Prototype debugging

… an investigation of hardware / software issues

The process of rapid prototyping is often performed by the integration of several main system components from various vendors. Components include software and hardware items such as off-the-shelf 32 bit processor board, Real Time Operating System, Board Support Package, and protocol stack/s. It is not uncommon that at the end of a lengthy period of integration and application development, that the testing phase reveals that the overall function has not been achieved. Tracing the source is difficult as each individual component vendor can claim correct function out-of-box. Six weeks later, the basic overall software and hardware functions still can not be achieved. Software vendors claim that it is not a software fault, hardware vendors claim it is not a hardware fault and none of the standard tests or tools have proved conclusively one way or another. This is a classic debugging scenario nightmare.

Wikipedia defines debugging with the phrases "Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another."

It appears that computing has taken the word from an older usage as whilst "The terms "bug" and "debugging" are both popularly attributed to Admiral Grace Hopper in the 1940s[1]. While she was working on a Mark II Computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. However the term "bug" in the meaning of technical error dates back at least to 1878 ..., and "debugging" seems to have been used as a term in aeronautics before entering the world of computers."

Wikepedia goes on to say that "Debugging is, in general, a lengthy and tiresome task. The debugging skill of the programmer is probably the biggest factor in the ability to debug a problem, but the difficulty of software debugging varies greatly with the programming language used and the available tools, such as debuggers."

This view has been support as early as 1997 when researchers Vranken, H.P.E. Stevens, M.P.J. Segers, M.T.M. Dept. of Electronics Eng., Eindhoven University of Technology stated that "...the debugging of hardware/software systems is still a very troublesome process. This is mainly due to the limited accessibility to the internals of embedded hardware/software systems."

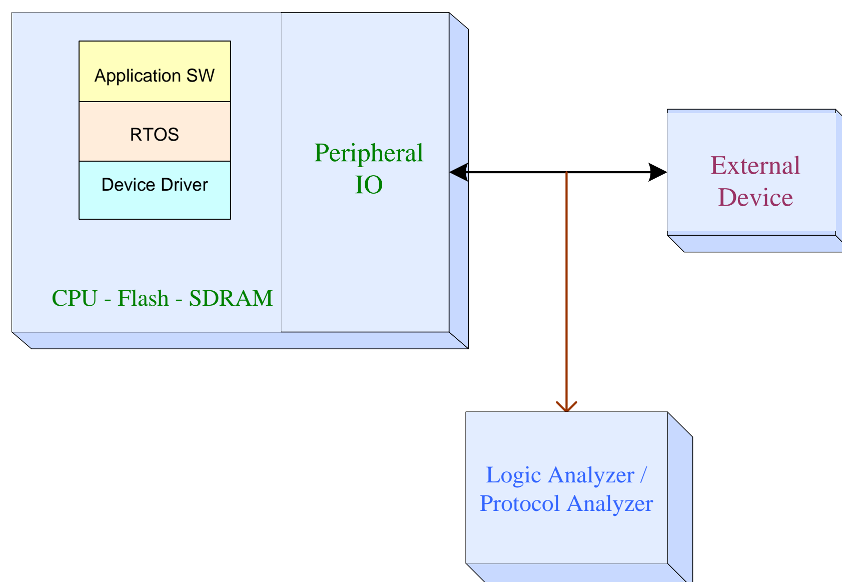
If this is a long held and recognised view of debugging does this relegate the task to those few creative individuals who love the challenge – or is there hope for more ordinary mortals?

Let us return to our debugging nightmare. You are faced with a system that clearly does not work as desired. Many engineers have a natural belief in their own work and natural disbelief in the work of others, however a good engineer believes only in facts and isolates the issue by using facts.

Part of the function required is to interface to an external hardware device. The application software is using the device driver provided by the RTOS vendor. The issue seen is that, on rare occasions, the correct data is not received by the application. There are several elements to this,

1. The transmit side of the application
2. The RTOS provided device driver on transmit side
3. The external hardware
4. The device driver on receive side
5. The receive side of the application

To determine which element has the issue, rewrite code to isolate to a particular code segment. Just include the code concerned with send and receive data and eliminate other sections irrelevant to the issue. This assumes of course that the issue is not a timing issue or a corruption caused within the seemingly irrelevant code portions. Below picture depicts this hypothetical scenario.



To make sure that the application is sending and receiving the correct data, print all data sent to device driver and print all data received from the device driver. Printing depends on the available debugging ports on the hardware. It could be a spare serial port, on device Flash file system or as simple as a hardware port or LED that could be monitored by a Logic Analyzer or a storage oscilloscope. If the printing proves that the data is as per design, then the application transmit side is proven to work correctly.

If the data sent by the application prints correctly but the received data is not correct then it appears that we need to look deeper for a device driver or hardware issue. Assume this to be the finding in this example. After this it is time to use a protocol or logic analyzer to analyze bytes coming out of the processor board. If the data monitored at the hardware port of the board matches the data printed by the application, then we are satisfied with the items 1 and 2 of the suspect list. However, if the application sends the correct data, proven by printing, but the device driver does not generate the same data coming from application on the hardware port then the error can be assumed to be in the device driver/ Board Support Package.

If the hardware sends the data presented to it from the device driver and the external device generates data back to the board then the error can be assumed to be in the external hardware device. But, wait, life may not be that simple, we may see the expected sequence of data received at the input port of the hardware. This clears the doubts about the external device being faulty. We then

look at the possibility of the device driver receive side/Board Support Package receive issues versus the most likely problem of us using the device driver incorrectly. How do we decide whether we do the right thing and the device driver is occasionally missing bytes or our receiving thread does not have high enough priority and while processing the previous packet we are causing an overrun of the hardware receive buffer. Now, this is a good time to add error checking after each API call and reading the hardware status register to be able to detect the unexpected. Most of the times we will find the issue in our interpretation of the device driver APIs and the mechanism. But, do not spend weeks on this issue, email the vendor or online communities to find someone else who has been down the same path.

Finally, the most likely culprit has been found and the decision is in fact that there is an error within the RTOS supplied device driver. This needs to be forwarded to the real-time operating system vendor to obtain a fix (assuming not open source). Presenting all the facts gained during testing is the key factor to get this required technical support in a timely manner. Calming down to restate every step taken and every result observed is time consuming and will take some effort. Writing even what seem obvious or trivial steps such as "power on the unit, wait until the green LED is on, etc" will enable the correct flow to be tracked through the third party code. In our example the RTOS vendor needs to know which elements of the device driver code have failed (given the problem is intermittent it must be a portion of code only entered upon the meeting of specific, infrequently met, criteria) to be able to rectify the issue.

In our experience adoption of this methodical and scientific approach will pay back. Of course it is also important to design to avoid bugs in the first place, but that topic we will tackle at another time.

About Clarinox Technologies

Clarinox Technologies specializes in embedded systems and short range wireless technologies, products include ClarinoxSoftFrame, ClarinoxBlue, ClarinoxScan, ClarinoxGPRS. Together these products provide the building blocks for constructing an embedded short range wireless product.