DEBUGGING THE STH ON RPI2

DIDRIK LUNDBERG
DIDRIKL@KTH.SE

This is a comprehensive guide that will cover all steps of debugging the STH on the RPi2 platform. Ubuntu 15.04 was used in the writing of this guide, but things should be similar on similar systems. This guide has a small amount of overlap with the "Bare-metal programming on the Raspberry Pi 2" guide which deals with more basic points, so if you find yourself lost trying to follow the steps in this guide, go back to that guide to learns the basics first. Similarly, the guide "How to add new platforms to the STH" deals with building the hypervisor. If you have any questions, please send an E-mail to the author.

1. Introduction

Firstly, you will need a few tools for doing this. There are several ways to debug using JTAG with `gdb`. We will be using `gdb` in conjunction with `emacs`, and the FT2232H Mini-Module JTAG adapter.

If you have not yet done so, install:

- OpenOCD (`git clone git://git.code.sf.net/p/openocd/code`)
- Minicom (`sudo apt-get install minicom`)
- ARM cross-compiler (`sudo apt-get install arm-none-eabi`)
- ARM cross-debugger (`sudo apt-get install gdb-arm-none-eabi`)
- Emacs (`sudo apt-get install emacs`)
- Telnet (`sudo apt-get install telnet`)

After cloning OpenOCD, navigate to the directory where OpenOCD was cloned, and type `./configure -enable-ftdi`, which should do the trick for the configuration stage. Then type `make` (make sure there are no errors!) and then `sudo make install`.

2. Setting up debugging

There are two things we would like to note regarding the JTAG pins on the Raspberry Pi 2. One, there is no SRST pin. Two, no pins have JTAG output by default, meaning they must be configured to serve that purpose. These two facts discourage us from using ordinary reset techniques. I have also had problems with removing breakpoints. There are workarounds for this, though.

First, you need initialize JTAG somewhere. Look at the file `debug_gpio.c` and specifically the function `debug_gpio_init`. This function can be called in `boot.S`, just after you set the stack pointer, by adding the line

```
bl debug_gpio_init
```

Date: August 2, 2015.

At some point after this, add the assembly snippet

```
ldr r0, =0x47 //Set r0 to 0x47.
debug_loop:
cmp r0, #0x11 //Compare r0 to 0x11.
blne debug_loop //Loop if r0 is not equal to 0x11.
```

This will stick the hypervisor inside a perpetual loop. The point here is that when you reach this loop, you can be sure that JTAG is initialized and that you are at a particular place in the code. In practice, it will take a negligible amount of time for the hypervisor to reach this code if you put it inside `boot.S`.

Always keep a Minicom window connected to the RPi2 over serial port. With Minicom installed, simply execute `sudo minicom -b 115200 -D /dev/ttyUSB1` to initialize this connection. Additionally, press Ctrl+a, then w and Ctrl+a, then u to enable line wrap and addition of carriage returns when needed. Remember to shut down Minicom in an orderly fashion - if you do not do this, you will start getting garbled signals over the UART. Shut down Minicom by pressing Ctrl+a, then x; but if you find yourself in a situation where you are getting garbled signals, you can just restart your PC.

As soon as you plug in power to the Raspberry Pi 2, you should start seeing log info appear in the Minicom window. After you see the message

```
Starting kernel ...
```

You know that the kernel is running. Then, in a new terminal window on your PC, execute the command

```
openocd -f interface/ftdi/minimodule.cfg -f target/raspberrypi2.cfg
```

This will set up an OpenOCD server which is connected to the Raspberry Pi 2 over JTAG. Your OpenOCD should now detect the processor of your debug target. It will list the number of processors, their respective breakpoints and so on, and then wait (not giving you the prompt back). To stop the execution of the server, press Ctrl+C - but do not do so until you are completely finished debugging, and disconnect the programs which are connected to the server first.

Now, we want to halt the execution of the hypervisor inside the aforementioned perpetual loop, so that we can get out of it and start debugging. First, connect with Telnet to the OpenOCD server by executing the command `telnet localhost 4444`. Then, at the Telnet prompt execute `targets` to look at which processors there are. Then, again at the Telnet prompt, execute `halt` (because you want the processors to halt), and then `targets bcmrpi2.cpu0` to set the first processor core as the debug target. Now you are ready to start debugging with gdb.

Start Emacs, and once inside press Macro+x (the Macro button is typically the left Alt) and type `gdb`. The prompt will now say

```
Run gdb (like this):
```

and you should execute the command `arm-none-eabi-gdb -i=mi`. You are now running the GDB cross-debugger in one of the Emacs buffers, but this is not enough. Again, press Macro+x and then execute the command `gdb-many-windows` at the resulting prompt. This will give you more useful buffers, so you can monitor registers, variables and breakpoints continuously.

It can be useful to load the symbol file. This will allow you to track variables and follow your debugging in the C code of the hypervisor. Do so by executing the command

```
symbol-file path/to/hypervisor/core/build/sth\_platform.elf
```

at the `(gdb)` prompt, where you replace `path/to/hypervisor` with the path to the hypervisor directory relative to the home directory, and `platform` with the name of your platform. Also, be sure to open the disassembled file (at `core/build/sth_platform.asm`, where you replace `platform` with the name of your platform) to keep track of what happens at or around the memory address you are currently debugging.

Finally, connect to the OpenOCD server by executing the command `target extended-remote localhost:3333` at the `(gdb)` prompt. You should now be able to confirm that you are inside `debug_loop`.