Solving testability problems of resource-constrained embedded systems with interpreted languages

White paper

MacroExpressions

http://www.macroexpressions.com

If you experienced that test, self-test and diagnostics functionality tends to bloat the precious code space and is never sufficient, implementing such functionality in an interpreted language may be the solution, especially if this code can be downloaded to the target only when needed.

Challenges of compiled-in diagnostic code

Compiled-in test code

- *drives up the hardware costs or drives away useful features*
- *lacks necessary flexibility in updating the test code executed on as-needed basis*

Code executes only once

A typical case is manufacturing support code. During different phases of manufacturing (from electronics assembly to end-of-line testing of the fully integrated device) different tests have to be run. In a sophisticated device, the tests can be sophisticated and must be supported by the device's firmware.

The trouble is, the supporting code occupies code space even though it is never needed again. It feels like wasted code space, doesn't it?

Tests may change or are not known in advance

Quality assurance people want to get to the root cause of failures in field return units. That elusive root cause was probably not anticipated and there is no readily available test to diagnose the problem. Thus, the tests have to be invented on the fly and tried on the device, but the device's firmware cannot support new tests.

Once the root cause is found and happens to be not just a plain software bug, it is often a good idea to upgrade manufacturing tests to cover a potential failure mode. Yet you do not want to release a new firmware for that purpose alone: It would be much more practical to contain manufacturing tests in the manufacturing equipment.

In a less gloomy scenario, the device is not released yet, and the testers are set out to break it by exposing to unusual conditions of events timing and coincidence, processor load, network load and anything else they can come up with. The testers would be grateful if the firmware readily supported their efforts by enabling simulation of various conditions. But the conditions to be simulated may not be known during development.

Some tests execute only on demand from an external tool

It is common to have a set of tests that normally do not run; their execution is stimulated by an external test tool connected to the device. A typical case may be in automotive electronics where a device may detect a failure of certain functionality, and a test tool is employed to identify the failed component with greater precision.

Again, keeping the failure identification code in the device "just in case" appears a waste of code space. And again, update of the test code requires another release of the firmware.

Built-in self-tests can take up a lot of code space

Some tests can be rather sophisticated, especially in regulated industries. They may involve statistical trend analysis and interdependencies of different inputs. For instance, measured wheel speeds of a car could be required to be compatible with the idea that the four wheels are in the vertices of a rectangle.

Complex tests occupy significant code space and drive up the cost of the hardware or drive away useful features.

Summary

Test code which is built into the device firmware

- occupies code space and thus drives up the cost of the hardware or drives away useful features
- does not provide the necessary flexibility in updating the test code executed on as-needed basis

Possible improvements

Downloadable or built-in modules in an interpreted language are a very attractive proposition for implementing diagnostic/test code.

Downloadable test code

The idea is to have

(a) One or more hooks in the firmware to execute any machine code when properly signaled and

(b) A method to download the test code to the device (usually, in RAM)

If the test code is recognized by the firmware, it signals the execution hook to execute the code. The execution mechanism must provide for resource conflicts resolution (for an extreme example, by disabling the resident code completely).

Of course, there are endless variations on this theme (e.g., secure download), but the underlying idea remains the same.

This is a neat solution but not without its own problems, some of which are mentioned below. Not that the problems cannot be solved (they can) but the solution is by no means in casual programming: It requires a very crafty programmer, probably working in Assembler language.

First, downloadable code is very difficult to write and maintain. For instance, it should be position-independent (to let the firmware place it where it feels like) or it has to know the execution address in advance (which requires a good deal of anticipative planning). If it intends to use any services provided by the standard firmware, it somehow has to know where to find them.

Second, the downloadable code is not portable. The end result is machine code, so if you change the CPU in the next generation of the product, you'll need to change all downloadable modules. What's worse, if the agreement on where to find firmware resources changes in the next build (on the same CPU), all related downloadable modules must be changed. Sounds like a maintenance nightmare.

Third, the downloadable machine code may have poor code density. This simply means that the code implementing a sensible functionality tends to be large, especially for RISC machines. This affects the download time and the size of RAM the firmware has to set aside for the downloadable code.

Fourth, there may not be enough RAM to hold downloaded code. Well, tight RAM resources are one thing. Quite a different thing is that on some critters in the CPUs realm, you cannot execute code from RAM (or certain part of it). This may be a show-stopper for this method.

Interpreted test code

It is sometimes known under different names, e.g., p-code. The firmware has a built-in interpreter (a.k.a. virtual machine) of some interpreted language. Pieces of code where the execution speed is not critical are written in the interpreted language and the interpreter knows how to deal with the interfaces to the rest of the firmware.

The hope is that the interpreted code has much higher code density, so the total amount of code space needed for the interpreter and all p-code is much smaller than the code space needed for the same functionality implemented in machine code.

An important special case is when the interpreted language is itself a machine language, as strictly speaking p-code is, of some virtual machine. (In this case the interpreter is sort of an instruction set simulator.) This intermediate interpreted language (a.k.a. bytecode) is compiled from a human-made text source. In such a case, code density is the highest because source text strings are compacted to small numbers. For brevity, we do not make a terminological distinction between the two variants.

Downloadable interpreted test code

This idea naturally combines the two above. A test module written in an interpreted language is downloaded to the device and executed (interpreted) by the built-in interpreter. This plan promises the following benefits:

High code density of the interpreted code requires less RAM to be set aside for the downloadable code. Moreover, from the CPU architecture perspective, the interpreted code is actually *data*, so it can be executed from any addressable area. Additionally, high code density provides for smaller download times.

Code in interpreted language is much better portable (provided that you can port the interpreter). Besides, rigid requirements on the format of the code disappear: the interpreter takes care of locating the device resources. So, maintenance problems are greatly reduced.

And of course any built-in interpreted code can be executed by the same virtual machine.

This approach, therefore, is very attractive for implementing diagnostic/test code.

What to look for in an interpreted language

When selecting the interpreted language, consider, in addition to code sizes,

- *Input/output virtualization*
- *Execution model of the virtual machine*
- *Debugging support*

Interpreted languages and their virtual machines are created with different philosophies in mind and target different application areas. So when choosing a language to implement test modules, whether downloadable or built-in, certain properties must be understood and weighed.

Hardware interfaces (I/O) and portability

This is a crucial issue for any test module. How does a virtual machine know how to set or read a status of an I/O pin high or low or how to send a byte over a UART or SPI interface of your microcontroller?

There are two basic approaches.

First: I/O interfaces are part of the virtual machine which must be *ported* to your microcontroller.

Second: I/O interfaces are virtualized *outside* the virtual machine. Roughly speaking, the virtual machine sees inputs and outputs as port numbers to read from or to write to. This approach requires a very small set of native (machine code) functions mapping virtual port numbers to physical inputs and outputs.

The second approach is preferred for the following reasons:

- porting a virtual machine becomes a matter of recompiling it
- your test module is portable verbatim provided you made virtualization of I/Os compatible with the previous platform

- native implementation of virtual I/O allows you to encapsulate resource protection
- a virtual I/O may not map to a peripheral at all: it can map to a memory location, semaphore, mutex etc. for added flexibility in resource protection

Execution model, memory management and multitasking considerations

A fundamental question to be answered in selecting the interpreted language is whether the execution model of the virtual machine facilitates what you set out to do, or whether it stands in the way.

For executing testing modules in particular,

- Where does the VM get its working memory from? Will it manage the memory buffer supplied in VM initialization call or will it require you to provide dynamic memory allocation mechanism? Some embedded coding standards ban (standard) dynamic memory allocation, and for sound reasons, too.
- Can you start several instances of the VM and simultaneously execute different test modules in different contexts of your choice? Ability to do this is extremely important. E.g., in studying how robust your task schedule is, you could simply add test code producing delays in various contexts and register the results. All this without modifying the embedded code.
- Can you execute a module piecemeal, with repetitive calls? This ability is crucial in the absence of preemptive task scheduling in your system (i.e., if you have simple executive or cooperative scheduler). Another example – watchdog-related – is considered below.

Example: the watchdog

Watchdog is a device that resets the system if it is not "tickled" every so often according to a defined protocol. The basic idea is that the code checks whether the system state is OK and tickles the watchdog if it is. The complexity of this is in the strategies used to determine if the system is OK. These strategies vary and are application-dependent, and it is best to keep their innards away from a portable virtual machine.

The problem is, interpreted code is slow and may, in any case, wait for an event, so the virtual machine will not return control for a long time. This is likely to trigger the watchdog.

There are two different solutions to this problem.

First, the interpreted code is aware of the watchdog as a virtual output and outputs to it frequently enough. The underlying native function then implements its part of the watchdog servicing strategy. This method requires, therefore, I/O virtualization outside the virtual machine.

Second, the virtual machine allows execution of the interpreted code in small pieces, i.e., it returns control to the caller which, in turn, can resume the execution from where the VM had left off. This method, therefore, requires a virtual machine capable of piecemeal execution of the code.

Code density and size of the virtual machine

These issues are straightforward yet important:

- the smaller the size of typical test modules, the faster the downloading and the less RAM is required
- the smaller the size of the virtual machine, the smaller the total overhead of diagnostic functionality

Support of programming techniques

Buzzwords of the past and the present – scope, visibility, modularity, structured programming, object orientation – do not necessarily apply to the development of stand-alone test modules.

The reason is that individual tests are typically small and have few interfaces to the outside world. Programs of this size are comfortably handled in a language of any reasonable style.

Tools for developing and debugging the code

What tools do you need to compile a human-made source code into bytecode understood by the interpreter?

What debugging support is available? Can the debugging be done on the target hardware?

C-SLang – a solution from MacroExpressions

C-SLang is a language optimized for small procedure-oriented tasks.

It requires no additional tools and meets the requirements of embedded diagnostics.

If you agree with the analysis above, you might make an observation that commonly used interpreted languages are not very well suited for writing embedded diagnostic modules. However, your solution may be readily available.

C-SLang (http://www.macroexpressions.com/c-slang.html) is a niche language designed specifically for the purpose of implementing small downloadable modules, such as diagnostic/test code.

On the surface, C-SLang is an assembler-like language for a simple virtual machine.

Under the hood, the language elements are (rather unusual) constructs in the standard ISO/ANSI C language. If your C compiler is standard-compliant, you do not need any additional tools to compile the source to bytecode. (If your compiler is *not* compliant, there are a few workarounds.)

The instruction set of the C-SLang virtual machine is optimized for small procedure-oriented tasks. As an example of resulting code density, filtering a SAE 1978 diagnostic message complete with building a negative response takes well under 150 bytes.

C-SLang virtual machine is written in standard C and is completely portable. Its size is about 2K for RISC machines (and smaller for other architectures).

The execution model of C-SLang VM assumes I/O interfaces via virtual ports. The VM must be supplied with memory buffers to operate; it doesn't have any memory of its own and is therefore fully reentrant. This automatically allows executing several C-SLang modules from different contexts.

C-SLang VM provides single-step execution and unlimited breakpoints. These features can be used both for debugging C-SLang modules (including inspection and modification of variables) and for piecemeal execution.

Conclusion

Providing ever-growing diagnostic and testing functionality in small embedded devices makes using interpreted downloadable code an attractive proposition. To get the most benefits from this approach, the interpreted language and the underlying virtual machine should be selected with great care.

MacroExpressions offers a solution with C-SLang, a simple language requiring no additional tools. For more on C-SLang, visit http://www.macroexpressions.com/c-slang.html.