BARE-METAL PROGRAMMING ON THE RASPBERRY PI 2

DIDRIK LUNDBERG
DIDRIKL@KTH.SE

This is a comprehensive guide that will cover all steps of bare-metal programming necessary to run and debug the SICS Thin Hypervisor [2] for the Raspberry Pi 2. Ubuntu 15.04 was used in the writing of this guide, but expect similar behaviour on similar systems. Please open this .pdf file in a viewer which can display hyperlinks, as we provide links to various documentation files. If you have any questions, please send an E-mail to the author.

Furthermore, I would like to thank Brian Sidebotham [7] and David Welch [8] for their excellent guides for bare-metal programming on the Raspberry Pi 1 and 2.

1. Introduction

In order to follow this guide, you will need some equipment:

- A Raspberry Pi 2 Model B single-board computer
- A PC to program on
- Internet connection via an Ethernet cable
- An SD card reader
- A serial-to-USB converter cable
- A USB to serial interface converter module (for JTAG)
  - ... including the USB cable to connect it to your PC
- For testing the hardware:
  - A USB mouse and a USB keyboard
  - A monitor with a HDMI cable connection

Most items should be pretty straightforward to acquire. RPi2 uses micro-SD cards, so you will require your SD card reader to be able to read those. The serial port of the RPi2 is more specifically a UART using the 3.3V TTL standard of voltages - please note that converter cables which use other standards, such as RS-232, may fry your Pi2! However, some cables work for both RS-232 and 3.3V TTL. Look up the specifications of the chip in the converter cable you are buying just to be sure.

The UART communicates via two GPIO pins on the ordinary RPi2 pin block. However, the serial-to-USB converter cable should typically have four jumper wire connectors on one end and a USB connector on the other (the two additional wires providing ground and 5V power). If you find this to be confusing to sort it out yourself, I bought a serial-to-USB converter cable which uses the PL2303HX chip to transform the signal which I recommend -

Date: July 16, 2015.

that chip name might be a good word to search the Internet for if you want to buy one yourself.

2. Testing the equipment

First, we must make sure that all our equipment is functional. To do this, we first download NOOBS from the homepage of the Raspberry Pi Foundation [6]. NOOBS is an operating systems installer, which we will use to comfortably set up Raspbian for testing purposes.

Simply format your SD card in the FAT32 file system (using for example `gparted`, which can be obtained by `sudo apt-get install gparted` if you do not have it already) and transfer the decompressed contents of the NOOBS folder to it, and you are ready to boot your RPi2 for the first time. Put your SD card in the RPi2, connect the Ethernet cable, the USB keyboard and mouse, the monitor, and then finally plug in the micro-USB cable (connected to your PC or to a USB power adapter of good quality on the other end) to supply power and your RPi2 will boot into NOOBS.

Install Raspbian, and inside Raspbian install the `screen` utility by typing `sudo apt-get install screen` in the terminal. Also install `screen` on the PC you are programming on in the same way.

Now it is time to connect the RPi2 to your computer with serial-to-USB converter cable. You must do this with great care, since connecting the wrong jumper wire to certain pins might fry your equipment. We will explain how to do this for our generic PL2303HX cable. It has four jumper wires:

- **TxD**: Green. The "T" in the acronym stands for "transmission" - should be connected to the UART receiver pin (RxD) on the RPi2 block.
- **RxD**: White. The "R" in the acronym stands for "receiving" - should be connected to the UART transmission pin (TxD) on the RPi2 block.
- **Ground**: Black. Should be connected to the Ground pin on the RPi2 pin block.
- **5V power**: Red. WARNING! Do not plug this cable in if you are also supplying power via a micro-USB cable - that might fry the RPi2. This cable is an alternate way of providing power to the RPi2 without using a micro-USB cable.

It is important to note that the pin layout of the RPi2 is closely resembles that of other versions, the difference being that it has more pins (40 compared to 26 for the RPi1). The 26 first pins counting from the pin closest to the SD card have the same functions. There are two ways of numbering the GPIO pins - using their GPIO numbers by which they are referred to by the computer, or using how they are physically ordered on the board, a terminological detail which might be a source of some confusion.

So, by the above physical numbering seen in Figure 2, the ground wire should be connected to pin 6, the receieve wire should be connected to pin 8, and the transmit wire to pin 10, as shown in Figure 1. If you know what you are doing, you can also power the RPi2 by connecting the power wire to pin 2 or 4, but then you risk frying your equipment if you supply power over the micro-USB at the same time, so I would not recommend this.

Now, you first want to check that the serial-to-USB cable works correctly. To do this, write `lsusb` in a terminal window, and you should see a list of connected USB devices, in which the serial-to-USB cable should be identified. Second, we need to know the device name of the cable. Simply use `dmesg | grep usb` and you should be able to filter out the device name of the cable among the driver messages (in my case `/dev/ttyUSB0`).

Setting up the connection on the RPi2 should be easy - simply write `sudo screen /dev/ttyAMA0 115200`. The number at the end is the symbol rate of the connection, measured in bauds (bits per second). Similarly, on your computer, write `sudo screen /dev/ttyUSB0 115200`, where you replace the device name if needed. Now, if everything works correctly you should be able to write on the PC and see the characters appearing on the monitor connected to the RPi2, and vice versa. This ensures us that all the hardware works correctly, saving us valuable time looking for errors later if that should not be the case.

3. Writing a test program

Now, you will need an empty SD card. Either re-format the one you have, again to FAT-32, or use another one, which might be more handy if you need Raspbian later on.

Writing a bare-metal program requires intimate knowledge about the hardware. The hardware of the RPi2 is very similar to that of the RPi, whose chip BCM2835 has excellent documentation [1]. The only significant

Figure 1. The serial-to-USB converter as it looks connected to the RPi2

difference is the processor, and the increased RAM of the RPi2, the consequences of which we need to handle. Indeed, there are more differences, such as for example a mini-UART on the RPi1 which is not on the RPi2 board, but these differences are not relevant for this guide.

It follows that we can use the BCM2835 documentation for working with the RPi2, with a few important address corrections: the physical base address of the peripherals (which we will use when programming) has changed from `0x20000000` to `0x3F000000`, and the size of the memory has increased to 1 GiB (although some is reserved for the GPU by the bootloader which is run before the kernel, but that is irrelevant at this point).

Accordingly, all addresses you read in guides for programming on the RPi1 will have to be adjusted with this in mind. Typically, this just means that you replace `20` at the start of a physical address with `3F`.

Technically speaking, we are going to write a kernel - although our "kernel" will not have much of the functionality we associate with one, it will take the place of a kernel during the boot process. It follows that we require a bootloader and other accessories to get running. The easiest way to solve this is to obtain the GitHub repository `raspberrypi/firmware` through `git clone https://github.com/raspberrypi/firmware.git` and copy the contents of the `boot` folder to your SD card, deleting the files

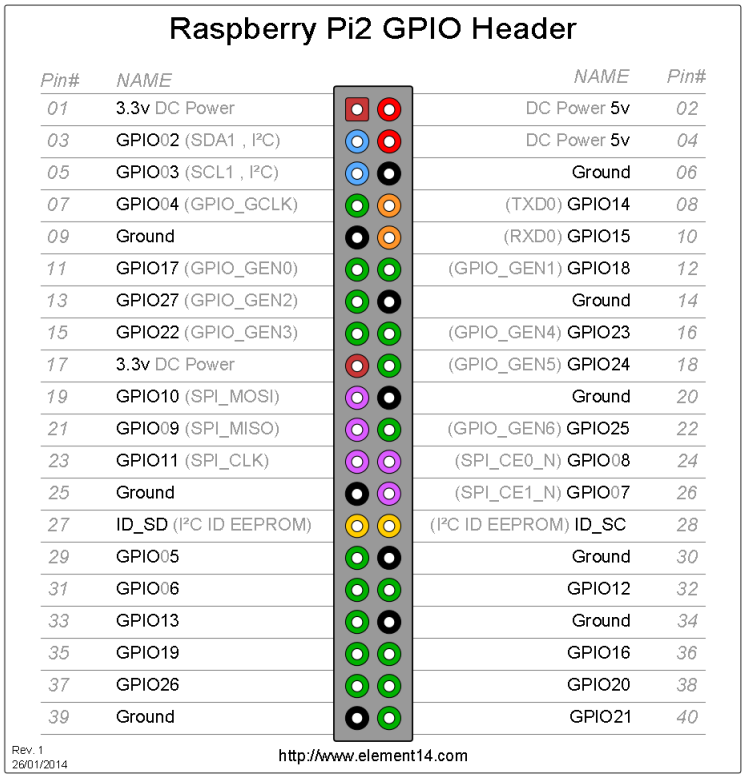

FIGURE 2. The RPi2 pin numbering [5]

`kernel.img` and `kernel7.img` (which are the existing kernels). We end up with some redundant files, but are guaranteed things will work.

3.1. **Setting up a cross-compiler.** You will require a compiler which can compile code to run on other hardware. This is easily solved by the GNU ARM Embedded Toolchain, which is installed by the command `sudo apt-get install gcc-arm-none-eabi` and you are good to go. Then, to compile (for example) `test.c`, you would use (on one line):

```
arm-none-eabi-gcc -O2 -mfpu=vfp -mfloat-abi=hard -mcpu=cortex-a7
-nostartfiles test.c -o kernel.elf
```

However, we do not want an ELF file, but a binary which only includes machine code, so we extract that part from `kernel.elf` using the `objcopy` utility: `arm-none-eabi-objcopy kernel.elf -O binary kernel.img`, and get the binary we want in `kernel.img`. This can then be put on the SD card together with the bootloader, as described above, which will look for the kernel.img[1] and execute it.

3.2. **Setting up CMake.** However, in reality we never just compile a single C file. We are going to have multiple files in C and ARM assembly language and so we will need CMake, in order to keep things neat and tidy. To install CMake, execute the command `sudo apt-get install cmake`. Now, the CMake-related files you start out with are `CMakeLists.txt`, `configure.sh`, and `toolchain-arm-none-eabi.cmake`. These are fairly self-explanatory, but you should know that you will need to change them if you change the structure of your project (add more files, change names of files, and so on). Otherwise, all you need to know is that you set up CMake by executing `./configure.sh`, and then build simply with `make`.

3.3. **Example code.** Several useful example programs (or "test kernels") can be found in the `rpi2-port` sub-directory if you

```
git clone https://github.com/guancio/kth-on-rpi2
```

These consists of code in C, ARM assembly language, and a linker file called `linker.ld`

The linker script decides where to physically put different parts of the code in the binary file, for example the magic number we want in order to pose as a Linux kernel when using U-Boot. If you change the linker script or the assembly code in `boot.S`, make sure that the magic number is in the correct position if you are using U-Boot as described below (this can easily be ensured by adding new sections for your own code in `boot.S` and inserting them after .magic in the linker script).

A more thorough description of the C and assembly code of these test kernels can be found in the comments inside the files, please also keep the BCM2835 documentation as a reading companion.

[1]Actually, the RPi2 bootloader will first look for kernel7.img, and then, upon not finding kernel7.img, for kernel.img. Do not have unwanted kernel7.img files lying around on your SD card or the bootloader will use these instead!

3.3.1. *test-kernel-1.* This test kernel will make the green LED on the RPi2 blink, with approximately one-second intervals between each flash. This simple program is good for testing if your pipeline to get code running on the RPi2 is working correctly.

3.3.2. *test-kernel-2.* This test kernel will blink and write alternately "Hello, Kernel World!" and "Goodbye, Kernel World!" over the UART in perpetuity, with approximately one second of waiting in between each output. This is useful for testing serial connection over the UART.

3.3.3. *test-kernel-3.* This test kernel will first setup certain pins on the RPi2 for JTAG communication, and then blink and write alternately "Hello, kernel world!" and "Goodbye, kernel world!" over the UART in perpetuity, with approximately two seconds of waiting in between each output. This is useful for testing both JTAG and serial connection.

4. Booting over a network

Start by cloning the GitHub repository `raspberrypi/firmware` (if you have not already done so by following the steps above) and copy the contents of the `boot` folder to your SD card, as described before. Then, open the file `config.txt` on your SD card, and change it to read `kernel=u-boot.bin`, instead of where `kernel` pointed to before.

First, we have to download U-Boot. Download it by executing the command

```
git clone https://github.com/swarren/u-boot.git
```

which will create U-Boot to a directory at your current location. After that is done, we get the required cross-compiler by `sudo apt-get install gcc-arm-linux-gnueabihf`. Also install U-Boot tools with `sudo apt-get install u-boot-tools`. Now open a terminal window in the folder you have U-Boot in, and execute the command `make rpi_2_defconfig`, followed by `CROSS_COMPILE=arm-linux-gnueabihf- make -j8`. After the build is finished, copy all files starting with `u-boot` to your SD card.

Next, you will need to create a script that U-Boot will run once it is started. Create a file called `boot.scr`, and enter the following:

```
usb start
setenv scriptaddr 0x02000000
setenv serverip 130.229.149.236
setenv ipaddr 130.237.224.169
setenv bootargs "console=tty0"
tftp ${kernel_addr_r} zImage
bootz ${kernel_addr_r}
```

Figure 3. Boot script for U-Boot

where you replace the IP address after `serverip` by the one of your computer (which will act as a TFTP server later on). Your IP address can be found by looking at your `inet addr` after executing the `ifconfig` command in a terminal window.

Also replace the IP address after `ipaddr` in the boot script by an address belonging to your local network which is currently unoccupied - this will be the IP address of the RPi2. As a strategy, choose an address close to yours (alter only numbers near the right end of the IP) and ping it, using the command `ping 130.237.224.169`, where you replace the IP address with the one you want as the IP address of your device.

It might also be worth noting that `usb start` scans bus 0 for storage and Ethernet devices, which include your Ethernet port. Without this line, you will not be able to connect to your computer via TFTP. The `scriptaddr` is explicitly defined in the script for pedagogic reasons. This might not be needed, but you need to remember to not overwrite U-Boot on memory with the boot script.

This bootscript presupposes that you have a monitor connected to the RPi2 via a HDMI cable. If you instead rely on serial connection to see what is happening (not recommended at this point), you should replace `console=tty0` with `console=ttyAMA0` (or simply use both). Bootargs usually also sets properties of the file system, but since we are not using it in our trivial examples, it is not needed here.

Now that we have written the bootscript, we want to convert it to an image. Do so by executing the command `mkimage -A arm -O linux -T script -C none -n boot_script -d boot.scr boot.scr.uimg`. The result will be stored in `boot.scr.uimg` - transfer this file to your SD card.

Now, you can eject your SD card and place it in your RPi2.

We are now going to set up your computer as a TFTP server. First, we install the required tools by executing the command `sudo apt-get install xinetd tftpd tftp`. Edit (or create, if it does not exist) `tftp` in `etc/xinet.d/`, which should contain the following:

```
service tftp
{
    protocol        = udp
    port            = 69
    socket_type     = dgram
    wait            = yes
    user            = nobody
    server          = /usr/sbin/in.tftpd
    server_args     = /tftpboot
    disable         = no
}
```

FIGURE 4. TFTP server configuration

Then, create a directory called `tftpboot` in `/`. Make sure you do not have too stingy access rights by executing the commands `sudo chmod -R 777 /tftpboot` and `sudo chown -R nobody /tftpboot` in the directory where you created `tftpboot`. Then, make your changes take effect by executing `sudo service xinetd restart`.

Now, we need to prepare the kernel in question we want to remote boot. The boot script requires us to trick bootz into believeing it has received

a Linux kernel. To do this, we must insert a so-called "magic number" `0x016F2818` into the code. Here, you will need a hex editor to verify what you are doing. Get one with the command `sudo apt-get install bless`. Enter the `boot.S` (or whatever the ARM assembler part of your kernel is named) and declare a dummy variable (with no name) by inserting the line `.word 0x016f2818` outside of a function. Compile the kernel, and then search for "18286F01" in the image file (not the ELF file) using the hex editor. These bytes should be found starting at the offset `0x24`. If they are at a higher offset in the program, move the declaration up in the `boot.S` file. If the bytes are found at a too low offset, declare more dummy variables (`.word 0x01010101`, and so on...) before the magic number until you can find it at the correct offset. Finally, when this is done, rename your kernel `zImage` and move it to the `tftpboot` directory.

The setup is now complete. Plug in an Ethernet cable and a HDMI cable connected to a monitor to your RPi2 and power it from micro-USB to observe your program booting from U-Boot.

If at any point you should shut down your computer, you will need to start the server again by writing `sudo /etc/init.d/xinetd restart`, and take a second look if the IP addresses in your boot script can still be used; otherwise you will need to change the bootscript, create the boot script image again, and transfer it to the SD card.

5. Setting up JTAG

Communicating over JTAG with your PC can be very useful when debugging code on an external device. We will therefore look into the matter of using a JTAG adapter for this purpose. In this guide, we will be using a FT2232H Mini-Module [4] [3], which is very cheap, fast, and allows us to communicate both over UART and JTAG at the same time using only one adapter.

First, we need to be sure that the hardware is working. The FT2232H Mini-Module has two pin blocks, which are distinguished by their names "CN2" and "CN3" (channel 2 and channel 3, respectively) which can be found at the edge of the pin blocks on both sides of the board. The pins on the blocks are physically numbered starting from the pin marked by a square on one side of the board as pin 1, the pin which is on the same row of two as pin 1 being pin 2, and so on. So one of the two long rows of the pin blocks consists of odd numbers, and vice versa.

The adapter is not powered unless you make some connections between the pins on the board itself. For this purpose, you will need to use jumper cables.

Connect jumper cables between the pins as described in the list above, and then plug a mini-USB cable into the adapter on one end and your PC on the other end. At this point, you should be able to find the adapter listed when using `lsusb` and the two different device names of the different channels when using `dmesg | grep usb` (using the connections above, `/dev/tty0` for the JTAG and `/dev/tty1` for UART).

When you have verified that the above works, disconnect the adapter from its power source and set up the cabling described below.

If you happen to have a FT4232H Mini-Module instead, the connections at this point are identical except for the UART, where you would instead connect pins 8 and 10 on the RPi2 to the CN2-17 and the CN2-18 pins on the adapter, respectively. I have also listed the GPIO number of the RPi2 pins.

Next, before we plug this in, we need to install the software we need. To communicate over UART, we will simply use the tools we used before (`screen` or `minicom`). To communicate over JTAG, we will need the latest version of OpenOCD. For following this guide, we recommend at least version 0.9.0. In general, for getting the latest version of this tool, I recommend cloning the OpenOCD Git repository. First, you need to acquire some dependencies. Do so by executing the command `sudo apt-get install make libtool pkg-config autoconf automake texinfo`.

You can use either FTDI's own drivers (ftd2xx) or the open-source driver library libFTDI. At some point in time, the ftd2xx driver was much faster, although the issues with using it consists of not being able to redistribute your compiled code. Currently, I think they are about as fast and so it is more convenient for us to use libFTDI. Install libFTDI by executing the command `sudo apt-get install libusb-1.0-0`.

- VBUS to VCC:
  - CN3-1 to CN3-3
- V3V3 to VIO:
  - CN2-1 to CN2-11
  - CN2-3 to CN3-12
  - CN2-5 to CN2-21
  - CN2-9 to CN2-10

FIGURE 5. Trivial jumper cable connections

- JTAG: (FT2232H board left, RPi2 right)
  - TCK: CN2-7 to 22 (GPIO number 25)
  - TDI: CN2-10 to 7 (GPIO number 4)
  - TDO: CN2-9 to 18 (GPIO number 24)
  - TMS: CN2-12 to 13 (GPIO number 27)
- UART:
  - TXRX: CN3-26 to 10 (GPIO number 15)
  - RXTX: CN3-25 to 8 (GPIO number 14)
- Ground:
  - GND: CN2-2 to 6
- RPi2 to RPi2:
  - V3V3 to TRST: 1 to 15 (GPIO number 22)
- FTDI board to FTDI board:
  - V3V3 to VCCIO: CN2-1 to CN2-11
  - VBUS to VCC: CN3-1 to CN3-3

FIGURE 6. JTAG and UART jumper cable connections

Now clone the OpenOCD Git repository by executing the command `git clone git://git.code.sf.net/p/openocd/code`. 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`.

OpenOCD is now installed! Now, you need to run it. For this, you will need two configuration files. One for the JTAG adapter and one for the SoC board. In other words, use something similar to:

```
openocd -f interface/ADAPTER.cfg -f target/MYTARGET.cfg
```

where you replace `interface/ADAPTER.cfg` and `target/MYTARGET.cfg` with the corresponding files for your JTAG adapter and the board you want to debug on. In our case this becomes

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

You might not have a file for the RPi2 on your OpenOCD installation, though. If you do not, you must create one. The installed OpenOCD scripts are stored at `/usr/local/share/OpenOCD/scripts`. To easily create new files, execute the command `gksudo thunar` (do `sudo apt-get install gksudo` if you do not have this utility), and use the Thunar graphical file manager to navigate the the directory of the scripts and create and modify files there. Specifically for the RPi2 script, create a new file containing the Tcl

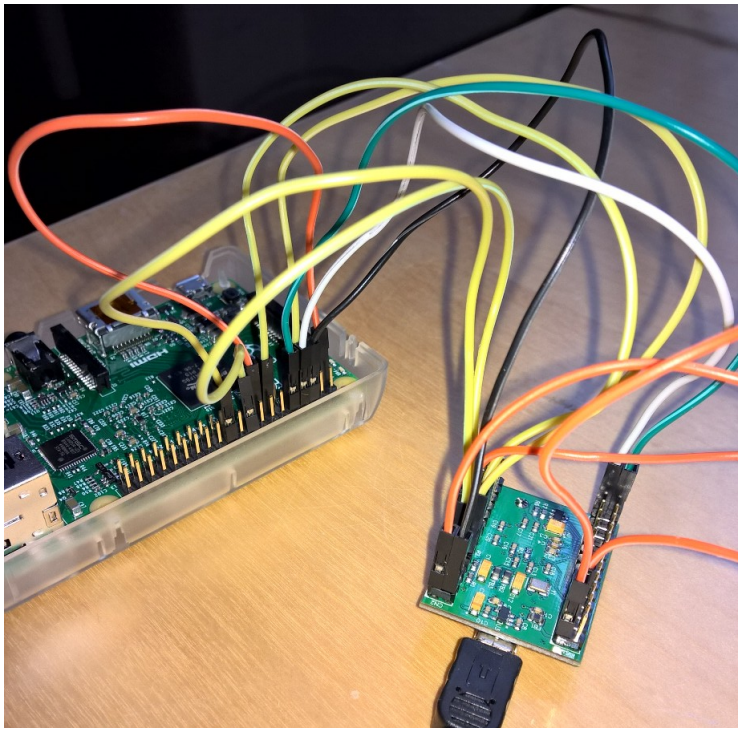

FIGURE 7. The FT2232H Mini-Module as it looks connected to the RPi2 via both JTAG (yellow cables), ground (black cable) and the UART (green and white cable)

code found at `rpi2-port/raspberrypi2.cfg`, in the `guancio/kth-on-rpi2` GitHub repository - clone this with

```
git clone https://github.com/guancio/kth-on-rpi2
```

if you have not already done so.

Now, you will need to write a kernel which enables the pins you have connected your JTAG adapter for JTAG communication. One such example, which can be used with U-Boot as described above, can be found in the GitHub repository `guancio/kth-on-rpi2`, in the `rpi2-port` folder, named `test-kernel-3`. Start your RPi2 as described in the U-Boot section above[2], and plug the USB cable connected to your FT2232H Mini-Module into your computer. You must then wait for the pins to be enabled for JTAG use. You can tell this is the case when the green "OK" LED is blinking steadily with one-second intervals. Then, execute the command

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

5.1. **Connecting to the OpenOCD server.** Your OpenOCD should now detect the processor of your debug target. It will list the number of processors, and 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 that until you are completely finished debugging!

At this point, you want to open a new terminal window and connect to the OpenOCD server you have set up, for example by executing the command `telnet localhost 4444` to communicate via Telnet. That will put you in communication with OpenOCD and you will be able to execute a number of commands in that environment. First, try the command `help` to see what you can do. For a simple test, you can execute the command `targets` followed by `halt` and then `targets` again to verify that you can use JTAG (you should see that the processors have been halted). Execute the command `exit` to exit the Telnet connection. Remember that the pins on the Raspberry Pi 2 are not configured as JTAG pins from the start, so if you reset your board by unplugging and plugging in the power, you might want also to re-connect with OpenOCD.

5.2. **Using both UART and JTAG.** Now, we also want to be able to communicate via UART at the same time. At some point before you connect to OpenOCD and halt the processors (if you have already done this, shut down OpenOCD and remove power from your RPi2, then connect to power and re-connect OpenOCD as described above), you can connect to one of the channels of the adapter. Use `dmesg | grep usb` and look for lines which say `FTDI USB Serial Device converter now attached to ttyUSBX`, where `X` is an integer. If you have connected OpenOCD, you will see that one of these have been disconnected - that is where the adapter communicates via JTAG with OpenOCD. Connect to the remaining one using `sudo screen /dev/ttyUSBY 115200`, where `Y` is the number of the remaining connection (if you are unsure, this problem is quickly solved by trial-and-error). Using any of the test kernels described in this section, you should see the lines

[2]If you do not want to use U-Boot, change all occurences of `0x1000000` in `linker.ld` to `0x8000` and make sure that the function `uart_disable` is commented out inside the function `kernel_main` inside `kernel.c`.

"Hello, kernel world!" and "Goodbye, kernel world!" printed alternately in perpetuity.

5.3. **Using JTAG with the GDB debugger.** You might want some more easily accessible and more useful tools when debugging. One which is very easy to use is the GDB functionality in Emacs. You probably already have Emacs on your computer, if not, execute the command `sudo apt-get install emacs24` (where you might want to replace 24 with a higher version number). You will also need a cross-debugger. Get one by executing `sudo apt-get install gdb-arm-none-eabi`.

Open Emacs. Press the "meta" button (Escape or maybe left Alt on your keyboard) and then x. You will get a prompt where you can write `gdb`, and press Enter. Upon doing that, you will get another prompt where something along the lines of `gdb -i=mi` is already filled in for you. Change this to `arm-none-eabi-gdb -i=mi`. GDB will now start. But once again press "meta" (Escape) and then x, type `gdb-many-windows` and then Enter. This will give you more frames, which is universally considered a good thing.

Now, you need to connect to the OpenOCD server you have set up before. At the `(gdb)` prompt in one of the Emacs frames, write `target extended-remote localhost:3333`. You are now connected remotely to the OpenOCD server. This can be confirmed by looking at the OpenOCD server output in the other terminal window.

To be able to debug successfully, you will need to know which processor core the registers you are looking at belongs to. At the `(gdb)` prompt, write `monitor cortex_a smp_gdb`. Now, in the terminal window with the OpenOCD server running, you should see something along the lines of `gdb coreid 0 -> -1`, where in this case we are looking at processor core zero. Now, this might not be the processor which is actually executing our program. To be sure you are looking at the right processor, you must set a breakpoint at an address inside your main program the processor will reach - this is done live in the GDB session, but first you must make some preparations.

First, it will be handy to load the symbol file (the ELF file you created your binary from). Do this by executing the command `symbol-file path/to/file` at the `(gdb)` prompt, where `path/to/` is replaced by the path to your file relative to the home directory, and `file` is replaced by the name of your file. For example, the ELF file for our test kernel 3 is called simply `test-kernel-3`.

Second, you will need to look at the disassembly of your binary. Execute the command

```
arm-none-eabi-objdump -D test-kernel-3 > /tmp/test-kernel-3.asm
```

and then open the resulting file for example by `gedit /tmp/test-kernel-3.asm`, where you substitute `text-kernel-3.asm` for the name of your disassembled binary and `gedit` for your favourite text viewer. In the disassembled code, you can see the ARM assembly instructions of all the different functions of your program, and which physical memory addresses these correspond to.

In order to set a breakpoint, you can either use the name of a function (in which case the breakpoint will be reached immediately inside that function call) or a physical address (in which case you use the physical address, the leftmost number in the disassembly). Examples of commands you can use at the GDB prompt to set breakpoints are `b *0x1000020` to suspend a processor core reaching the address `0x1000020`, or `b strlen` to suspend the processor which calls `strlen`. When a core reaches a breakpoint and suspends execution, you will be handling that core automatically with your GDB-in-Emacs! To verify this, either look at the registers in their separate Emacs frames or execute `info registers` at the `(gdb)` prompt. The program counter (the `pc` register) should be at the address you put your breakpoint. Then, you know the registers you are looking at belong to the right processor.

One thing we might be particularly interested in is changing the value of a register. This can be done through the command `set $r0 = 0x4711`, where you can swap `r0` for the name of the register you want to change and `0x4711` for another value. Then, execute the command `c` to continue. When you are finished, execute `quit`.

To a limited degree, you can also use the Emacs UI to toggle breakpoints and other things. There are a gazillion useful GDB commands. If you want to learn more, you can type `help` at the `gdb` prompt.

References

[1] Broadcom Europe Ltd. *BCM2835 ARM Peripherals*, 02 2012.
[2] Do, V. *SICS Thin Hypervisor Reference Manual*, 2014.
[3] Future Technology Devices International Ltd. *FT2232H Mini Module Datasheet*, 2011.
[4] Future Technology Devices International Ltd. *FT2232H Datasheet*, 2012.
[5] pchan. Raspberry Pi 2 Model B GPIO 40 Pin Block Pinout.
[6] The Raspberry Pi Foundation. *Raspberry Pi Downloads*.
[7] Sidebotham, B. Bare Metal Programming in C.
[8] Welch, D. Raspberry Pi ARM based bare metal examples .