unit under test.

On a closer view, you see all the green cards on the left are standard interface cards from National. The two cards on the right are actually 2 microprocessor boards that are under test. We run one of them at a time, one is a test unit, whereas the other one is a driver. You will sometimes want to use other cards for drivers for signals that are difficult for you to simulate and emulate.

How do I do This?

I've covered the steps about how to design and construct your own system. This is methods is similar to how system vendors do it.

It is very important that you determine the focus of your testing. This is extremely important. If you don't understand the focus on your testing, you can spend time building the wrong infrastructure. You may end up not being able implement the kind of test you are interested in. Be very careful of this and aware of what you are doing when you decide to do implement automated testing.

You need to partition the interface and decide what kind of test you are going to do. You need to detail the signals that are transitioning across that interface and understand which of those you are going to simulate and emulate and what the magnitude and the type of each of them is.

We have generated automated systems with as many as 250 switched signal methods in slide 9. We only modified a small number of signals at one time and the rest of them are real-time signals that are being passed through the system.

Once you’ve detailed out these signals, you need to sit down and instrument each of them and develop the LabView scripts that will generate and manage them for you. What I suggest is that for each of the signals, you generate input and output scripts to generate the input and output. Do that for each of them as a starting point and as proof that you actually have each of them up and running. If you don’t do this, you will have parts of your system instrumented correctly and parts that aren’t. You will not find out about this until you are deep into testing. If you’ve done this, you now have a learning tool and an aide for those who are new to the system who want to see it on a fundamental basis and want to understand it. Once you have these scripts use them as examples and proofs. This is a good set of cases to run to show that the hardware and system implemented correctly. So, there are lots of uses for this kind of example set.

Knowledge transfer is very important because if there is only one person who understands how it works, you are not going to have a lot of people utilizing it and taking advantage of it. It will end up and island and quickly pass from favor and won’t be used.

The more the automated embedded system tester is use the more valuable it becomes!