Debugging embedded systems

By Gokhan Tanyeri and Trish Messiter, Clarinox Technologies Pty Ltd, www.clarinox.com

Embedded systems are designed to accomplish a very specific task or group of tasks. Although no single set of constraints will factor in all embedded systems, it is likely that the designer must balance constraints such as robustness, small size and weight, real-time requirements, long life cycle, low price, and low (or no) tolerance for malfunctions.

According to Robert Cravotta, Technical Editor – EDN, in his article "Shedding light on embedded debugging", 9/4/2008 "For each year of Embedded Systems Design's annual market survey of embedded-system developers, the single most requested area of improvement for design activities is debugging tools. The percentage of respondents making this request has remained steady at around 32% throughout the three years of the survey."

There are many reasons why debugging is seen as the most problematic and costly issue of the development cycle including increasing complexity, the balance of often conflicting constraints, "increased inaccessibility to silicon, lack of bug reproducibility and more pressure to meet shorter development schedule cycles." (http://www.virtutech.com/news_events/pr/pr2007_05_14-b.html)

The trends in the industry that these reasons will continue to intensify and so new approaches to debugging are required. In this paper we put forward some suggestions that have been found to assist. The underlying principle is simple - use all available tools, low level and high level, to isolate and identify the core issue. Our suggestions are:

1. Use both high and low level debugging tools
2. Have built-in unit testing in your code
3. Make sure that your code can also run on a desktop
4. Have integrated debugging architecture in your code
5. Use memory management tools
6. Use profilers and code coverage tools
7. Use and re-use proven and well-tested software components

1. Use both high and low level debugging tools

It is important to gain knowledge of all levels of the system and make the most of both high and low level tools. Use the IDE JTAG and Logic Analyzer for low level and hardware debugging however complex systems require more than use of simple set breakpoint and CPU register analysis. Some of the IDEs provide RTOS awareness to provide RTOS information. This does improve the visibility of the code to the user.

The low level tools are mostly useful during the hardware debugging and early stages of a project, e.g device driver development. Whereas most of the developer's time is spent integrating these device drivers to the framework used, I/O devices, wired/wireless protocols and the applications. Code size and complexity of an average project increase has a similar trend to the growth of computer hardware defined by Moore; "Moore's law describes a long-term trend in the history of computing hardware. Since the invention of the integrated circuit in 1958, the number of transistors that can be placed inexpensively on an integrated circuit has increased exponentially, doubling approximately every two years" http://en.wikipedia.org/wiki/Moore's_law.

Use a high level tool for debugging at protocol level to understand the characteristics of your system. Use an integrated debugger that would provide information about the RTOS events such as thread switching, memory use, inter-process communication and timers.

It helps to integrate protocol tracing for your I/O interfaces synchronized with the function level tracing and RTOS events.

2. Have built-in unit testing

Divide and conquer approach to debugging. Eliminate bugs in a step wise fashion so that overall bugs will be most likely due to overall co-ordination, inter-process communication and timing issues rather than from individual unit errors. Use unit test architecture to make sure that your units work against the design criteria, prepare automated test scripts for all your unit tests, so that before a major release is required, you can perform regression testing. Any bugs found during this initial phase of the development will save tremendous effort at a later phase.

3. Make sure that your code can also run on a desktop

There is general resistance within the industry to this idea, such as;

- It is a waste of effort, we can't run most of our code any way
- It is very hard to rewrite all our code for another platform
- Timings won't be the same, PC runs 100 times faster then our target
- It wont save much time anyway

However it our experience that this step is not a waste of effort.

Looking at these points, firstly let's examine the idea of most code will not run anyway, through an example:

Say the project is to integrate a GPRS module onto a target system. In such a case the purchase of an evaluation module which can be connected to PC via serial port will facilitate the majority of the development on the PC. The notable exception in this case is the UART device driver, but even if this item is left out, any issues here can be identified more quickly if other issues have already been resolved.

Use a middleware that enables running the code on PC and on target.

Next, the concept that it is very hard to rewrite all code for another platform. This is undoubtedly true after the fact however if you integrate the idea from the beginning of the project, through the use of a middleware that supports the concept then most code will not need to be rewritten. In addition, as an extra benefit, code is now more readily portable to other platforms in case, in the future the processor became obsolete, or a lower cost alternative became available.

Now to the issue that timings won't be the same, since a PC runs many times faster than the target. This is again undoubtedly true and, if the code depends on sleep() or similar timing calls to synchronise with external events or other threads, then it will be an issue. If however the design uses interrupt, semaphore or conditional variable concepts then this issue can be largely avoided. It also avoids the debug of costly synchronisation issues and race conditions.

Finally, to examine the idea that it won't save much time. Just consider the time required to connect a JTAG to your target and downloading a 500KB application every time a test is run. Compile, link, connect, download and run could be easily in the order of 1-3 minutes, giving you a required time of at least a half an hour for 30 reruns just waiting for the operation itself. On a PC the test will run in 10 seconds. For 30 runs this is 300 sec, or 5 mins. A time saving of over 80%. Multiple this out over an intensive test period of two week; and it is possible to save about a week and a half. Imagine a larger project that employs a team of ten people. The additional benefit is then that focus can remain on issue to be resolved rather than the distraction of the time consuming process.

So consider taking advantage of the increased speed and visibility available in a desktop computing environment by doing as much debugging within this environment as possible. Use the IDE, virtualization, simulation and middleware tools.

4. Have integrated debugging architecture in your code

This is especially useful when running the desktop version of the code. Use integrated function tracing with threading information to enable viewing of the execution of functions versus RTOS events. It is also possible to use a remote debugger to extract information out of an embedded target device.With a remote debugger, you can analyze and display debug information and then it is possible to visualize the flow of the code, regardless of how complex that code is.

Use macro definitions to output debug information, later on harness this macro to output the information to a serial UART port, Ethernet, console or even on an LCD display. For the production code you can turn the debugging off by using the same macro definition to reduce the overhead and the code size, once you are confident with the product.

Take care to note that by using an external debugger, you will be running an agent component of the debugger on your target device to send information to the debugger and receive commands from the debugger. This may create a large amount of information to be sent to the remote debugger changing the timing characteristics of the system hence forcing an overload of the system and creating the situation where you must make sure the code is tolerant to timing changes.

5. Use memory management tools

Seek out memory leeks early to refine your code to be extremely robust. Failure to do so will result in hard to reproduce and apparently inconsistent behaviours that are extremely difficult to diagnose. Stay away from compiler's dynamic memory allocation / deallocations as much as possible. Memory fragmentations will cause a slow but eventually fatal ending. Instead use

dynamic allocation / deallocation of fixed size buffer pools to eliminate memory fragmentation. Make sure to analyze use of memory pools and define optimal pool sizes.

6. Use profilers and code coverage tools

Refine and tune the code and find out performance bottlenecks by using profilers. Identification, and remedy, of performance issues, results in lower likelihood of random system crashes due to the overloading of the processor. Eliminate the unused code as most times such code has a root cause in design errors which can be uncovered by this chasing up of such dead code.

7. Use proven and well-tested software components

Use code that is proven from many projects. Avod re-inventing the wheel for every component to ensure:

- Keeping focus on the product application
- Reduce the introduction of bugs
- Minimise the requirement of resources for maintenance

Engineers by nature often prefer to start with a clear slate rather than reuse or adapt existing designs. A balance however is required as companies in general require the development of products rather than the development of increased fundamental knowledge of their engineers. Companies too must produce new products in decreasing time frames. It is in the interest of the companies for engineers to make best use of whatever tools and reusable code is at their disposal to gain extra efficiencies and hence faster time to market. This is a fundamental conflict between the engineering drive for what is best and the economic drive to be first. A sensible balance is required.

An example how things change is to consider that a few decades ago it was common for RTOS to be developed in-house for complex projects. These days the RTOS is far more likely to be considered an off-the-shelf item

There are other efficiencies of similar nature however that are not yet so common place. Reuse of in-house code is reportedly quite low. Some increased level of reuse where the code is of sufficient standard and applicability makes commercial as well as technical sense.

Wireless protocols are becoming increasingly required in designs and it can be quite tempting for an engineer to spend time learning the new wireless technology. However it may be more cost effective for the company to acquire a tested wireless protocol and have the engineer develop intellectual property at the level that sets them apart from the competition – the product application level. Once the design phase is over, the company needs resources for the next generation of the product not for maintaining the code for the wireless technology. It makes economic sense to avoid being involved to follow new specifications and instead source this technology when it is available. Let the vendor spend the tens of engineer-year effort to develop and follow the technology for you.

Conclusion

While the ultimate goal may be to develop code that has zero issues, embedded software engineers must live in the real and imperfect world where system complexity is such that invariably issues will occur. The fact that debugging remains the most problematic and costly issue of the development cycle as reported by industry survey indicates that engineers must digest the idea of ever increasing complexity, prepare for the challenge from day one and acquire more and better debugging tools to keep pace with the increasing demands on developments.

About Clarinox

Clarinox provides embedded systems tools, middleware, wireless and wired protocols, device drivers and engineering services particularly for complex wireless embedded developments.

Clarinox have used the ClarinoxSoftFrame embedded wireless middleware platform over many years and across a large variety of platforms and applications. The value of such tried and tested software is;

- to reduce risks within commonly required code and hence reduce debug times
- to be able to use common architectural blocks (finite state machines, inter-process communication, timers, integrated debugging)
- Ready to use wireless and wired protocols and I/O device interfaces
- Portability over almost any RTOS and processor
- Speedy time to market

Contact: enquiries@clarinox.com