University of Eindhoven

A WI-FI JTAG DEBUG ADAPTER FOR EMBEDDED SYSTEMS

August 27, 2015

Internship preliminary report

Sander Vocke
0923556
`s.vocke@student.tue.nl`

Contents

1 Introduction

Many aspects play a role in successful and effective development of (embedded) software. These include design methodology, software architecture, understanding of the platform and specification, programming skill and generally adopted “good practices” such as unit testing and code review by peers. Also in the list is the practice of debugging, which in the widest sense refers to finding the root cause(s) which cause the application to not behave as intended. Depending on the project, debugging can take up the majority of development time **reference**.

This preliminary internship report explores debugging from an academic perspective - both in general and more specific research advancements related to the internship project (a wireless JTAG adapter for embedded code debugging).

2 Part 1: Literature Research

2.1 Common Debugging Technologies

2.1.1 System Observation

The most basic debugging technique, apart from studying application code, is to observe the system in operation in order to come up with a hypothesis about the fault. The developer may make changes to application code (blinking LEDs, *printf()* statements, etc) and re-run the application in order to gain more information about the system state and execution flow.

While many developers may believe that this technique is superseded by symbolic debugging in most cases, this technique is still widely used **ref?**.

2.1.2 Symbolic Debuggers

Symbolic debuggers are widely used, crucial tools in software debugging. They have access to information about symbols used in the application code, such as variable and function names, and their relationship to the application binary. Also, in general they are able to control execution flow of running applications and inspect or alter the state of the system, for example by accessing memory and registers. This combination of features makes them into versatile tools to track down the root causes of bugs and fix them in the application code.

However, multiple problems are associated with symbolic debugging, some specific to embedded systems. One of them is the "probe effect": in general, execution of the binary while debugging symbolically is not behaviourally identical to executing the binary normally. This may lead to difficulties in reproducing the fault or give inaccurate information about its cause(s). Arguably this is especially problematic in embedded systems, where correct execution tends to rely more on correct timing of hardware inputs and outputs. Another problem is the resource overhead imposed by symbolic debugging. The resources required to run a symbolic debugger are not always available in embedded systems, prompting the use of techniques to debug an application remotely from a host system **refer to section**.

in some shape or form, have existed ever since debugging information could be graphically displayed (en.wikipedia.org/wiki/Debug_symbol). Modern debuggers typically support advanced features such as conditional breakpoints, ...

Regardless of the drawbacks of symbolic debugging, it is **probably** still the most widely-used debugging technique for software engineers, embedded or not.

2.2 Emerging Debugging Technologies

2.2.1 Back-in-time Debugging

2.2.2 Automatic Fault Localization

2.3 Debugging of Embedded Systems

2.3.1 Remote Symbolic Debugging

As described in section 2.1.2, symbolic debugging on embedded systems amplifies the problems of "probe effect" and debugging overhead. It has become commonplace to debug embedded systems remotely from a host system, in order to make the necessary resources available and to debug with minimal overhead on the embedded device itself.

In order to perform symbolic debugging remotely, the debugger, running on the host system, must have similar access to the embedded system's state as it would when debugging an application on the host system. Therefore, some kind of interface is required between host and target. An example of such an interface is the **official name** JTAG standard **ref to section**.

In many cases, a debug adapter is needed to bridge the debugging interface to whatever interface the host supports, such as USB or a serial port.

The embedded target device must also provide a certain level of access to its internals via the interface, in order for the symbolic debugger to be able to inspect its state and control execution flow. Typical modern embedded processors have various levels of provisions for this, which may include halt/continue/step/reset control and read/write access to registers and/or memory through the debugging interface.

All in all, this makes remote symbolic debugging a similar experience to regular symbolic debugging, although the addition of a communications interface in-between tends to result in extra performance overhead.

2.3.2 Device-specific Debugging Features

In order to alleviate some of the drawbacks of remote symbolic debugging, embedded devices and/or processors may incorporate extra features which are useful for debugging. An example is the ARM Cortex-M family of embedded processors, which (optionally) include the following features:

- Access to system exception state
- Hardware break-/watchpoints
- Dedicated hardware for patching/remapping parts of code memory
- Run-time non-intrusive profiling
- Instruction tracing

More information about these features will be given in the final internship report.

2.4 The Practice of Debugging

The practice of debugging in general has been studied thoroughly over the last few decades. Several relatively recent studies **references** give overviews of how debugging has evolved, highlighting the advancement of debugging tools and technology, but also investigating the experiences of software engineers in order to assess the amount of improvement these technologies have actually achieved.

Among other things, [**?**] investigates whether recent debugging technologies such as back-in-time debugging and automatic fault localization have made any difference in the daily debugging practices of engineers. While only several developers were interviewed for this study, an interesting result was that none of them had heard of either of these technologies before.