Remote Debugging of Raspberry Pi with JTAG interface

Arseny Kurnikov
Aalto University School of Electrical Engineering
PO Box 13000, FI-00076 Aalto
Espoo, Finland
arseny.kurnikov@aalto.fi

ABSTRACT

This paper discusses the JTAG (Join Test Action Group) standard and its use to obtain debug information from circuits and chips. This standard describes a protocol that gives the access to a very low-level outputs from the chips where this technology is enabled. The main described scenario is using this protocol on a Raspberry Pi (RPi) board. In particular, it comes with an ARM (Acorn RISC Machine, RISC - reduced instruction set computing) processor, and the ARM processor family has become dominant in the modern computing, especially with the wide usage of smartphones. Thus, understanding how a low-level debugging can be performed on this machine gives an important experience for anyone interested in embedded systems development.

Another significant part of the paper is designated to debugging Linux kernel on RPi . It is not possible to run Windows natively on ARM processors, so Linux is the most popular general purpose OS (Operating System) for RPi. The Linux kernel real-time debugging is of a great interest as well. JTAG debugging is not enabled by default on RPi. Providing a way to enable JTAG on RPi in order to debug a Linux kernel is the main goal of the paper.

Keywords

JTAG, Raspberry Pi, Linux Kernel

1. INTRODUCTION

JTAG [9] is the standard for a generic transport interface for integrated circuits. One possible application of it is a boundary scan of printed circuit boards. If such a board has special chips, so called debug ports, then another option is to use debuggers for single stepping, breakpointing, analyzing memory, and so on. This is how JTAG will be used in this paper.

Many silicon architectures have built-in software support for debugging, including ARM CPU of RPi [7]. It is accessed through test access ports (TAPs) via a JTAG adapter. A TAP might be a module inside one chip or a separate chip by itself. To communicate with a JTAG target the host connects to JTAG pins by the means of the adapter. The adapter takes care of actually forming the necessary signals and it is connected to the host computer through USB, Ethernet, PCI or another interface. Different boards require different adapters. Other characteristics of an adapter include [5]:

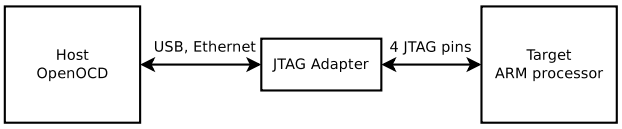

Figure 1: Principal scheme

- Throughput
- Voltage range
- Supported host software

In order to utilize the described hardware, the host machine should have a special software, i.e. the debugger. The one that is used in this work is the Open On-Chip Debugger (OpenOCD) [6].

Finally, the target of debugging is the RPi ARM processor [1]. It has a built-in JTAG module that can be controlled from the host machine to halt the execution, examine the registers state, read and write to the main memory. Overall, the physical layout of the connections, hardware and software is presented on Figure 1.

The paper is organized as follows. The next chapter gives the theoretical overview of JTAG, ARM JTAG, and OpenOCD. Chapter 3 describes how to enable JTAG on RPi. It also contains practical debug session examples. Chapter 4 focuses on debugging Linux kernel on RPi. The last chapter draws the conclusions.

2. OVERVIEW

In this chapter, an introduction to JTAG is given. In particular, the signals and the state machine of a JTAG TAP are described. OpenOCD is introduced, as well. RPi JTAG signals can be accessed via General Purpose Input Output (GPIO) pins of the RPi board [8].

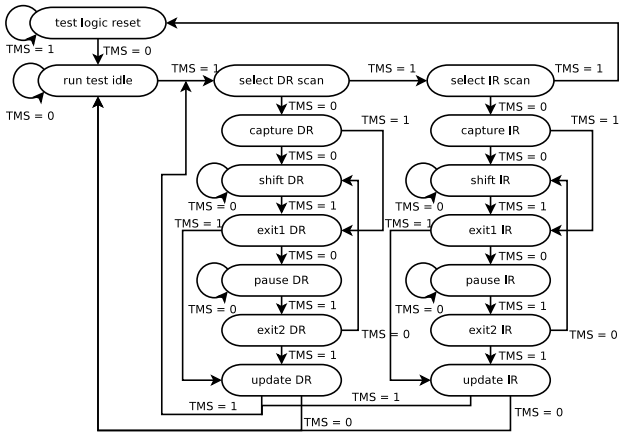

Figure 2: JTAG TAP state transition diagram

2.1 JTAG

JTAG is a generic transport interface for debugging circuits or using the boundary scan. The core logic of JTAG is described by the state machine shown in Figure 2. There are two registers: Data Register (DR), and Instruction Register (IR). IR selects which particular data register is used. The data is read or written on DR. The registers and the state of JTAG is controlled by four signals, corresponding to four pins of JTAG:

- Test Data In (TDI) – when set to 1, the data is written to the selected register.
- Test Data Out (TDO) – for reading the data.
- Test Clock (TC) – synchronization signal. All other signals should appear in accordance to the rise of TC. Its frequency determines the speed of the adapter.
- Test Mode Select (TMS) – controls the state diagram transitions.

For example, to reset the state it is necessary to clock TMS at least 5 times. The `shift` states are the ones that allow to input data to a register using TDI. The actual update of the whole register happens by reaching the `update` state. The data is read with TDO from the `capture` state.

This is the basis of JTAG, and the rest of the functionality varies from one chip to another. For example, ARM JTAG [10] introduces several optional signals, like "System Reset" (TRST), or "Debug Request". The later one defines the hardware halt of the processor. The former one is analogous to 5 TMS signals.

2.2 JTAG on Raspberry Pi GPIO

RPi board contains GPIO pins that can be configured for different purpose inputs and outputs. There are several alternative configurations, and one of them enables JTAG functionality. The function configurations are marked `ALT0 - ALT5`. The configuration of GPIO occurs via several function selection registers: `GPFSEL0 - GPFSEL5`. The exact steps to enable JTAG are presented in Table 1, where `GPIOn` corresponds to GPIO pin number n:

Table 1: RPi GPIO configuration for JTAG

Register	Pin	Configuration	JTAG signal
GPFSEL0	GPIO4	ALT5	TDI
GPFSEL2	GPIO22	ALT4	TRST
GPFSEL2	GPIO24	ALT4	TDO
GPFSEL2	GPIO25	ALT4	TCK
GPFSEL2	GPIO27	ALT4	TMS

The basis for `GPFSEL` registers lies at the memory address `0x20200000`. The length of each register is 4 bytes, so that `GPFSEL2` is at `0x20200008`. Enabling JTAG on RPi can be achieved by reading the memory at the selection registers 0 and 2, setting the necessary bits (i.e. `ALT5` corresponds to '011' in binary, and `ALT4` is '010') at the location of the corresponding GPIO pins, and writing the result back to the selection registers. Writing a bare metal program that accomplishes this task in C or in Assembler is straightforward.

2.3 Cross-compiling

In order to compile the software for RPi it is possible either to do it natively on RPi itself, or to use a host machine, which is usually a preferable way, because the host computer is generally faster than an RPi. However it requires a so-called cross-compiling "toolchain" to compile for RPi from a machine with `x86` processor, for example.

There are ready-made toolchains available, or it can be built from the source files. When building from source the compiler, the assembler, the linker and other tools must be configured to produce output files for the target architecture [2]. It is necessary to understand the difference between the following terms:

- Host – the machine where the toolchain will be run to produce the output files.
- Build – the machine where the toolchain itself is being built.
- Target – the machine for which the toolchain will produce outputs.

Then, during the configuration phase of the build, the necessary variables should be set according to the environment, and later the toolchain can be used to produce executables for the target architecture.

2.4 RPi Booting

Simply put, RPi booting consists of two stages. First, GPU (Graphics Processing Unit) of RPi boots from its ROM (read-only memory). The main advantage of having this process is that it is highly unlikely to render RPi completely unusable. It is said that RPi is "unbrickable". For the first stage the following files are required to be put on the SD card: `bootcode.bin`, `loader.elf`, and `start.elf`. After that, on the second stage, the user provided `kernel.img` will be loaded into the memory location `0x8000` and will be jumped to by the loader.

This "kernel" image can be anything: from a bare metal program that does nothing but blinks the board LEDs to a

Linux kernel. It will be the main point of interest for the rest of the paper.

2.5 OpenOCD

The OpenOCD is an open-source tool for debugging embedded systems. It runs on the host machine as a server, allowing connections over Telnet or from GDB (The GNU Project Debugger) [3] in order to debug the target systems, connected to the host machine through a JTAG adapter.

Since there are many variant of the adapters, as well as many JTAG implementations, OpenOCD should be configured to work with a particular adapter and the board. For example, the configuration file for Olimex ARM-JTAG-USB adapter is the following:

```
interface ft2232
ft2232_device_desc
  "Olimex OpenOCD JTAG ARM-USB-TINY-H"
ft2232_layout olimex-jtag
ft2232_vid_pid 0x15ba 0x002a
```

It uses `FT2232` drivers for the USB interface and configures it accordingly. The OpenOCD configuration file for RPi is:

```
# Broadcom 2835 on Raspberry Pi

telnet_port 4444
gdb_port 5555
#tcl_port 0

adapter_khz 1000

if { [info exists CHIPNAME] } {
   set  _CHIPNAME $CHIPNAME
} else {
   set  _CHIPNAME raspi
}

reset_config none

if { [info exists CPU_TAPID ] } {
   set _CPU_TAPID $CPU_TAPID
} else {
   set _CPU_TAPID 0x07b7617F
}
jtag newtap $_CHIPNAME arm
  -irlen 5 -expected-id $_CPU_TAPID

set _TARGETNAME $_CHIPNAME.arm
target create $_TARGETNAME
  arm11 -chain-position $_TARGETNAME
```

First, it sets up the ports for Telnet and GDB connections. Then a new JTAG TAP is created. The mandatory parameter `irlen` defines how many bits the instruction register contains. The optional `expected-id` helps OpenOCD to detect when the connected board does not actually contain the required TAP. Finally, the new target for an ARM processor is created.

OpenOCD works according to a client/server architecture. It allows to configure ports on the host machine, where Telnet or GDB connections are accepted. Telnet session gives access to the commands of OpenOCD itself, whereas GDB session provides a wider set of capabilities like, for example, loading a symbol file.

The GDB command to enable remote debugging is `target remote`. It should be followed by the address and the port number of the machine that is running OpenOCD. If it is the same computer that is used as a host for both debuggers and the OpenOCD configuration specified GDB port to be 5555, then the command would be `target remote localhost:5555`.

After GDB is connected to OpenOCD it is necessary to tell GDB what symbols are available for the program that is being run on the board. The GDB command is `symbol-file`. For example, if the JTAG enabling program was compiled with debugging information, then it is possible to load the symbols from its executable to GDB and perform typical debugging tasks, like single stepping, examining memory and registers, and so on.

OpenOCD commands can still be accessed from GDB via the `monitor` command.

3. DEBUG SESSION

In this chapter, an example of a debug session is provided. Some operations are shown and outputs are given. The status of the processor and the memory can be monitored and modified. A useful example of loading binary images into memory without rebooting RPi is described.

3.1 Execution, registers, memory

After the hardware is setup and OpenOCD configuration files are downloaded, the debug session can start. An example output of an OpenOCD running instance is the following:

```
Open On-Chip Debugger 0.5.0 (2011-08-11-06:56)
Licensed under GNU GPL v2
For bug reports, read
 http://openocd.berlios.de/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
1000 kHz
none separate
raspi.arm
Info : max TCK change to: 30000 kHz
Info : clock speed 1000 kHz
Info : JTAG tap: raspi.arm tap/device found:
  0x07b7617f (mfg: 0x0bf, part: 0x7b76, ver: 0x0)
Info : found ARM1176
Info : raspi.arm: hardware has 6 breakpoints,
   2 watchpoints
Info : accepting 'telnet' connection from 4444
target state: halted
target halted in ARM state due to debug-request,
  current mode: Supervisor
cpsr: 0x800001d3 pc: 0x0000813c
```

OpenOCD opens the configured ports for Telnet or GDB and accepts the connections on those ports. A typical Telnet session will start with `halt` command. After issuing it, the processor will be halted, and other OpenOCD commands can be given. The execution can be continued with `resume` command, that has an optional argument – the memory address, which the execution will be resumed from. Another flow control command is `step` that perform a single stepping.

OpenOCD comes with a set of commands to introspect the status of the processor and memory. Processor registers can be displayed with `reg` command over Telnet. For example, register 15 is the program counter (`pc`). It points to the instruction that is to be executed. Modifying it corresponds to issuing `step` command with the parameter - the address from which to perform the next step.

The memory area can be displayed with a set of `mdX` commands, where 'X' is one of 'b', 'w', ... for a byte, a word and so on. The memory can also be written to with a set of `mwX` commands. For example, knowing the RPi memory layout, where `0x20200000` is a GPIO base address, `(base + 0x28)` is `GPSET0`, and `(base + 0x1c)` is `GPCLR0` register; and the fact that `0x10000`, the 16th bit, corresponds to GPIO LED, it is possible to turn the RPi green LED on and off from an OpenOCD Telnet session:

```
>mww 0x20200028 0x10000 // turns LED on
>mww 0x2020001c 0x10000 // turns LED off
```

3.2 Loading images

In order to start debugging a program it is necessary to put it on the SD card first. This process takes a lot of time after each recompilation. One needs to power-off RPi, take the SD card out and put it in the host card reader, copy the file, unmount the SD card, put it back to RPi, power it on. JTAG provides a more convenient way to get the program into the memory for debugging purposes.

When JTAG is enabled and the processor is halted, one can upload the program image directly to the memory over JTAG. The required OpenOCD command is `load_image <file_name> <address>`, where the image to be uploaded is given as the first parameter, and it will be loaded to the address given as the second parameter. GDB has a command to load images as well, `load`. It will automatically put the executable to the correct address, specified at the compilation time. The uploading speed depends solely on the hardware solution used for JTAG. Olimex ARM-USB-TINY-H shows the speed of up to 40 KiB/s. So 2 MiB file can be uploaded in 50 seconds. The speed of writing to an SD card is much bigger but a lot of time is wasted on physically moving it from an RPi to the host computer and backwards. For small images, like bare metal programs, downloading over JTAG is definitely faster.

4. LINUX KERNEL

This section describes how to enable JTAG on RPi for debugging Linux kernel. Linux is a default choice as a general purpose OS for RPi [4]. Kernel debugging is not a trivial task, but doing it remotely simplifies the approach a lot. By default, JTAG is not enabled in the kernel, so special steps are needed before the kernel can be debugged over JTAG.

4.1 Compressed kernel and the Raspberry Pi image

The kernel compilation for RPi follows similar steps as for `x86` processors, for example. As with bare metal programs, the compiler should be configured to build programs for ARM processors.

The kernel expects to be loaded at the address `0x8000`. But usually the kernel image is put on the booting partition in a compressed form as (`zImage`) along with the decompressor. The decompressor is ignorant to where it is loaded to, and it is smart enough to uncompress the kernel and put it to the correct memory address. This is important because when building an image for the RPi that will enable JTAG before booting the kernel, the actual `zImage` will be shifted in memory to introduce the JTAG enabling code in front of it.

4.2 Enabling JTAG before decompressing the kernel

The RPi community provides the script to create the image `kernel.img` that can be put to the card from a newly compiled Linux kernel. The script puts a small bootloader at the address 0x0, the kernel arguments at the address 0x100 and `zImage` at 0x8000. The bootloader and the arguments are provided as text files with hexadecimal strings most probably converted from binary files.

Following the same idea, it is possible to convert the bare metal JTAG enabling program into a text file with the required hexadecimal strings and put it at 0x8000, so that JTAG gets enabled first. In this case, `zImage` is appended after that code.

In order to be able to debug the kernel booting process from the very beginning, the JTAG enabling software can go into an infinite loop, so that the kernel booting should be started manually: by opening an OpenOCD session, halting the execution, and jumping to the address, where `zImage` ended up being loaded to.

So, the kernel can be debugged over JTAG from the very beginning of the booting process. But during its execution some kernel drivers might override the GPIO pins used for JTAG for their purposes. All GPIO JTAG pins have some alternative configurations for other functions. Possible overriding candidates are I^2C (Inter-Integrated Circuit) and SPI (Serial Peripheral Interface) drivers. Neither I^2C, nor SPI signal pins overlap with the ones of JTAG. More than that, both I^2C and SPI drivers are disabled in the kernel by default.

4.3 Kernel session debugging example

When the kernel image with JTAG enabling code is put to the SD card it is possible to debug the kernel with GDB. After RPi is turned on and OpenOCD is started, GDB can connect to OpenOCD over GDB protocol. The command to attach to the process running on RPi is `target remote`, that takes the address and port where OpenOCD is running.

OpenOCD commands can be issued from GDB with its `monitor` command. For example, `monitor halt` equals to halting the processor from a Telnet session. In the same way the disassembled code can be retrieved via `monitor arm disassemble`. All other OpenOCD functions available through Telnet protocol work from GDB as well.

To start booting the kernel it is necessary to manually continue the execution from the address where the decompressor of the kernel got loaded to. For the tool described earlier it is `0x81b0`. Hence `monitor step 0x81b0` should be given.

In order to make sense out of the kernel debugging it is necessary to load symbol tables from the kernel image. First of all, the kernel should be configured to include the debugging information during its building process. Then the tables can be loaded by `symbol-file vmlinux`, where `vmlinux` is the uncompressed kernel image containing debugging information.

When the symbols are loaded, then after the kernel is decompressed, the debugging of it can go along the same lines as a normal debugging of a user-space program on any Linux machine. It is possible to set breakpoints, examine registers and memory locations, do single stepping, continue execution from another place and so on.

An example output of a GDB session is the following:

```
GNU gdb (GDB) 7.4.1-debian
...
This GDB was configured as "arm-linux-gnueabi".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) symbol-file vmlinux
Reading symbols from vmlinux...done.
(gdb) target remote localhost:5555
(gdb) monitor halt
(gdb) b start_kernel
At 0xc04f34a0: file init/main.c, line 467.
(gdb) monitor resume 0x81b0
```

GDB finds where the function is located and shows it, when creating a breakpoint.

5. SUMMARY

This paper gave a brief introduction to JTAG as it is used for debugging embedded systems. The practical setup included debugging Raspberry Pi ARM processor using OpenOCD over Olimex ARM-JTAG-USB adapter. Debugging bare metal programs is enabled by configuring RPi GPIO pins, so that they correspond to JTAG signals. Debugging Linux kernel can be enabled by configuring the necessary pins before the kernel starts to boot. Then it is possible, for example, to single step through the booting sequence and figure out what the kernel does during its boot. It is possible to debug kernel modules as well by setting the necessary breakpoints and examining the call stack, the memory, the registers state.

JTAG debugging is a powerful instrument, that gives a lot of capabilities for an embedded systems developer.

6. REFERENCES

[1] ARM, home page. `http://www.arm.com`.

[2] Autotools manual. `http://www.gnu.org/software/automake/manual/`.

[3] GDB: The GNU Project Debugger, home page. `http://www.gnu.org/software/gdb/`.

[4] Linux for embedded systems. `http://www.elinux.org`.

[5] Olimex ARM-USB-TINY-H. `https://www.olimex.com/Products/ARM/JTAG/ARM-USB-TINY-H/`.

[6] OpenOCD, documentation. `http://openocd.sourceforge.net/doc/html/index.html`.

[7] Raspberry Pi, home page. `http://www.raspberrypi.org`.

[8] Broadcom Corporation. *BCM2835 ARM Peripherals*, 2012.

[9] IEEE, Computer Society. *1149.1-2001 - IEEE Standard Test Access Port and Boundary Scan Architecture*, 2001.

[10] Lauterbach GmbH. *ARM JTAG Interface Specification*, 2013. `http://www2.lauterbach.com/pdf/arm_app_jtag.pdf`.