Real-Time BeagleBone Interfacing

Chapter 13 – Exploring the BeagleBone

RealTime BeagleBone

- Real-time systems guarantee a response within a specified time
- *Hard real-time* systems are when systems fail if a deadline is missed (think: braking systems)
- *Soft real-time* systems are when a missed deadline may be reduced quality (think: streaming video)
- Mainline Linux Kernels typically meet soft real-time
- Other processes may be running when the real-time event occurs

Real-time Kernels

- Running a GPIO shell script can result in lots of *jitter*

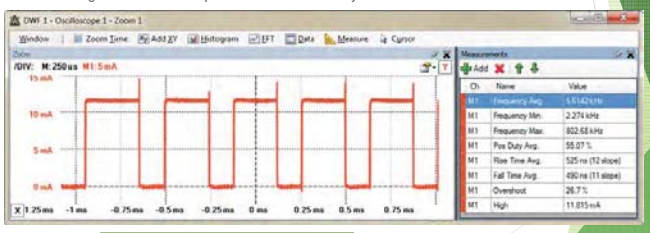
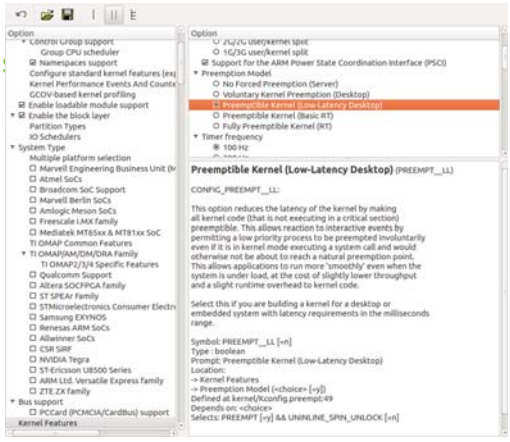


Figure 6-3 EBB

Real-time Kernels

- Using mmap() can help, but
- Linux is a *nonpreemptive* OS
- The kernel can be compiled using **`CONFIG_PREEMPT_RT_FULL=y`**
- You can load precompiled RT kernels…



Precompiled Kernels

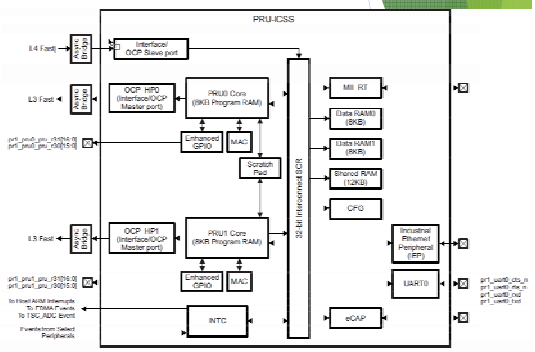
```
bone$ apt-cache pkgnames | grep linux-image-4.4.19
linux-image-4.4.19-ti-r41
linux-image-4.4.19-ti-r41-dbg
linux-image-4.4.19-ti-r42
linux-image-4.4.19-ti-r42-dbg
linux-image-4.4.19-ti-rt-r41
linux-image-4.4.19-ti-rt-r41-dbg
linux-image-4.4.19-ti-rt-r42
linux-image-4.4.19-ti-rt-r42-dbg
```

Real-time Hardware

- You can get a cape with an FPGA on it such as the Valent F(x) LOGi-Bone

Programmable Real-Time Unit (PRU)

- The Bone comes with two 32-bit 200 MHz RISC cores, PRUs
- Enhanced GPIO: fast GPIO on P8/P9 headers
- 32x32 Multiplier/Accumulator (MAC)
- Interrupt controller (INTC)
- UART0: 192 MHz on P8/P9



Important Documents

- **The AM335x PRU-ICSS Reference Guide: This document is the main** reference for the PRU-ICSS hardware (289 pages): tiny.cc/ebb1307
- **The Texas Instruments PRU Wiki: tiny.cc/ebb1308**
- ~~**The PRU Linux Application Loader API Guide: tiny.cc/ebb1309**~~
- **The PRU Debugger User Guide: tiny.cc/ebb1310**
- **Updated Debugger: https://github.com/MarkAYoder/BeagleBoard-exercises/tree/master/pru/prudebug**

TECH DAYS
Texas Instruments

Building Blocks for Programmable Real-Time Unit (PRU) Programming & Development

Jason Reeder & Melissa Watkins

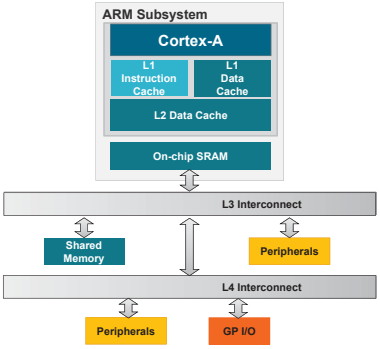
Agenda

TECH DAYS
Texas Instruments

- PRU Hardware Overview
- PRU Firmware Development
- ~~Linux Drivers Introduction~~
- ~~PRU Application Examples~~
- ~~Getting Started with the PRU~~

ARM SoC Architecture

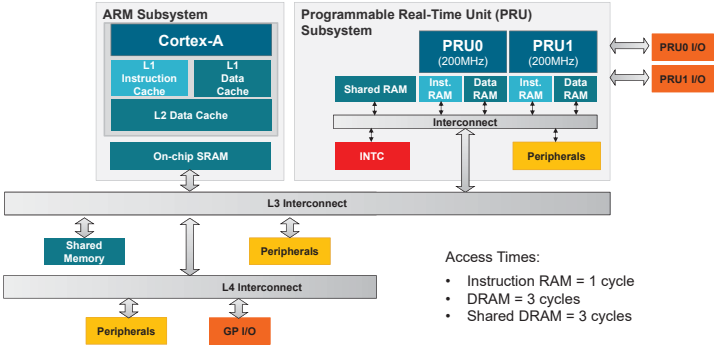
TECH DAYS
Texas Instruments



- L1 D/I caches:
  - Single cycle access
- L2 cache:
  - Min latency of 8 cycles
- Access to on-chip SRAM:
  - 20 cycles
- Access to shared memory over L3 Interconnect:
  - 40 cycles

ARM + PRU SoC Architecture

TECH DAYS
Texas Instruments



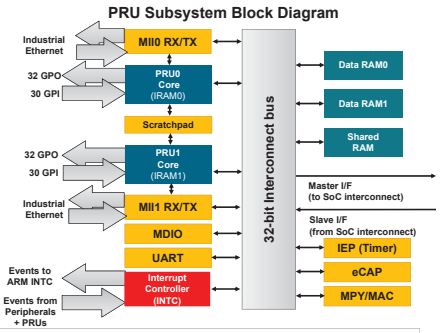
Access Times:

- Instruction RAM = 1 cycle
- DRAM = 3 cycles
- Shared DRAM = 3 cycles

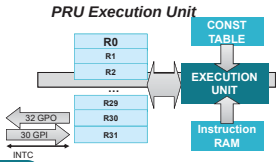
Now let's go a little deeper…

Programmable Real-Time Unit (PRU) Subsystem

- Programmable Real-Time Unit (PRU) is a low-latency microcontroller subsystem
- Two independent PRU execution units
  - 32-Bit RISC architecture
  - 200MHz – 5ns per instruction
  - Single cycle execution - No pipeline
  - Dedicated instruction and data RAM per core
  - Shared RAM
- Includes Interrupt Controller for system event handling
- Fast I/O interface
  - Up to 30 inputs and 32 outputs on external pins per PRU unit



PRU Functional Block Diagram



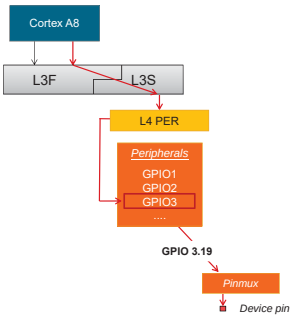
Constant Table
- Saves PRU cycles by providing frequently used peripheral base addresses

Execution Unit
- Logical, arithmetic, and flow control instructions
- Scalar, no Pipeline, Little Endian
- Single-cycle execution

Instruction RAM
- Typical size is a multiple of 4KB (or 1K Instructions)
- Can be updated with PRU reset

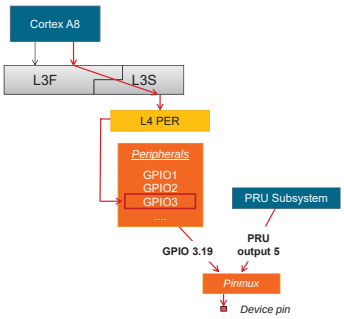
Special Registers (R30 and R31)
- R30
  - Write: 32 GPO
- R31
  - Read: 30 GPI + 2 Host Int status
  - Write: Generate INTC Event

Fast I/O Interface



Fast I/O Interface

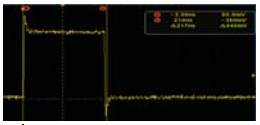
- Reduced latency through direct access to pins
  - Read or toggle I/O within a single PRU cycle
  - Detect and react to I/O event within two PRU cycles
- Independent general purpose inputs (GPIs) and general purpose outputs (GPOs)
  - PRU R31 directly reads from up to 30 GPI pins
  - PRU R30 directly writes up to 32 PRU GPOs
- Configurable I/O modes per PRU core
  - GP input modes
    - Direct connect
    - 16-bit parallel capture
    - 28-bit shift
  - GP output modes
    - Direct connect
    - Shift out



GPIO Toggle: Bench measurements

ARM GPIO Toggle:

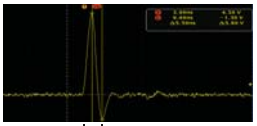
```
int main(){
    // Configure GPIO module, pinmuxing, etc.

    // Toggle system-level GPIO 3.19 from ARM core
    BitToggle(GPIO_INSTANCE_ADDRESS+GPIO_SETDATAOUT,
              GPIO_INSTANCE_ADDRESS+GPIO_CLEARDATAOUT);

    while(1);
}

unsigned long BitToggle(unsigned long val1, unsigned long val2){
    asm(
        "  mov r2, 0x00000000" "\n\t"
        "  str   r2,[r0]" "\n\t"      // Set GPIO 3.19
        "  str   r2,[r1]" "\n\t"      // Clear GPIO 3.19
    );
    return val1;
}
```



~200ns

PRU IO Toggle:

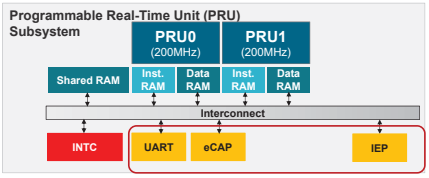
```
.origin 0
.entrypoint PRU_GPIO_TOGGLE

PRU_GPIO_TOGGLE:

    // Set PRU GPO 5
    SET     R30, R30, 5

    // Clear PRU GPO 5
    CLR     R30, R30, 5

    HALT
```



~5ns = ~40x Faster

Integrated Peripherals

- Provide reduced PRU read/write access latency compared to external peripherals
- Local peripherals don't need to go through external L3 or L4 interconnects
- Can be used by PRU or by the ARM as additional hardware peripherals on the device
- Integrated peripherals:
  - PRU UART
  - PRU eCAP
  - PRU IEP (Timer & DigIO)



PRU "Interrupts"

- The PRU does not support asynchronous interrupts.
  - However, specialized h/w and instructions facilitate efficient polling of system events.
  - The PRU-ICSS can also generate interrupts for the ARM, other PRU-ICSS, and sync events for EDMA.
- From UofT CSC469 lecture notes, "*Polling is like picking up your phone every few seconds to see if you have a call. Interrupts are like waiting for the phone to ring.*
  - *Interrupts win if processor has other work to do and event response time is not critical*
  - *Polling can be better if processor has to respond to an event ASAP*"
- Asynchronous interrupts can introduce jitter in execution time and generally reduce determinism. The PRU is optimized for highly deterministic operation.

Using PRU via remoteproc and RPMsg

- **Up to date details are here:** http://elinux.org/EBC_Exercise_30_PRU_via_remoteproc_and_RPMsg
- **To use the PRU you need to:**

1. Disable HDMI
2. Configure PRU compiler
3. Edit device tree
4. Set pin mux
5. Run example

Disable HDMI

- **If you get this…**

```
bone$ config-pin -q P8_45
P8_45 pinmux file not found!
cape-universala overlay not found
run "config-pin overlay cape-universala" to load the cape
```

- **You need to do this: Edit** `/boot/uEnv.xtx` **and remove # from:**

```
##BeagleBone Black: HDMI (Audio/Video) disabled:
dtb=am335x-boneblack-emmc-overlay.dtb
```

- **Reboot and you should see:**

```
bone$ config-pin -q P8_45
config-pin -q P8_45
P8_45 Mode: default Direction: in Value: 0
```

Configure PRU compiler

- **To configure the compiler**

```
bone$ export PRU_CGT=/usr/share/ti/cgt-pru
bone$ export PRU_SUPPORT=/opt/source/pru-software-support-package
bone$ cd $PRU_CGT
bone$ mkdir -p bin
bone$ cd bin
bone$ ln -s `which clpru` .
bone$ ln -s `which lnkpru` .
```

Make sure remoteproc is running

```
bone$ lsmod | grep pru
pru_rproc   13507 0
pruss_intc  7451  1 pru_rproc
pruss       10611 1 pru_rproc
```

- If you get the above response, you are good, move on
- Else, edit your device tree

Edit device tree

```
bone$ cd /opt/source/dtb-4.4-ti
bone$ nano src/arm/am335x-boneblack-emmc-overlay.dtb
```

- Change:

```
/* #include "am33xx-pruss-rproc.dtsi" */
```

- To:

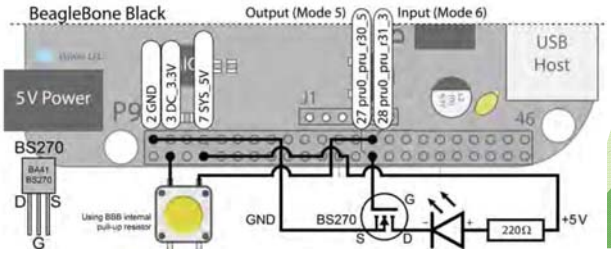
```
#include "am33xx-pruss-rproc.dtsi"
```

- Then make

```
bone$ make
bone$ make install
bone$ echo "blacklist uio_pruss" > /etc/modprobe.d/pruss-blacklist.conf
bone$ reboot
```

Set pin mux

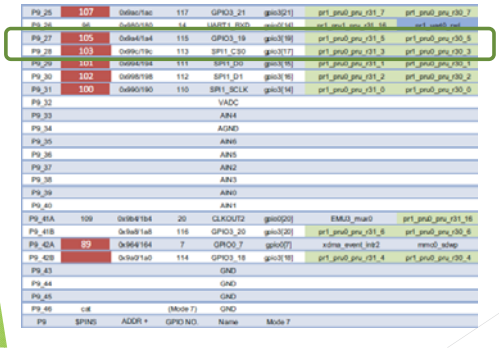
```
bone$ config-pin -a P9_27 pruout
bone$ config-pin -a P9_28 pruin
```



P8 Header `pr1_pru1_pru_r31_11`

P8_27	56	0x8e0/0e0	86	GPIO2_22	gpio2[22]	pr1_pru1_pru_r31_8	pr1_pru1_pru_r30_8
P8_28	58	0x8e8/0e8	88	GPIO2_24	gpio2[24]	pr1_pru1_pru_r31_10	pr1_pru1_pru_r30_10
P8_29	57	0x8e4/0e4	87	GPIO2_23	gpio2[23]	pr1_pru1_pru_r31_9	pr1_pru1_pru_r30_9
P8_30	59	0x8ec/0ec	89	GPIO2_25	gpio2[25]	pr1_pru1_pru_r31_11	pr1_pru1_pru_r30_11
P8_31	54	0x8d8/0d8	10	UART5_CTSN	gpio0[10]	uart5_ctsn	
P8_32	55	0x8dc/0dc	11	UART5_RTSN	gpio0[11]	uart5_rtsn	
P8_33	53	0x8d4/0d4	9	UART4_RTSN	gpio0[9]	uart4_rtsn	
P8_34	51	0x8cc/0cc	81	UART3_RTSN	gpio2[17]	uart3_rtsn	
P8_35	52	0x8d0/0d0	8	UART4_CTSN	gpio0[8]	uart4_ctsn	
P8_36	50	0x8c8/0c8	80	UART3_CTSN	gpio2[16]	uart3_ctsn	
P8_37	48	0x8c0/0c0	78	UART5_TXD	gpio2[14]	uart2_ctsn	
P8_38	49	0x8c4/0c4	79	UART5_RXD	gpio2[15]	uart2_rtsn	
P8_39	46	0x8b8/0b8	76	GPIO2_12	gpio2[12]	pr1_pru1_pru_r31_6	pr1_pru1_pru_r30_6
P8_40	47	0x8bc/0bc	77	GPIO2_13	gpio2[13]	pr1_pru1_pru_r31_7	pr1_pru1_pru_r30_7
P8_41	44	0x8b0/0b0	74	GPIO2_10	gpio2[10]	pr1_pru1_pru_r31_4	pr1_pru1_pru_r30_4
P8_42	45	0x8b4/0b4	75	GPIO2_11	gpio2[11]	pr1_pru1_pru_r31_5	pr1_pru1_pru_r30_5
P8_43	42	0x8a8/0a8	72	GPIO2_8	gpio2[8]	pr1_pru1_pru_r31_2	pr1_pru1_pru_r30_2
P8_44	43	0x8ac/0ac	73	GPIO2_9	gpio2[9]	pr1_pru1_pru_r31_3	pr1_pru1_pru_r30_3
P8_45	40	0x8a0/0a0	70	GPIO2_6	gpio2[6]	pr1_pru1_pru_r31_0	pr1_pru1_pru_r30_0
P8_46	41	0x8a4/0a4	71	GPIO2_7	gpio2[7]	pr1_pru1_pru_r31_1	pr1_pru1_pru_r30_1

P9 Header `pr1_pru1_pru_r31_11`



Run an example

- Finally you can run a simple example
- This is from EBB, chapter 13, converted to remoteproc

```
bone$ exercises/pru/examples/ebb
bone$ make
CC  main_pru0.c
CC  pru0-ledButton.asm
LD  gen/main_pru0.object gen/pru0-ledButton.object
-   Generated firmwares are : gen/main_pru0_fw.out
bone$ make install
-   copying firmware to /lib/firmware/am335x_pru0_fw
-   rebooting pru core 0
-   pru core 0 is now loaded with gen/main_pru0_fw.out
```

This is where the kernel looks for PRU code

dmesg

```
[Oct18 10:13] pru-rproc 4a334000.pru0: pru_rproc_remove: removing rproc 4a334000.pru0
[  +0.000063] pru-rproc 4a334000.pru0: stopping the manually booted PRU core
[  +0.000169] ti-pruss 4a300000.pruss: unconfigured system_events = 0xffffffffffffffff host_intr = 0x000000ff
[  +0.000037]  remoteproc2: stopped remote processor 4a334000.pru0
[  +0.000234]  remoteproc2: releasing 4a334000.pru0
[  +0.033808]  remoteproc2: 4a334000.pru0 is available
[  +0.000054]  remoteproc2: Note: remoteproc is still under development and considered experimental.
[  +0.000026]  remoteproc2: THE BINARY FORMAT IS NOT YET FINALIZED, and backward compatibility isn't yet guaranteed.
[  +0.000973] pru-rproc 4a334000.pru0: booting the PRU core manually
[  +0.008382]  remoteproc2: powering up 4a334000.pru0
[  +0.000519]  remoteproc2: Booting fw image am335x-pru0-fw, size 33476
[  +0.000124]  remoteproc2: remote processor 4a334000.pru0 is now up
[  +0.000082] pru-rproc 4a334000.pru0: PRU rproc node /ocp/pruss@4a300000/pru0@4a334000 probed successfully
```

Programming the PRU - c

- You can use either C or assembly, Let's do both

```
#include <stdint.h>
#include <pru_cfg.h>
#include <pru_ctrl.h>
#include "resource_table_pru1.h"

#define INS_PER_US 200            // 5ns per instruction
#define TIME 50 * 1000 * INS_PER_US    // set up a 50ms delay
extern void start(int time);
volatile register unsigned int __R30;
volatile register unsigned int __R31;

void main(void) {
 /* Clear SYSCFG[STANDBY_INIT] to enable OCP master port */
    CT_CFG.SYSCFG_bit.STANDBY_INIT = 0;
```

Programming the PRU - c

- You can use either C or assembly, Let's do both

```
void main(void){
…
    while(!(__R31&(1<<3))) {
        __R30 |= 1<<5;
        __delay_cycles(TIME);
        __R30 &= ~(1<<5);
        __delay_cycles(TIME);
    }
    __delay_cycles(TIME);  // Give some time for press to release
    // Call assembly language
    start(TIME);
    __halt();
}
```

P9_27	105	0x9a4/1a4	115	GPIO3_19	gpio3[19]	pr1_pru0_pru_r31_5	pr1_pru0_pru_r30_5
P9_28	103	0x99c/19c	113	SPI1_CS0	gpio3[17]	pr1_pru0_pru_r31_3	pr1_pru0_pru_r30_3

Output of C compiler

```
;-----------------------------------------------------------------------
;  59 | __R30 |= 1<<5;
;-----------------------------------------------------------------------
        SET     r30, r30, 0x00000005  ; [ALU_PRU] |59|
;-----------------------------------------------------------------------
;  60 | __delay_cycles(TIME);
;-----------------------------------------------------------------------
        .newblock
        LDI32   r1, 2499999
$1:     SUB     r1, r1, 1
        QBNE    $1, r1, 0           ; [ALU_PRU] |60|
```

```
;-----------------------------------------------------------------------
;  61 | __R30 &= ~(1<<5);
;-----------------------------------------------------------------------
        CLR     r30, r30, 0x00000005  ; [ALU_PRU] |61|
;-----------------------------------------------------------------------
;  62 | __delay_cycles(TIME);
;-----------------------------------------------------------------------
        .newblock
        LDI32   r0, 2499999
$1:     SUB     r0, r0, 1
        QBNE    $1, r0, 0           ; [ALU_PRU] |62|
;* ---------------------------------------------------------------------*
;*   BEGIN LOOP |$C$L2|
;* ---------------------------------------------------------------------*
|$C$L2|:
QBBC    |$C$L1|, r31, 0x03  ; [ALU_PRU] |58|
```

Programming the PRU - assembly

```
; Passed the number of cycles to delay in R14
; Register conventions are in "PRU Optimizing C/C++ Compiler v2.1 User's Guide"
; http://www.ti.com/lit/ug/spruhv7a/spruhv7a.pdf
; Section 6.3, Page 105
    .clink
    .global start
start:
    ; lsr       r14, r14, 1  ; Divide number of delay cycles by 2 since each
loop takes 2 cycles
    set     r30, r30.t5     ; turn on the output pin (LED on)
    mov     r0, r14         ; store the length of the delay in REG0

delayon:
    sub     r0, r0, 1       ; Decrement REG0 by 1
    qbne delayon, r0, 0   ; Loop to DELAYON, unless REG0=0
```

Destination

Programming the PRU - assembly

```
ledoff:
    clr     r30, r30.t5     ; clear the output bin (LED off)
    mov     r0, r14         ; Reset REG0 to the length of the delay

delayoff:
    sub     r0, r0, 1        ; decrement REG0 by 1
    qbne    delayoff, r0, 0  ; Loop to DELAYOFF, unless REG0=0

    qbbc    start, r31, 3    ; is the button pressed? If not, loop

end:
    jmp     r3              ; r3 contains the return address
                            ; Return value is in r14
```

Programming the PRU - c

- You can use either C or assembly, Let's do both

```
void main(void){
…
    while(!(__R31&(1<<3))) {
        __R30 |= 1<<5;
        __delay_cycles(TIME);
        __R30 &= ~(1<<5);
        __delay_cycles(TIME);
    }
    __delay_cycles(TIME);  // Give some time for press to release
    // Call assembly language
    start(TIME);
    __halt();
}
```

P9_27	105	0x9a4/1a4	115	GPIO3_19	gpio3[19]	pr1_pru0_pru_r31_5	pr1_pru0_pru_r30_5
P9_28	103	0x99c/19c	113	SPI1_CS0	gpio3[17]	pr1_pru0_pru_r31_3	pr1_pru0_pru_r30_3