Real-Time Embedded Debugging

Rafael Román Otero
University of Northern British Columbia

April 16, 2012

1 Introduction

Embedded computing had become an imprescindible part of today's society, the number of applications exceed by far those of desktop and server computing [1]. In fact, more than 90 percent of deployed processors in the world are part of embedded sytems [6]. Their applications range from small MP3 player devices, to satellites and millitary weapons.

As applications in embedded systems increase in complexity, the use of a Real-Time Operating System (RTOS) becomes innevitable [6]. Debugging embedded software is by itself a difficult task due to the scarceness of available resources, the difficulty in collecting state information, and the fact that any possible perturbation in the systems timing constraints might produce a mutated output [5, 3]. Additionally, real-time applications add the complexity of non-determinism raised by race-conditions [4], and present the challenge of debugging while keeping disturbances to the minimum to avoid changing timing behaviour in a way that could lead to a violation of deadlines. Hence the classic methods of using breakpoints and interactive programming are usually unsuited for these systems. [2, 3]. Consider, for instance, inserting a breakpoint in the middle of a task that must react upon certaing sensor reading. If the physical variable that would have activated the output occurs for a short moment, while the processor is halt, then the system will miss that input and the system is no longer being executed under real circunstances.

This project intends to compile and implement already known methods for debugging of real-time systems. More specifically, tracing and monitor debugging.

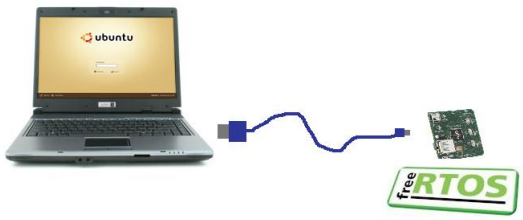

Figure 1: Host and embedded paltform

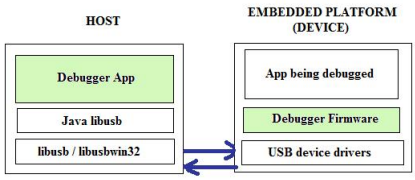

Figure 2: Debugger Architecture

2 Architecture

As it can be seen in Figure 1, the system is composed of the embedded platform being debugged, and a host computer. The host computer runs a Java Application that communicates via USB with the debugger firmware that is running as a task in the embedded platform, as depicted in Figure 2.

The debugger task gathers data from all the remaining tasks in the system through a buffer. So as to minimize any possible interference of the debugger task with the behaviour of the system, we set the debugger task with the lowest priority, so that it can be pre empted at any time. See Figure 3

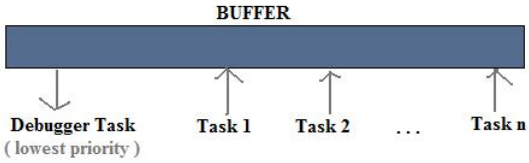

Figure 3: Debugger as a task

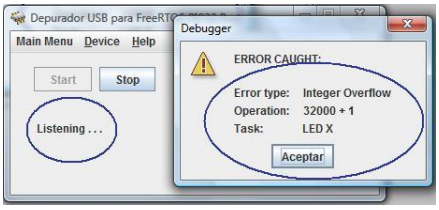

Figure 4: Monitor catches an integer overflow

3 Implementation

The tracer simply sends user-defined messages to the host with the intention of helping follow the execution flow of the system. It also throws some light on the questions: what's happening inside the my system? Are my messages getting where they whould be getting? Is the task executing at boundary values? etc.

The debugger monitor (running in the embedded platform) is responsible for detecting errors: Arithmetic overflows and underflows, stack overflows (not implemented yet), and indexes out of boundaries for arrays (not implemented yet). It also implements and allows the use of assertions, which are not supported neither by the language nor by the RTOS.

Everytime one of these error occurs, the monitor reports to the debugger software on the host, so that the user can be notified. The debugger monitor and tracer are always present in the system. Should the system want to be deployed without the debugger, then the debugger should be removed at compile time.

4 Results

In the first example, the debugger caughts an integer overflow. The limits for any overflow can be set manually to any value. In this case, 32,000 is setted as the maximum possible integer value. Thus we get an integer overflow when we try to increment an integer variable from 32,000 to 32,001. See Figure 4

We can also set assertions that will be catched when they are not fullfilled. Here we define the assertion that a variable must be greater than 100. Again when this is not true, the monitor catches it and reports to the user. See Figure 5.

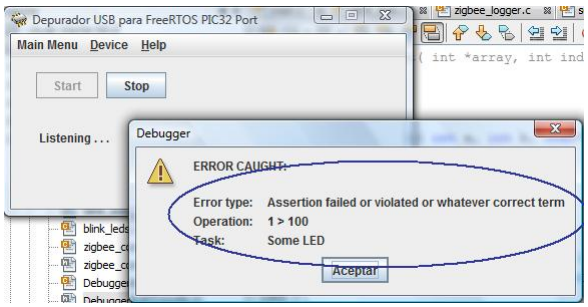

Figure 5: Monitor catches a false assertion

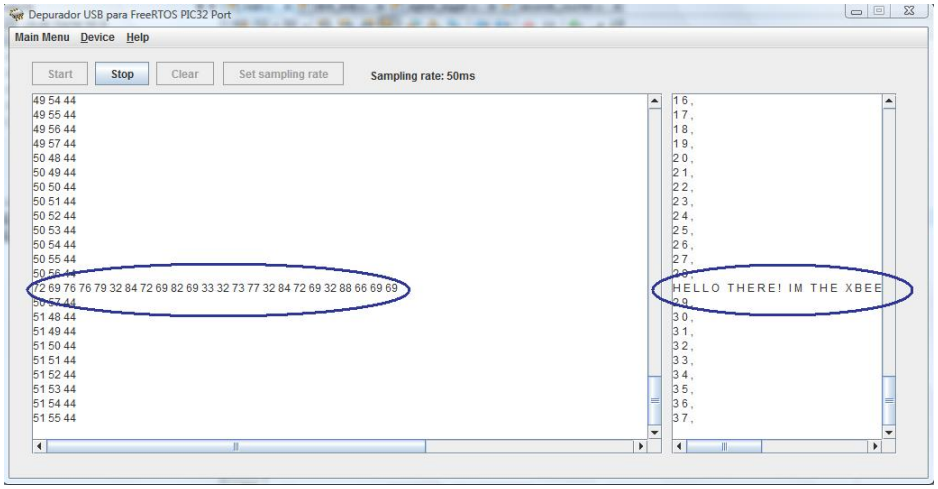

Figure 6: Tracing data from two tasks. A second-counter, and a Xbee Wireless Communication task

In thsi last example, there are two tasks tracing data. On tasks fires every seconds, and sends a message to the debugger everytime this happens. The second task is waiting for incoming data from an Xbee wireless communication module (it uses Zigbee protocol) that is attached to the serial port of the embedded device. So everytime a message is received from a remote computer, it re sends the message to the debugger. See Figure 6.

5 Conclusions

The project suceefully implemented some simple already-known methods for real-time embedded systems, and clearly improved the observability of the system being debugged, which according to [1] is one of the main requerirements to make software testable and debuggable.

As a matter of fact, the embedded platform comes with an in-circuit hardware debugger, which, as explained before, was useless due to the use of an RTOS. At least next time that I have to implement any application on the platform, I'll have a debugger.

References

[1] M. M. M. B. Aleksander Milenkovic, Vladimir Uzelac. Caches and predictors for real-time, unobtrusive and cost-effective program tracing in embedded systems. *IEEE TRansactions on Computers*, 60:992–994, 2011.

[2] B. Bonakdarpour and S. Fischmeister. Runtime monitoring of time-sensitive systems (tutorial suplement).

[3] S. Debugging and testing using the Abstract Diagnosis Theory. A debugger rtos for embedded systems. *LCTES'11*, 2011.

[4] J. W. E. P. Padma Lyenghar, Clemens Westerkamp. A model based approach for debugging embedded systems in real-time. *EMSOFT'10*, pages 69–76, 2010.

[5] V. J. m. Tankut Akgul, Pramote Kuacharoen and V. K. Madisetti. A debugger rtos for embedded systems.

[6] J. B. Wolfgang Haberl, Markus Herrmannsdoerfer and U. Baumgarten. Model-level debugging of embedded real-time systems. *2010 10th IEEE International COnference on Computer and Information Technology (CIT 2010)*, pages 1887–1894, 2010.