Kernel Course: Lecture 19

Hardware Tools in Embedded Kernel Development

Sam Protsenko
<joe.skb7@gmail.com>

June 12, 2020

Agenda

1. Multimeter
2. Scope
3. Logic Analyzer
4. JTAG

Hardware Tools Overview

Where it can be useful?

- Debugging hardware issues (obviously)
- Board bringup (measuring clocks, voltages, etc)
- Fixing adapter drivers
- Performance optimizations
- Power management optimizations
- Investigating low-level components
- Debugging subtle problems

Multimeter

Multimeter Overview

Masures constant electrical values:

- Voltage:
  - $R_V = \infty$
  - Can be used for power issues diagnostics
  - Helpful during board bring-up
- Current:
  - $R_A = 0$
  - Don't measure the current in power outlets!
  - Useful for power management improvements
  - Sometimes "10A" input jack must be used
- Resistance:
  - Resistors must be unsoldered before measurement
  - Use beeper mode to check a loop

Multimeter Demo: Suspend/Resume Current

Current Measurement

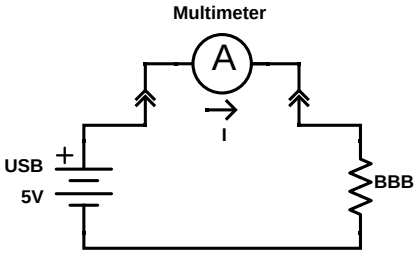


Figure 1: Ammeter Connection

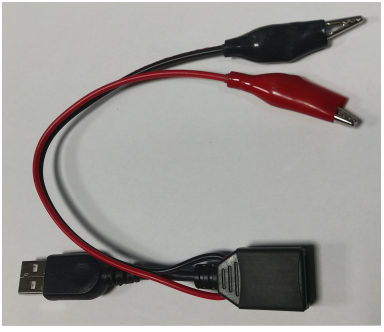


Figure 2: Measurement Cable

Power can be calculated:

$$P = V \cdot I$$

Current Measurement Setup

Figure 3: Setup for BBB suspend/resume current measurement

Measuring BBB suspend/resume current (page 1/3)

Figure 4: Current draw: before suspend ($I = 310\,\mathrm{mA}$)

Measuring BBB suspend/resume current (page 2/3)

Figure 5: Current draw: during suspend ($I = 20\,\mathrm{mA}$)

Measuring BBB suspend/resume current (page 3/3)

Figure 6: Current draw: after resume ($I = 310\,\mathrm{mA}$)

Scope

Scope Overview

- Software vs hardware
- Parameters: frequency, data rate; + probes (capacity)
- Trigger feature
- Divisors (/1, /10)
- 2 channels (common ground!)
- Scale: time, voltage
- “Auto” button
- Measure mode

Useful:

- To debug fast-changing stuff (can’t be done in software)
- To see what is actually happening on the line

Scope Demo #1: Measuring Delay

Button and LED Device

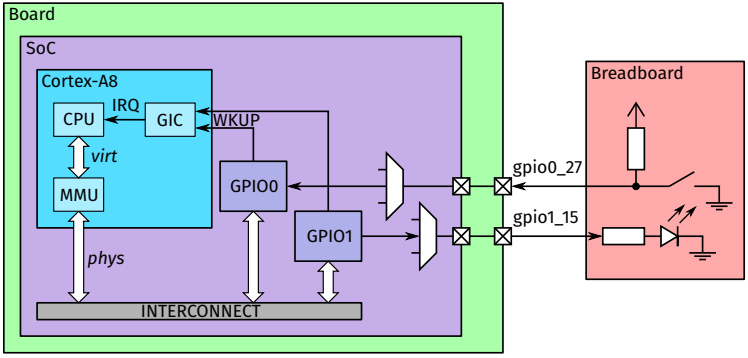


Figure 7: Button and LED Device: Functional Block Diagram

- There is a delay between button press and LED toggle events (`hw1.c`)

Scope Setup

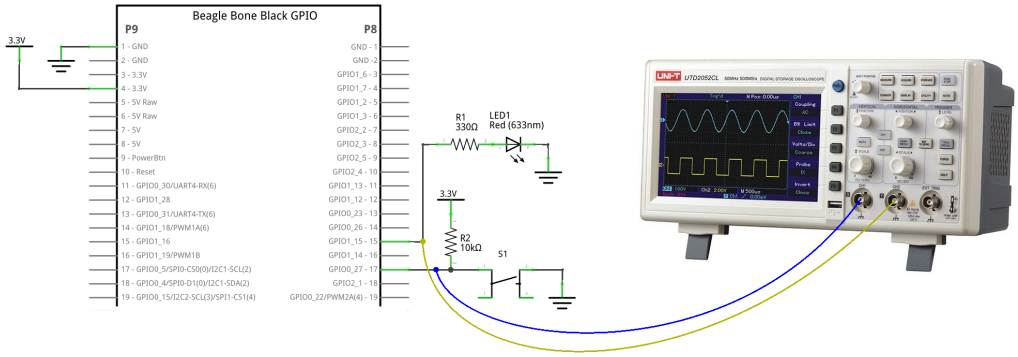


Figure 8: Setup for button/LED delay measurement

Scope Settings

Channel settings:

- CH1 (blue): Button line
- CH2 (yellow): LED line
- Coupling: DC
- Probe: 10X
- Voltage scale: $1\,\mathrm{V/div}$
- Time scale: $1\,\mathrm{ms/div}$

Trigger menu:

- Type: Edge; Slope: Rise,Fall
- Source: CH2 (LED line)
- Mode: Normal
- Coupling: DC
- Trigger level: $1\,\mathrm{V}$ ($0\,\mathrm{V}$ … $3.3\,\mathrm{V}$)

Measuring Delay (page 1/2)

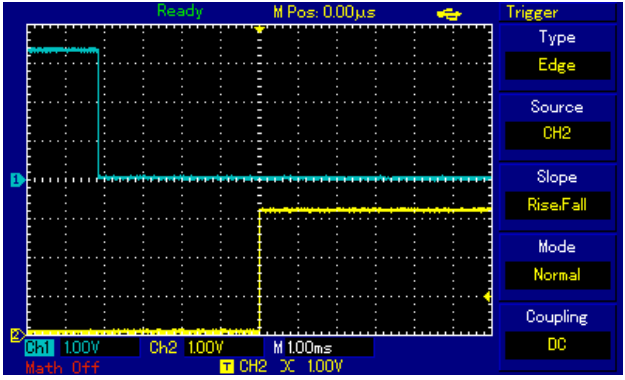


Figure 9: Scope screenshot: LED off → on

Measuring Delay (page 2/2)

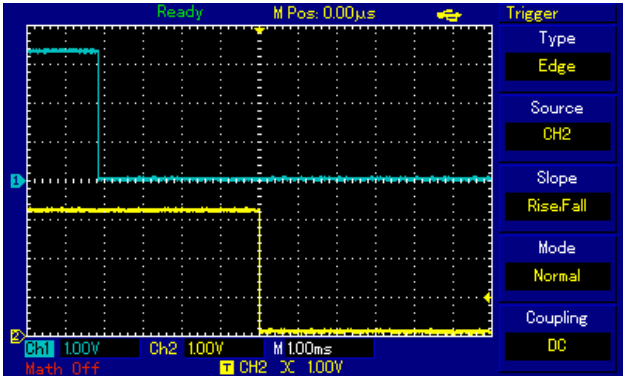


Figure 10: Scope screenshot: LED on → off

Scope Demo #2: Investigating Kernel Sleeps

Square Wave Module (1/3)

Listing 1: sqw.c

```
1  #include <linux/module.h>
2  #include <linux/gpio.h>
3  #include <linux/interrupt.h>
4  #include <linux/kthread.h>
5  #include <linux/delay.h>
6
7  #define AM335_GPIO(bank,line)   (32 * bank + line)
8  #define LED_GPIO                AM335_GPIO(1, 15)
9
10 static struct task_struct *thread;
11 static int led_val;
12
13 static int sqw_thread_func(void *data)
14 {
15         while (!kthread_should_stop()) {
16                 gpio_set_value(LED_GPIO, led_val);
17                 led_val ^= 0x1;
18                 /* Documentation/timers/timers-howto.txt */
19                 /* Don't use udelay(), we need to yield to scheduler */
20                 usleep_range(20, 20); /* can be 10 us - 20 ms */
```

Square Wave Module (2/3)

```
21          }
22
23          return 0;
24  }
25
26  static int __init sqw_init(void)
27  {
28          int err;
29
30          err = gpio_request(LED_GPIO, "my_led");
31          if (err) {
32                  pr_err("Unable to request LED GPIO\n");
33                  return -EINVAL;
34          }
35          gpio_direction_output(LED_GPIO, 0); /* init LED value = 0 */
36
37          thread = kthread_run(sqw_thread_func, NULL, "sqw_thread");
38          if (IS_ERR(thread)) {
39                  pr_err("kthread_run() failed\n");
40                  err = PTR_ERR(thread);
41                  return err;
42          }
43
```

Square Wave Module (3/3)

```
44          return 0;
45  }
46
47  static void __exit sqw_exit(void)
48  {
49          if (thread)
50                  kthread_stop(thread);
51          gpio_free(LED_GPIO);
52  }
53
54  module_init(sqw_init);
55  module_exit(sqw_exit);
56
57  MODULE_AUTHOR("Sam Protsenko <joe.skb7@gmail.com>");
58  MODULE_DESCRIPTION("Square Wave Module");
59  MODULE_LICENSE("GPL");
```

Load CPU script (1/1)

Listing 2: load_cpu.sh

```
#!/bin/sh

fulload() {
        dd if=/dev/zero of=/dev/null | \
        dd if=/dev/zero of=/dev/null | \
        dd if=/dev/zero of=/dev/null | \
        dd if=/dev/zero of=/dev/null &
}

fulload
read
killall dd
```

Stable Scope Display

To make periodic signal “stand still”, trigger must be configured:

- Type: Edge
- Source: CH2
- Slope: Rise
- Mode: Auto
- Coupling: DC
- Trigger level: put it in the wave center

Or just use “Auto” button to configure that for you.

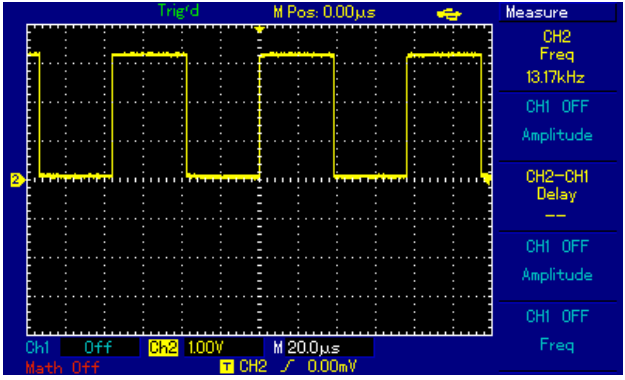


Figure 11: Scope screenshot: CPU load 0%

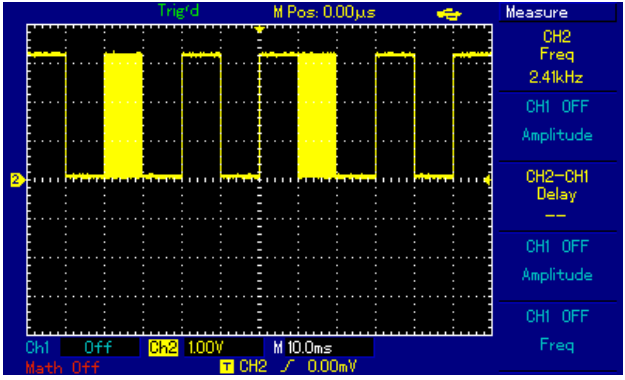


Figure 12: Scope screenshot: CPU load 100%

Scope Demo #3: Investigating I2C Transmission

Probes

- Use “10X” mode for regular operation:
  - $C_{in} = 20\,\text{pF} \Rightarrow T = R_{PU} \cdot C_{in} = 50\,\text{k}\Omega \cdot 20\,\text{pF} = 1\,\mu\text{s}$
- Use “1X” mode for low voltage signals:
  - $C_{in} = 100\,\text{pF} \Rightarrow T = R_{PU} \cdot C_{in} = 50\,\text{k}\Omega \cdot 100\,\text{pF} = 5\,\mu\text{s}$

$$T_{SDC} = \frac{1}{100\,\text{kHz}} = 10\,\mu\text{s}$$

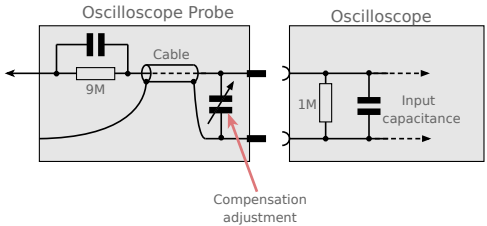


Figure 13: Oscilloscope probe (X10) circuit

Checking I2C Transmission (page 1/2)

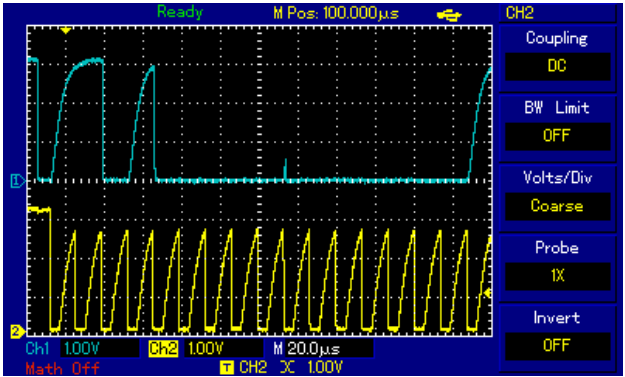


Figure 14: Scope screenshot: I2C Transmission using X1 probe

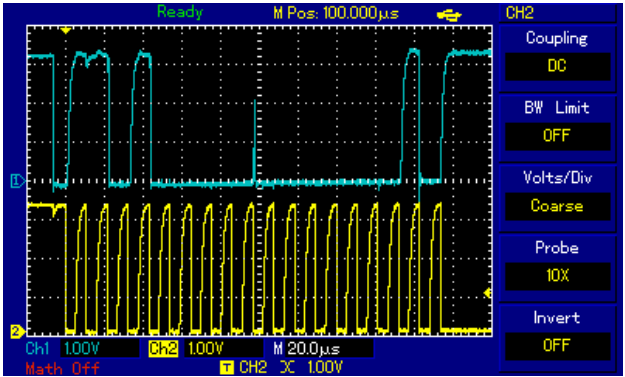


Figure 15: Scope screenshot: I2C Transmission using X10 probe

Logic Analyzer

Logic Analyzer Overview

- Catches digital levels
- Stores collected data to internal (hardware) buffer
- App can parse protocols (I2C, UART)
- Parameters: max data rate (freq), ports count, protocols support
- Some specialized LAs exist (e.g. USB)
- **Useful**:
  - For problem isolation / root-causing
  - For reverse engineering (“sniffing”)

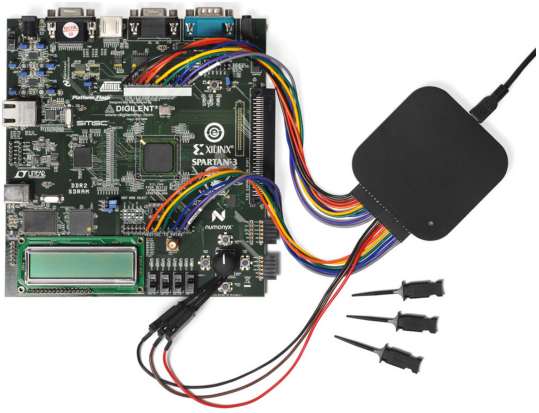


Figure 16: Saleae 16 Logic Analyzer

LA Demo #1: I2C (SDA, SCL)

I2C data exchange with RTC (page 1/3)

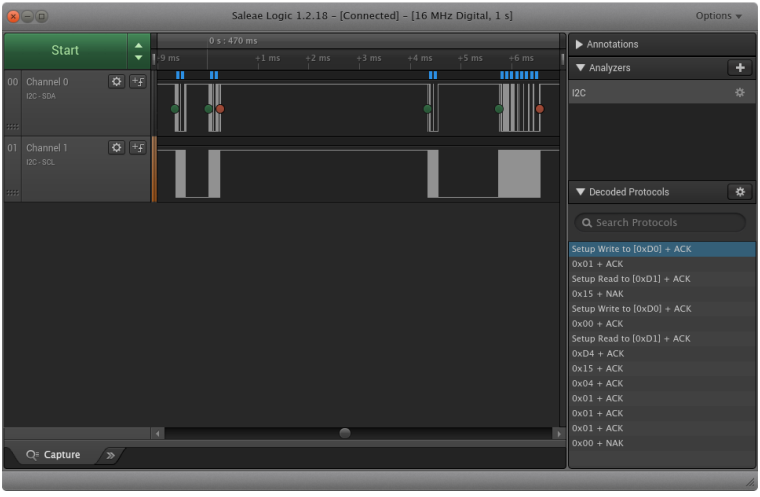


Figure 17: I2C data exchange with DS1307 sniffed by Logic Analyzer

I2C data exchange with RTC (page 2/3)

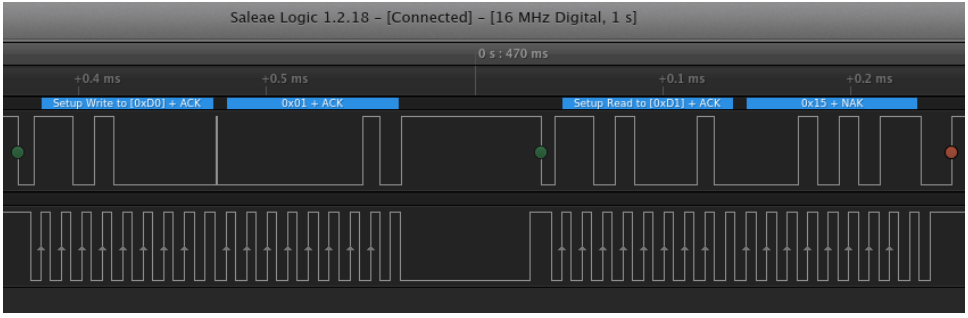


Figure 18: I2C read from 0x01 register

```
[15347.357045] ### my_rtc probe(): enter
[15347.362433] ### read data = 0x15
```

I2C data exchange with RTC (page 3/3)

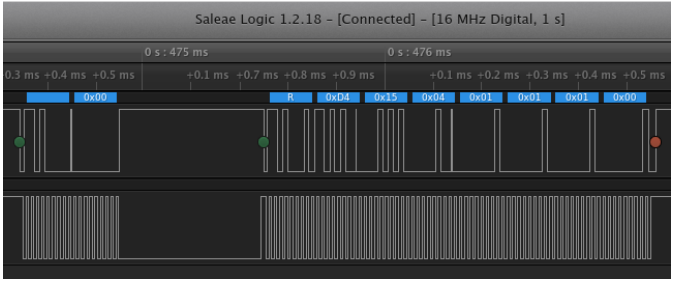


Figure 19: I2C bulk read (starting from 0x00 register)

```
[15347.368436] ds1307x 2-0068: reg 0x0  = 0xd4
[15347.372824] ds1307x 2-0068: reg 0x1  = 0x15
[15347.377113] ds1307x 2-0068: reg 0x2  = 0x4
[15347.381395] ds1307x 2-0068: reg 0x3  = 0x1
[15347.385593] ds1307x 2-0068: reg 0x4  = 0x1
[15347.389808] ds1307x 2-0068: reg 0x5  = 0x1
[15347.394005] ds1307x 2-0068: reg 0x6  = 0x0
```

LA Demo #2: UART (Tx, Rx)

LA UART Setup

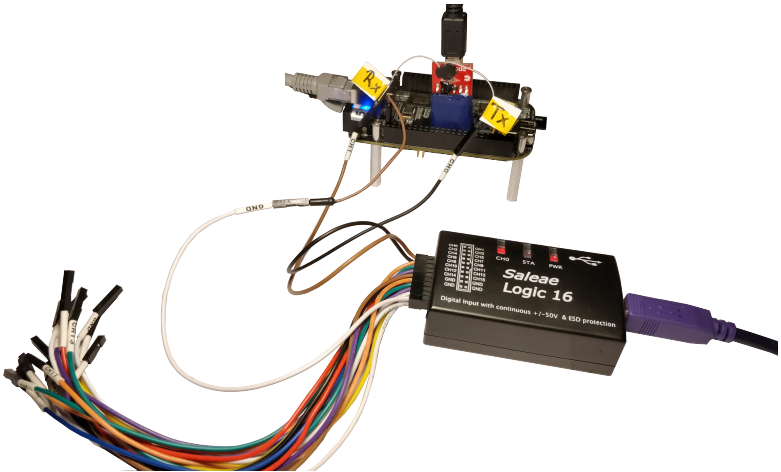


Figure 20: Setup for tracing UART with Logic Analyzer

UART data exchange with serial console (page 1/3)

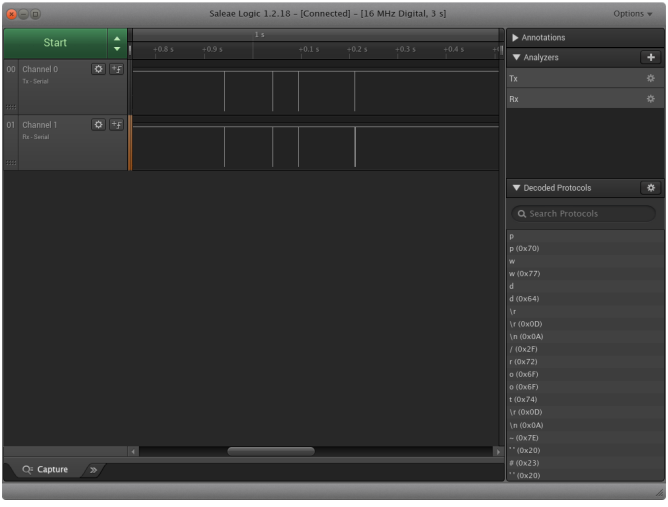


Figure 21: UART data exchange with FTDI sniffed by Logic Analyzer

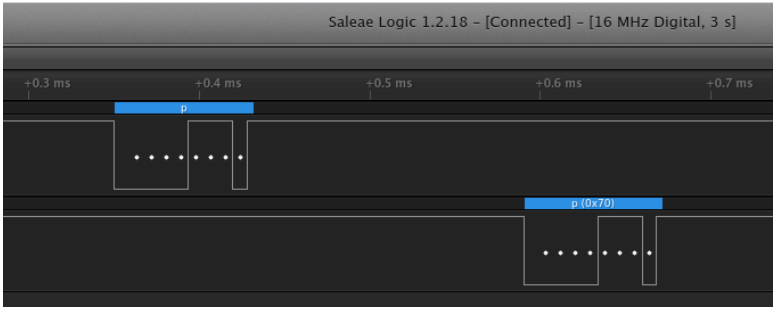


Figure 22: Type in ‘p’ character and get it back in minicom

UART data exchange with serial console (page 3/3)

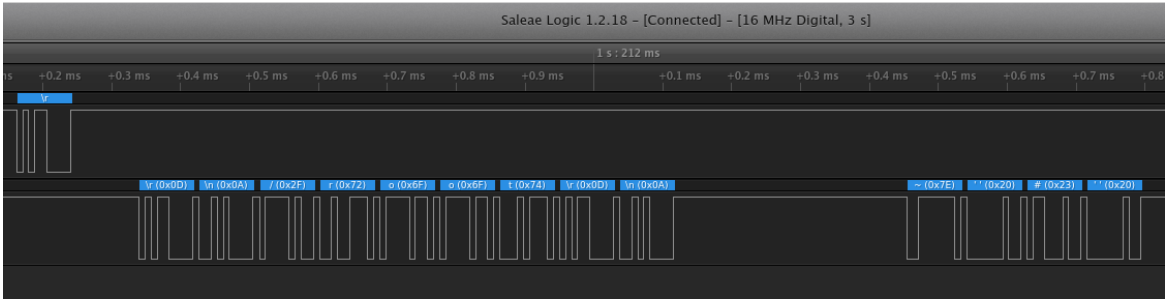


Figure 23: **pwd** command response

Take Five

JTAG

JTAG Intro

Allows one to debug program running on board just like user-space application with GDB:

- Breakpoints
- Read/write registers
- Stop/run CPU
- Read/write memory
- Load images
- Examine stack
- Can debug U-Boot, kernel, user-space apps/libs, even ROM-code...

JTAG Bus

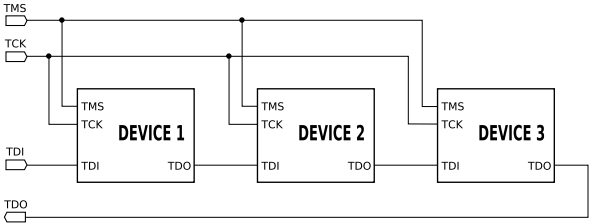


Figure 24: Daisy-chained JTAG

- JTAG is a serial bus (IEEE1149.1 standard)
- AM335x implements JTAG (has JTAG pins)
- PC doesn't have JTAG controller, so **JTAG adapter** must be used

JTAG Adapters

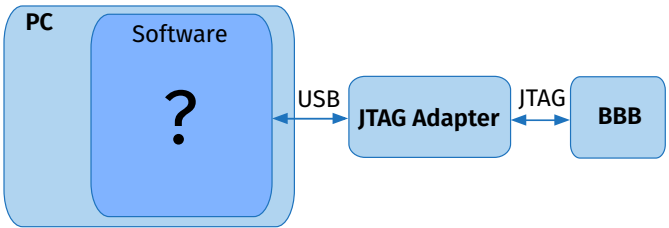


Figure 25: Connecting BBB JTAG to PC

Adapters require some software. Some examples for BBB:

- **Flyswatter 2** (FT2232H based, MPSSE): OpenOCD
- **XDS100v2**: Proprietary software

JTAG support in AM335x

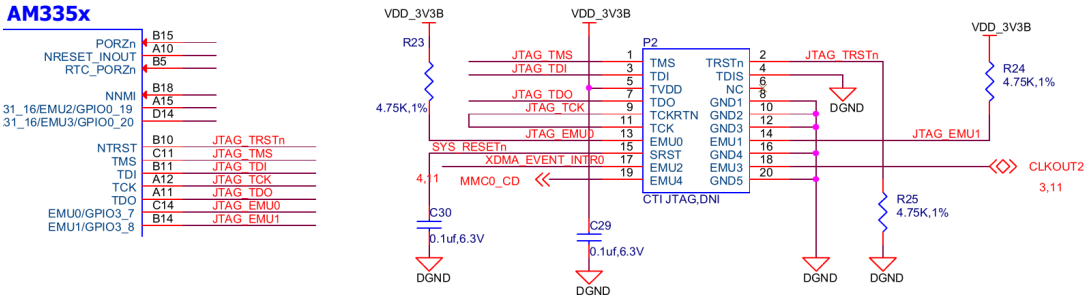


Figure 26: JTAG connector in BBB

See TRM for details on debug interface implementation in AM335x.

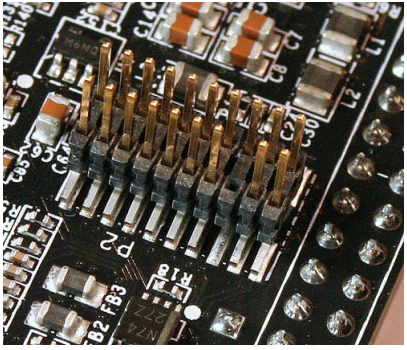


Figure 27: BBB JTAG Connector

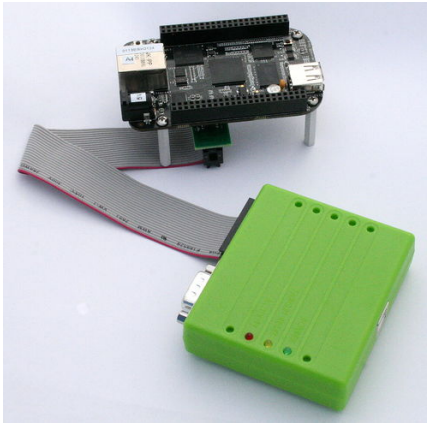


Figure 28: BBB and Flyswatter2

OpenOCD Overview

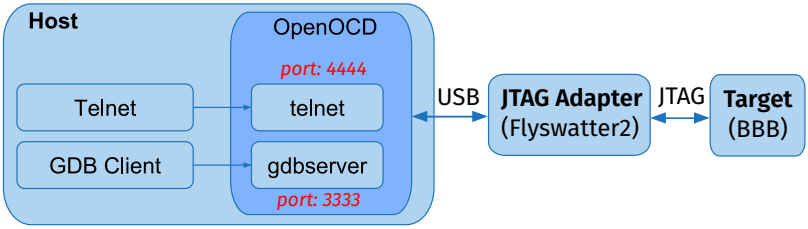


Figure 29: Debugging BBB with OpenOCD

OpenOCD creates servers for:

- Telnet (port 4444 on localhost)
- GDB (port 3333 on localhost)

Once OpenOCD is running, you can run:

```
1        $ telnet localhost 4444
2          or
3        $ arm-eabi-gdb vmlinux
```

Kernel Preparations

- Enable `CONFIG_DEBUG_INFO` (the same as `-g`)
- Disable `CONFIG_WATCHDOG` (just in case)
- Make sure JTAG clock (DEBUGSS on BBB) is alive in kernel (see next slide)
- Rebuild, add `zImage` to rootfs, flash to BBB

Kernel Preparations (cont'd)

`arch/arm/mach-omap2/omap_hwmod_33xx_data.c`:

```
1 static struct omap_hwmod am33xx_debugss_hwmod = {
2         .name           = "debugss",
3         .class          = &am33xx_debugss_hwmod_class,
4         .clkdm_name     = "l3_aon_clkdm",
5 +       .flags          = (HWMOD_INIT_NO_IDLE | HWMOD_INIT_NO_RESET),
6         .main_clk       = "trace_clk_div_ck",
7         .prcm           = {
```

Without this patch you'll see `STICKY BIT` errors (in OpenOCD terminal) when kernel starts running.

OpenOCD Preparations

Install OpenOCD (my version is 0.10.0):

```
$ sudo apt update
$ sudo apt install openocd
```

- Figure out correct OpenOCD interface and target config files:
  - `interface/ftdi/flyswatter2.cfg`
  - `target/am335x.cfg`
- ...but those files from upstream OpenOCD won't work
- Use Flyswatter2 config from TinCanTools site (see next page)

OpenOCD Working Script for Flyswatter2 (1/5)

Listing 3: ti_beaglebone_with_fs2_mod.cfg

```
# AM335x Beaglebone, for use with the TinCanTools Flyswatter2
#  http://beagleboard.org/bone
#  http://www.tincantools.com

# The JTAG interface is built directly on the board.
interface ftdi
ftdi_device_desc "Flyswatter2"
ftdi_vid_pid 0x0403 0x6010

ftdi_layout_init 0x0538 0x057b
ftdi_layout_signal LED -ndata 0x0400
ftdi_layout_signal nTRST -data 0x0010
ftdi_layout_signal nSRST -data 0x0020 -noe 0x0100

adapter_khz 16000


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

OpenOCD Working Script for Flyswatter2 (2/5)

```
21      set _CHIPNAME am335x
22  }
23
24  # This chip contains an IcePick-D JTAG router.  The IcePick-C configuration is almost
25  # compatible, but it doesn't work.  For now, we will just embed the IcePick-D
26  # routines here.
27  proc icepick_d_tapenable {jrc port} {
28          # select router
29          irscan $jrc 7 -endstate IRPAUSE
30          drscan $jrc 8 0x89 -endstate DRPAUSE
31
32          # set ip control
33          irscan $jrc 2 -endstate IRPAUSE
34          drscan $jrc 32 [expr 0xa0002108 + ($port << 24)] -endstate DRPAUSE
35
36          # for icepick_D
37          irscan $jrc 2 -endstate IRPAUSE
38          drscan $jrc 32 0xe0002008 -endstate DRPAUSE
39
40          irscan $jrc 0x3F -endstate RUN/IDLE
41          runtest 10
42  }
43
```

OpenOCD Working Script for Flyswatter2 (3/5)

```
44 #
45 # M3 DAP
46 #
47 if { [info exists M3_DAP_TAPID] } {
48         set _M3_DAP_TAPID $M3_DAP_TAPID
49 } else {
50         set _M3_DAP_TAPID 0x4b6b902f
51 }
52 jtag newtap $_CHIPNAME m3_dap -irlen 4 -ircapture 0x1 -irmask 0xf -expected-id $_M3_DAP_TAPID -disable
53 jtag configure $_CHIPNAME.m3_dap -event tap-enable "icepick_d_tapenable $_CHIPNAME.jrc 11"
54
55 #
56 # Main DAP
57 #
58 if { [info exists DAP_TAPID] } {
59    set _DAP_TAPID $DAP_TAPID
60 } else {
61    set _DAP_TAPID 0x4b6b902f
62 }
63 jtag newtap $_CHIPNAME dap -irlen 4 -ircapture 0x1 -irmask 0xf -expected-id $_DAP_TAPID -disable
64 jtag configure $_CHIPNAME.dap -event tap-enable "icepick_d_tapenable $_CHIPNAME.jrc 12"
65
66 #
```

OpenOCD Working Script for Flyswatter2 (4/5)

```
67 # ICEpick-D (JTAG route controller)
68 #
69 if { [info exists JRC_TAPID] } {
70    set _JRC_TAPID $JRC_TAPID
71 } else {
72    set _JRC_TAPID 0x0b94402f
73 }
74 jtag newtap $_CHIPNAME jrc -irlen 6 -ircapture 0x1 -irmask 0x3f -expected-id $_JRC_TAPID -ignore-version
75 jtag configure $_CHIPNAME.jrc -event setup "jtag tapenable $_CHIPNAME.dap"
76 # some TCK tycles are required to activate the DEBUG power domain
77 jtag configure $_CHIPNAME.jrc -event post-reset "runtest 100"
78
79 #
80 # Cortex A8 target
81 #
82 set _TARGETNAME $_CHIPNAME.cpu
83 target create $_TARGETNAME cortex_a8 -chain-position $_CHIPNAME.dap -dbgbase 0x80001000
84
85 # SRAM: 64K at 0x4030.0000; use the first 16K
86 $_TARGETNAME configure -work-area-phys 0x40300000 -work-area-size 0x4000
87
88 $_TARGETNAME configure -event gdb-attach {
89    cortex_a dbginit
```

OpenOCD Working Script for Flyswatter2 (5/5)

```
90     halt
91 }
92
93
94 reset_config trst_and_srst
```

Run OpenOCD

1. Connect Flyswatter2 to BBB using adapter kit
2. Connect Flyswatter2 to PC via USB
3. Connect BBB to PC via USB (power);
   “TARGET RESET” LED on Flyswatter is glowing, CPU is in reset state
4. Run OpenOCD command:

```
$ openocd -f ./ti_beaglebone_with_fs2_mod.cfg -c "init" -c "reset init"
```

We will use this terminal to track OpenOCD log.

Correct OpenOCD Command Output

```
adapter speed: 16000 kHz
Info : auto-selecting first available session transport "jtag". To override use 'transport select <transport↩
    >'.
Warn : target name is deprecated use: 'cortex_a'
trst_and_srst separate srst_gates_jtag trst_push_pull srst_open_drain connect_deassert_srst
Info : ftdi: if you experience problems at higher adapter clocks, try the command "ftdi_tdo_sample_edge ↩
    falling"
Info : clock speed 16000 kHz

Info : JTAG tap: am335x.jrc tap/device found: 0x2b94402f (mfg: 0x017 (Texas Instruments), part: 0xb944, ver: 0x2)
Info : JTAG tap: am335x.dap enabled
Info : DAP transaction stalled (WAIT) - slowing down
Info : DAP transaction stalled (WAIT) - slowing down
Info : DAP transaction stalled (WAIT) - slowing down
Error: target->coreid 0 powered down!

Info : JTAG tap: am335x.jrc tap/device found: 0x2b94402f (mfg: 0x017 (Texas Instruments), part: 0xb944, ver: 0x2)
Info : JTAG tap: am335x.dap enabled
Info : DAP transaction stalled (WAIT) - slowing down
Info : DAP transaction stalled (WAIT) - slowing down
Info : DAP transaction stalled (WAIT) - slowing down
Info : am335x.cpu: hardware has 6 breakpoints, 2 watchpoints
```

OpenOCD over GDB (page 1)

Let's start debug session in GDB:

```
$ arm-eabi-gdb vmlinux

(gdb) target remote localhost:3333
(gdb) monitor cortex_a dacrfixup on
(gdb) continue
```

What we did here:

1. Connect to OpenOCD's GDB server, using Linux kernel symbols
2. Using OpenOCD `cortex_a` command, do a workaround for software breakpoints (Linux kernel maps `.text` read-only, and the debugger cannot write a breakpoint due to that)
3. Continue CPU execution (as it was in reset state initially);
   we use GDB commands for debugging (not `monitor resume`, etc)

OpenOCD over GDB (page 2)

Set a breakpoint and trigger it:

```
^C
(gdb) break do_sys_open
(gdb) continue

/ # cat /proc/cmdline
```

1. Stop CPU execution by pressing `Ctrl-C`
2. Set breakpoint for `do_sys_open()` kernel function (it's a handler for `open()` syscall. Another interesting functions would be `load_module()`, `start_kernel()`, etc
3. Continue CPU execution
4. Print `/proc/cmdline` file, so that `open()` is triggered
5. Now we are in breakpoint, CPU is halted

OpenOCD over GDB (page 3)

Investigate code in breakpoint:

```
(gdb) monitor cortex_a maskisr on
(gdb) continue
(gdb) ...
(gdb) bt, list, info registers, disas, step, print, ...
(gdb) monitor cortex_a maskisr off
(gdb) continue
```

1. Don’t process interrupts when stepping (otherwise we won’t be able to perform `continue`)
2. Wait for `open("/proc/cmdline")`
3. Investigate caught code
4. Once we are done, enable interrupts processing and continue normal execution

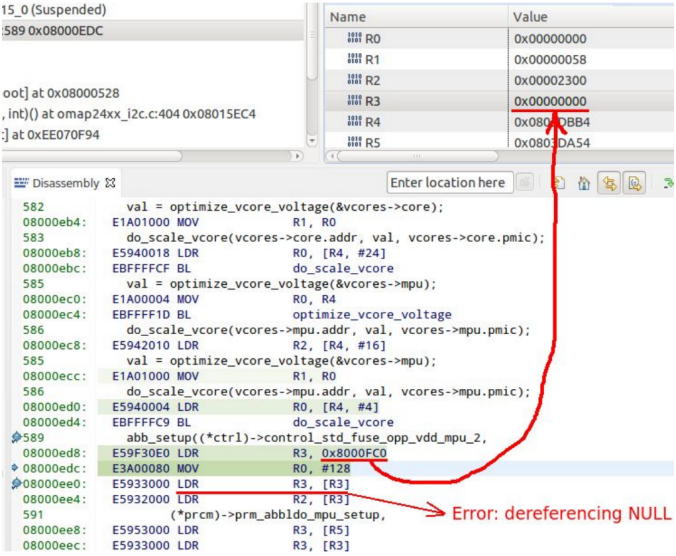
OpenOCD Telnet Commands

In Telnet, you can use regular OpenOCD monitor commands:

- help
- reset
- halt
- resume
- reg
- bp <address> <len> [hw]
- step [address]
- mdw, mww

See **OpenOCD User's Guide** for details.

Proprietary JTAG: XDS100v2 via CodeComposerStudio



JTAG Demo #1: Flyswatter2

JTAG Demo #2: XDS560v2

JTAG References

- https://www.tincantools.com/flyswatter2-beaglebone-black-how-to/
- https://e2e.ti.com/support/embedded/linux/f/354/t/363421
- https://www.tincantools.com/beaglebone-black-eclipse-gdb/
- https://devel.rtems.org/wiki/Debugging/OpenOCD/BeagleBoneBlack
- https://elinux.org/Debugging_The_Linux_Kernel_Using_Gdb
- https://github.com/n-aizu/freertos-multicore/wiki/Debugging-with-openocd
- http://openocd.org/doc/html/GDB-and-OpenOCD.html
- http://openocd.org/doc/html/General-Commands.html
- https://habr.com/post/206036/

Thank you!