What's Old Is New: A 6502-based Remote Processor

Matt Porter

Texas Instruments

6 November 2012

Texas Instruments

Overview

- The Big Picture(tm)
- 6502? Isn't that from the 70's?
- Why BeagleBone?
- PRU time!
- What is RemoteProc?
- Hardware Design
- Kernel Design
- Userspace Workflow
- Lessons Learned
- The Demo Scene
- Q&A

The Big Picture(tm)

- There's been some wishful thinking that AM335x had a general purpose M3 core to be used as a RemoteProc for offloading tasks
- AM335x did get the PRU...but it's not a well-known core with mature tools.
- At this time, my hobby projects were around some retro computing tinkering involving the MOS 6502.
- The vision then became, “I want a 6502 I can treat as a standard remote processor to Linux, develop and deploy 6502 applications from userspace, and communicate at runtime between Linux and the 6502 processor.”
- Why? Because we can.

You're Living in the Past!

- MOStech's 6502 MPU is one of the two most popular MPUs from the golden age of hobbyist and home computers.
  - Given hackers of a certain “maturity”, most can trace their early computing time to either 6502 or Z80
- Given that my first two computers were a Commodore VIC-20 and Apple][, the 6502 is pretty special to me and many others out there.
- The good news is we aren't alone. The retro computing scene is absolutely huge. With active software/hardware projects being born on a daily basis.
  - It's also very tied with FOSS principles and there's a lot of quality software to leverage.
- The retro applications for a 6502 remoteproc are huge as there is a lot of original games and applications written for vintage machines available to try out on a system of our own design.

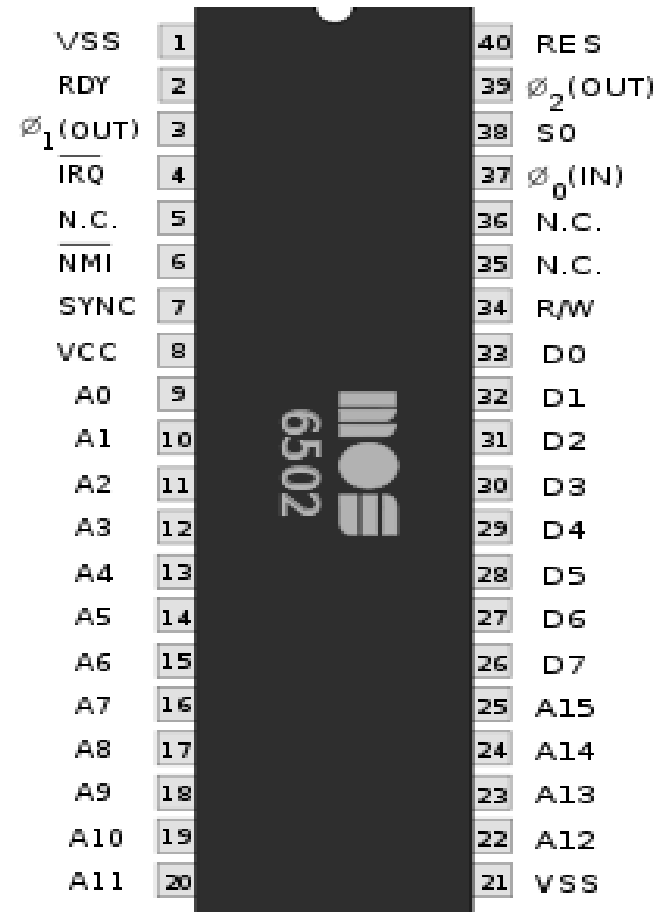
Texas Instruments

Back to the 21st Century

- The 6502 lives on in current production parts at Western Design Center
  - http://www.westerndesigncenter.com/wdc/
  - Seems to be a niche outside of retro computing for productions systems that need something with a full address/data bus rather than just a standard microcontroller.
- Current production parts include:
  - WDC65C02S in DIP and PLCC packages running up to 14MHz
  - WDC65C816S in DIP and PLCC packages running up to 14MHz
  - ASIC/FPGA implementations running up to 200MHz
- DIP40 version of the WDC65C02S is ideal for this design
  - Easy for a breadboard prototype
  - 3.3V/5.0V tolerant I/O

6502-Style

- 16-bit addressing, stored little endian
- 8-bit data bus
- Original clock speed was 1MHz at 5V
  - WDC part operates from DC->14MHz and 1.8V-5V
- Instructions vary in length and execution cycles
  - 69 instructions and 16 addressing modes
  - 6 registers (accumulator, x and y index, status, stack pointer, and program counter)
- NMI, IRQ, and Reset vectors at $FFFA, $FFFC, and $FFFE

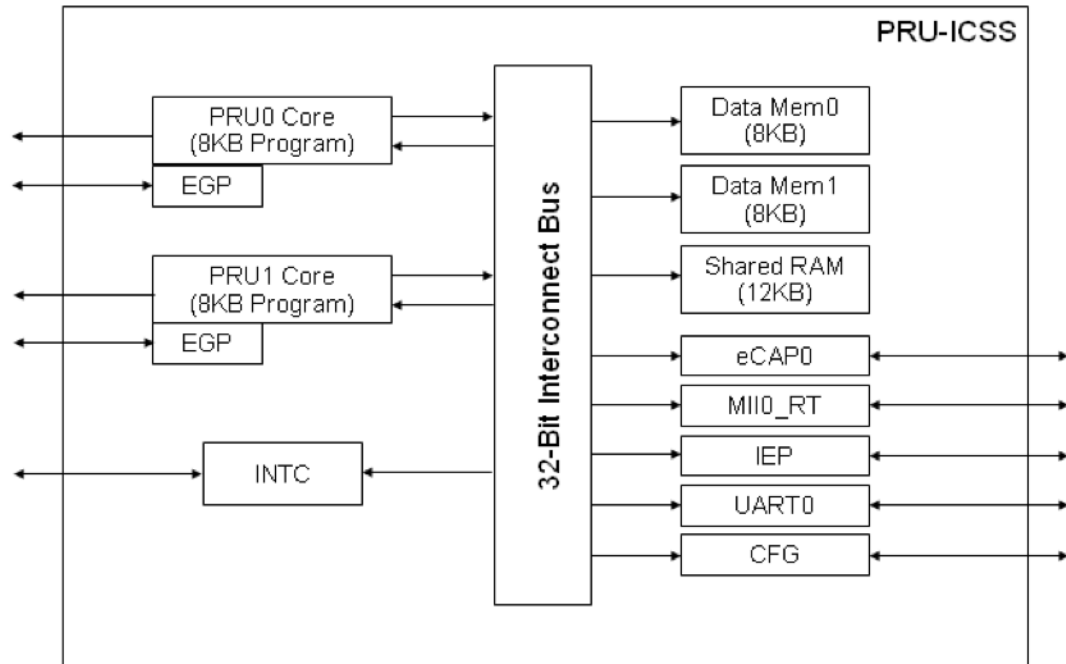


Texas Instruments

Can BeagleBone Help Here?

- Yes, some other fruity boards just don't have the peripherals we need to make this project a reality.
- BeagleBone fits well here because it has:
  - Oh so many expansion pins on those connectors
  - PWMs
  - PRUSS
  - GPIO (but then everybody has that)
  - Documentation with exact steps
  - “Pretty good nearly almost close to mainline kernel support”
    - https://github.com/beagleboard/kernel/tree/beaglebone-3.6
    - https://github.com/beagleboard/kernel/tree/3.7

PRUSS - Overview



- PRU Subsystem has two PRU cores. Each has 8KiB instruction RAM and 8KiB data RAM.
- Each PRU runs at 200MHz and instructions execute in 1 cycle (without stalls)
  - Stalls may occur when accessing SoC resources outside of PRUSS

Texas Instruments

PRUSS – The Ultimate Bitbanger

- 32 32-bit registers
- R30 register directly writes up to 16 PRU GPOs
- R31 register directly reads from up to 17 PRU GPIs
  - GPOs/GPIs are **not** the same as the GPIO hwmod
  - PRU GPOs/GPIs can be toggled with each 5ns cycle
- uio_pruss kernel driver facilitates loading of PRU code/data to either PRU core
- Userspace tools at https://github.com/beagleboard/am335x_pru_package
- Additional info at http://elinux.org/Ti_AM33XX_PRUSSv2

RemoteProc

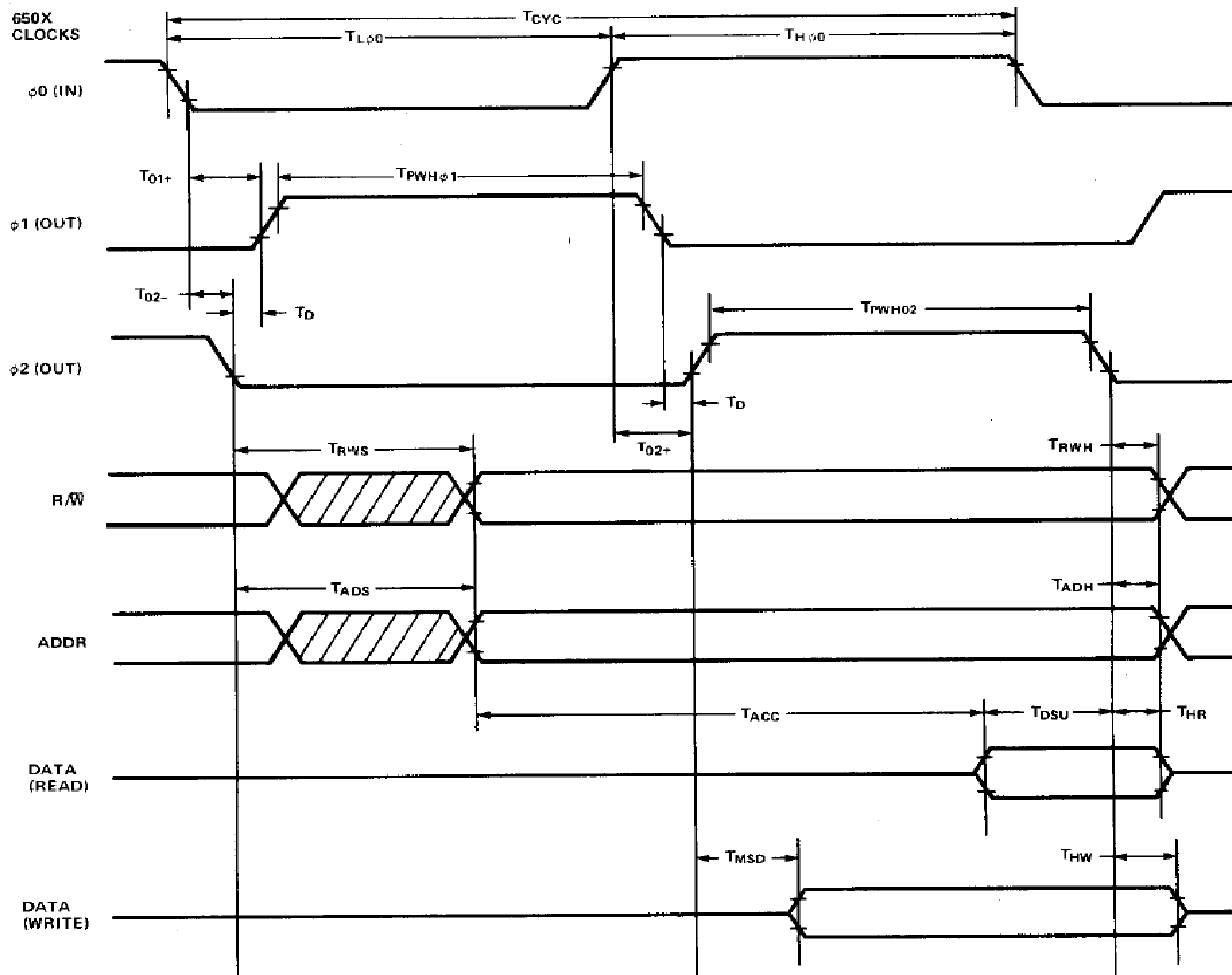
- Standard Linux kernel framework for managing remote processors
  - First implemented for OMAP M3 support
- RemoteProc features
  - Boot/shutdown processor
  - Load firmware image from /lib/firmware/
  - Supports creation of virtio devices from a firmware-resident resource table
  - Leverages existing virtio framework and drivers
    - Implementing RemoteProc allows use of the standard virtio drivers on Linux with our 6502 implementing the device.
- Why do we care?
  - No need to reinvent the wheel
- As is often the case though, RemoteProc/Virtio are missing a couple things that we need for our use case.

SYNERGY

Hardware Design

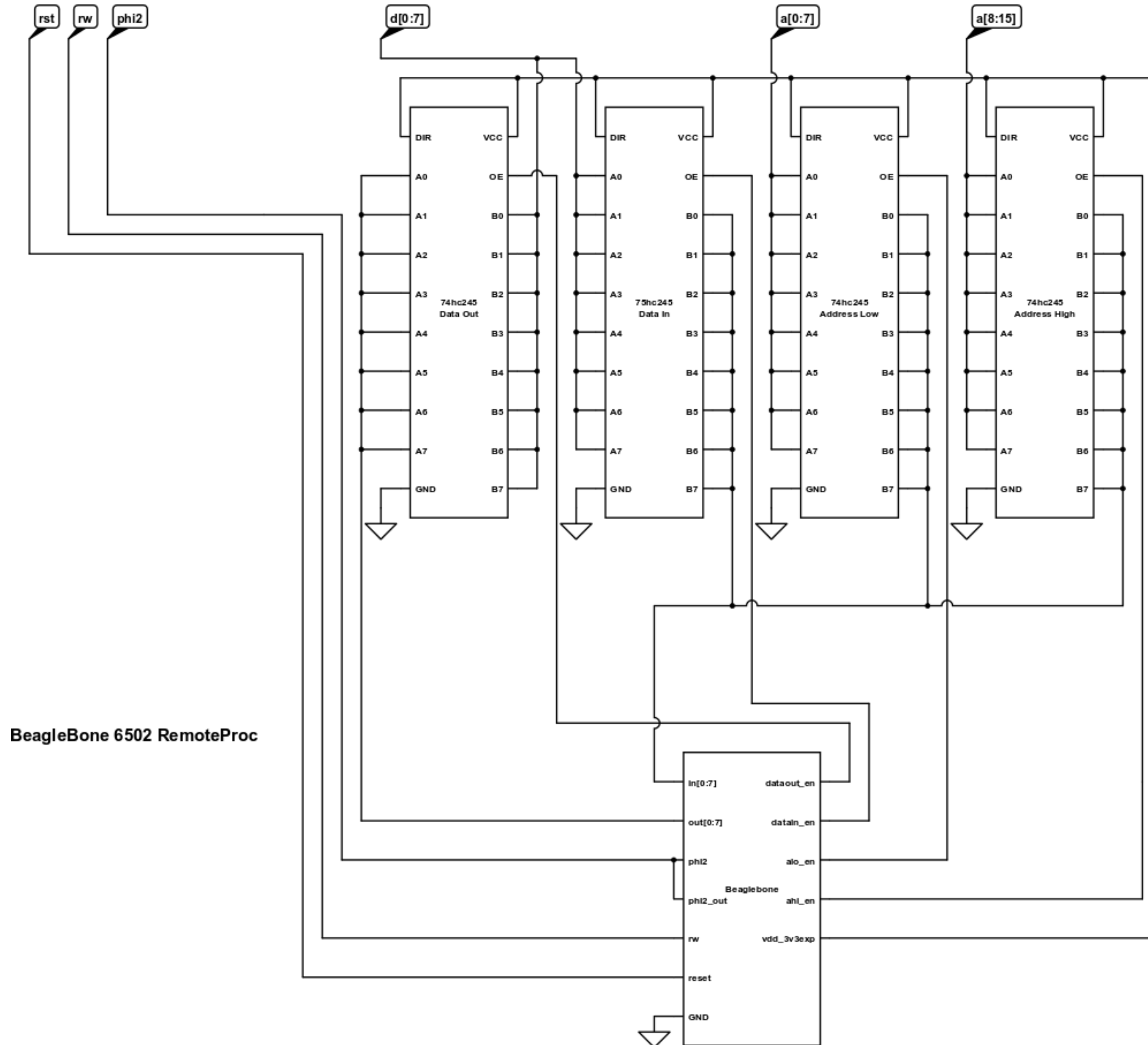
- GPIO for 6502 reset
- PWM for 6502 PHI2 clock input
- Interface 6502 address/data bus and control signals with PRUSS
  - Tricky! Inspired by lybrown's Atari Beaglebone expansion interface with the PRU. https://github.com/lybrown/abx
  - Very starved for PRU pins so address and data input needs to be muxed on to the same 8-bits of GPIs by controlling several 74HC245 octal bus transceivers
  - 75HC245 output enables for AHI, ALO, DATAIN, and DATAOUT controled by PRUSS GPOs.
  - Make use of GPI/GPO pins on both PRU0/1 to have enough

Hardware Design

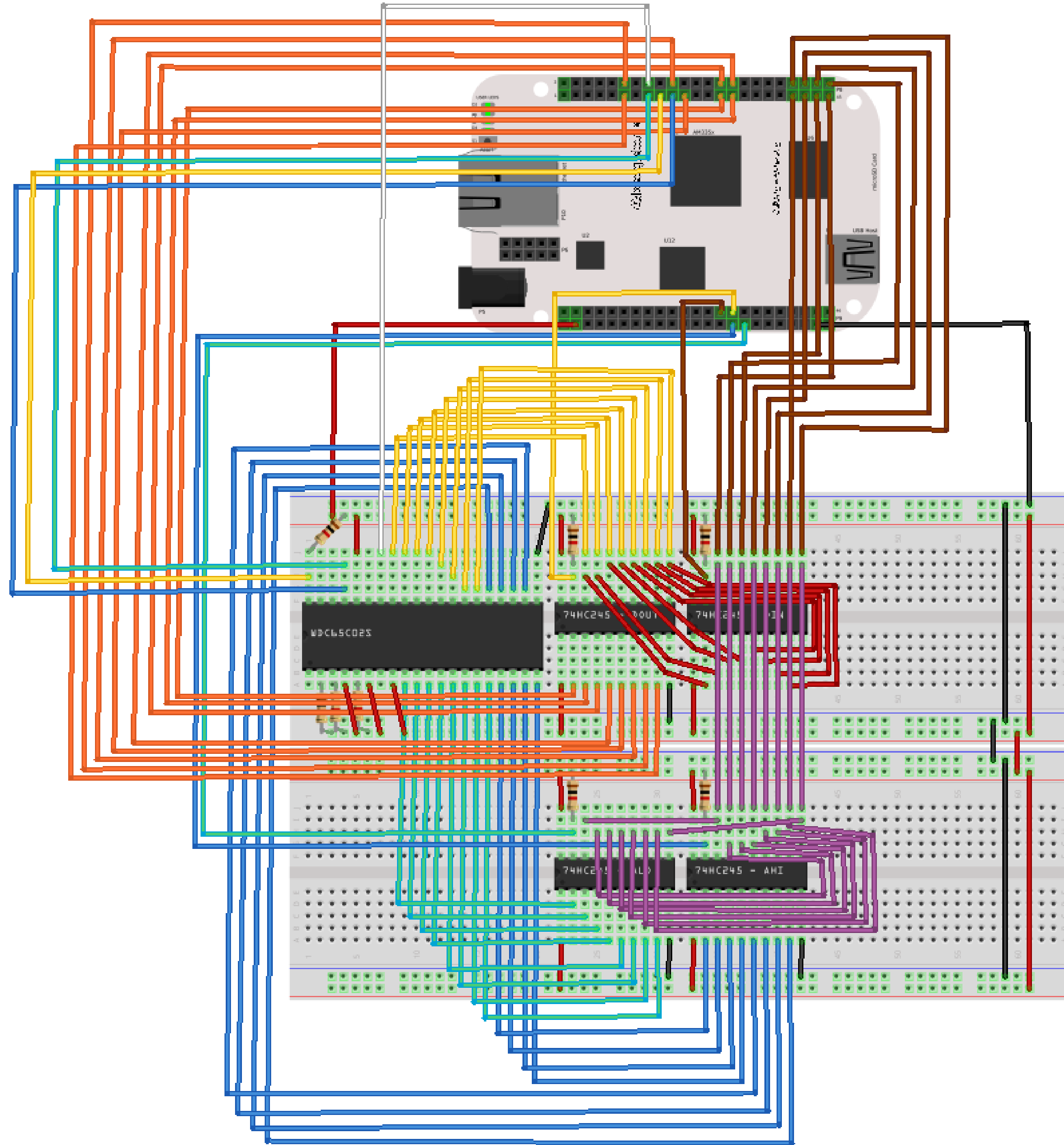


Texas Instruments

High Level Schematic



Texas Instruments



JMENTS

Pin Selection – What mux I do?

- PHI2 output
  - gpmc_ad8.ehrpwm2a
- Reset output
  - gpmc_ad11.gpio0_27
- Data Out 0..7
  - lcd_vsync.pr1_pru1_pru_r30_8
  - lcd_hsync.pr1_pru1_pru_r30_9
  - lcd_pclk.pr1_pru1_pru_r30_10
  - lcd_ac_bias_en.pr1_pru1_pru_r30_11
  - gpmc_csn1.pr1_pru1_pru_r30_12
  - gpmc_csn2.pr1_pru1_pru_r30_13
  - gpmc_ad12.pr1_pru0_pru_r30_14
  - gpmc_ad13.pr1_pru0_pru_r30_15
  - Note: bits 14 and 15 are on pru0 and the rest on pru1. This is due to GPO pin starvation and can be managed by shadowing writes to both PRU r30 registers.

Texas Instruments

Pin Selection – What mux I do?

- Data In 0..7
  - lcd_data0.pr1_pru1_pru_r31_0
  - lcd_data1.pr1_pru1_pru_r31_1
  - lcd_data2.pr1_pru1_pru_r31_2
  - lcd_data3.pr1_pru1_pru_r31_3
  - lcd_data4.pr1_pru1_pru_r31_4
  - lcd_data5.pr1_pru1_pru_r31_5
  - lcd_data6.pr1_pru1_pru_r31_6
  - lcd_data7.pr1_pru1_pru_r31_7
- RW In
  - gpmc_ad14.pr1_pru0_pru_r31_14
- PHI2 In
  - gpmc_ad15.pr1_pru0_pru_r31_15

Pin Selection – What mux I do?

- Address low enable
  - mcasp0_aclkx.pr1_pru0_pru_r30_0
- Address high enable
  - mcasp0_fsx.pr1_pru0_pru_r30_1
- Data out enable
  - mcasp0_axr0.pr1_pru0_pru_r30_2
- Data in enable
  - mcasp0_ahclkr.pr1_pru0_pru_r30_3

PRUSS Bitbanged RAM emulation for 65xx

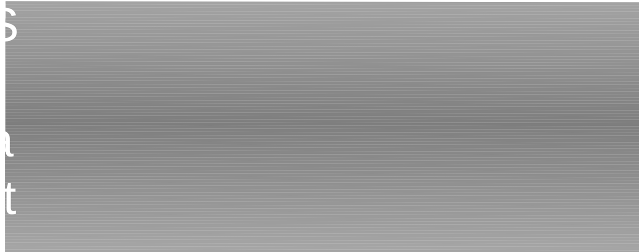
```
mem:
        wbc r31, PHI2_BIT // PHI2 falling edge
        ldi r30, AHI_EN
        wbs r31, PHI2_BIT // PHI2 rising edge
        lbbo r1.b1, pru1_r31, 0, 1 // AHI
        ldi r30, ALO_EN

        and r1.b1, r1.b1, 0x1f
        qbbs read, r31, RW_BIT // RW
write:
       EN_DELAY // 5 NOPS
       lbbo r1.b0, pru1_r31, 0, 1 // ALO
       ldi r30, DATAIN_EN
       EN_DELAY // 5 NOPS
       lbbo r3, pru1_r31, 0, 4 // DATAIN
       sbbo r3, membase, r1.w0, 1
       jmp mem

read:
       EN_DELAY // 5 NOPS

       lbbo r1.b0, pru1_r31, 0, 1 // ALO
       ldi r30, DATAOUT_EN
       lbbo r3.b1, 0x10000, r1.w0, 1
       sbbo r3, pru1_r30, 0, 2 // DATAOUT
       mov r30.b1, r3.b1 // DATAOUT
       jmp mem
```

RemoteProc Driver

- Implemented remoteproc driver as documented in http://www.mjmwired.net/kernel/Documentation/remoteproc.txt
- Uses
  - 8KiB of shared PRUSS SRAM
  - GPIO to generate 6502 reset
  - PWM to generate 6502 clock
- Currently does not support virtio resources as virtio requires buffers allocated via the DMA API with sizes that exceed that of the SRAM available as well as the total address space of the 6502.
- Uses trivial custom firmware header similar to the STE Modem remoteproc driver as opposed to ELF firmware
- Added a basic userspace sysfs interface to rproc_start()/stop() to allow development of 6502 applications and load/execution completely from userspace.

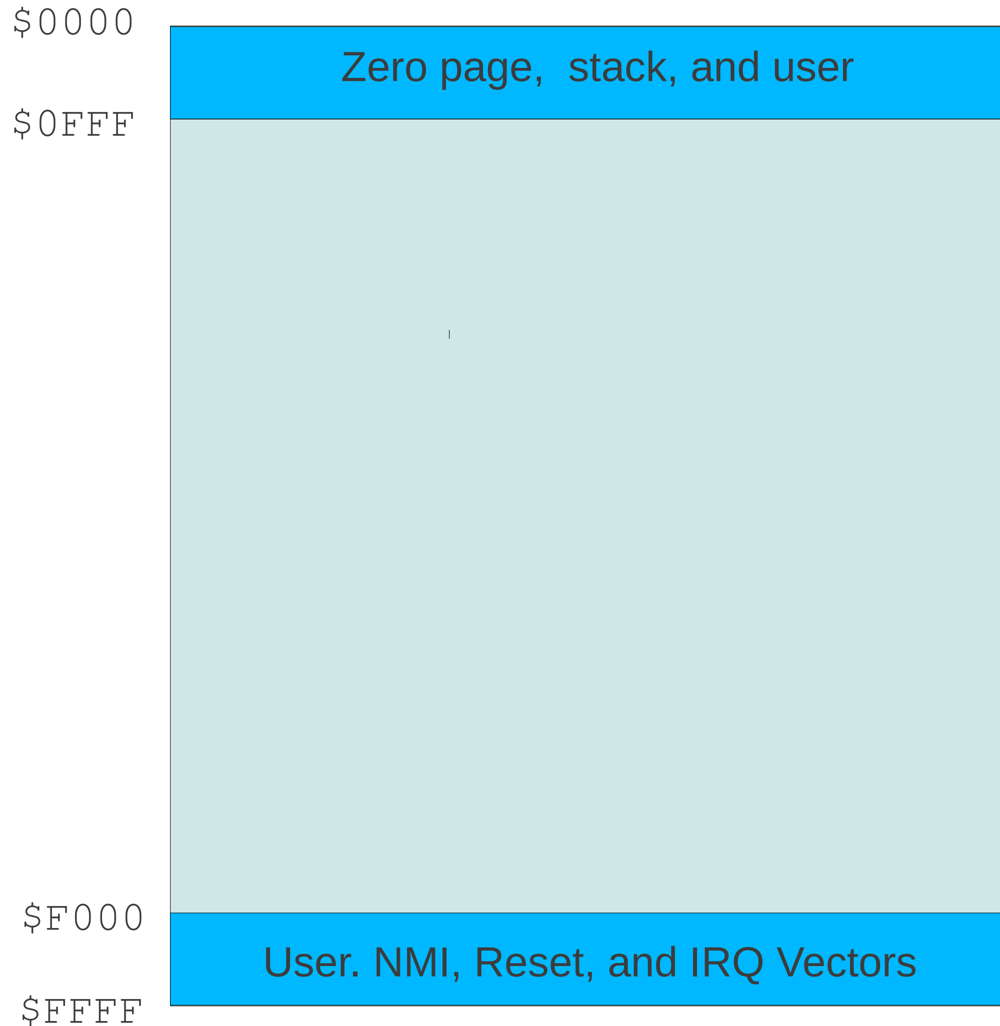
RemoteProc Driver

```
static int b6502_rproc_start(struct rproc *rproc)
{
        struct b6502_rproc *bproc = rproc->priv;
        int ret;

        /* Start pwm and deassert reset */
        ret = pwm_config(bproc->pwm, B6502_CLK_PER/2, B6502_CLK_PER);
        if (ret < 0)
                return ret;
        pwm_set_polarity(bproc->pwm, PWM_POLARITY_NORMAL);
        ret = pwm_enable(bproc->pwm);
        if (ret < 0)
                return ret;

        gpio_set_value(bproc->reset, 1);

        return ret;
}

static int b6502_rproc_stop(struct rproc *rproc)
{
        struct b6502_rproc *bproc = rproc->priv;

        /* Assert reset and stop pwm */
        gpio_set_value(bproc->reset, 0);
        pwm_disable(bproc->pwm);

        return 0;
}
```



Texas Instruments

6502 Memory Map



A15..A13 not decoded. 8K of SRAM is aliased throughout the 64K address space

Texas Instruments

Can DT help here?

```
b6502_pins: pinmux_b6502_pins {
        pinctrl-single,pins = <
                /* clock and reset */
                0x20 0x04       /* gpmc_ad8.ehrpwm2a, MODE4 */
                0x2c 0x17       /* gpmc_ad11.gpio0_27, OUTPUT_PULLUP | MODE7 */

                /* data out 0..7 */
                0xe0 0x15       /* lcd_vsync.pr1_pru1_pru_r30_8, OUTPUT_PULLUP | MODE5 */
                …..
       >
};
b6502rproc: rproc@0 {
        compatible = "bone6502";
        pinctrl-names = "default";
        pinctrl-0 = <&b6502_pins>;
        sram = <&sram6502>;
        reset = <&gpio0 27 0>;
        pwms = <&ehrpwm2 0 1000 0>;
        pwm-names = "clock";
};
```

Texas Instruments

Userspace Workflow

- The following tools are used:
  - b6502_pruss PRUSS loader and 6502 memory decoder firmware
  - cc65 toolchain from http://www.cc65.org
  - b6502 assembly examples
- With the Angstrom distribution (or the ARMv7 distro of your choice) and a native development environment installed, the cc65 toolchain can be built and installed.
- The b6502 assembly examples require only the ca65/ld65 assembler and linker to be in the path.

Running the Bone-6502 Hello World

```
root@bone2:~/b6502/pokechar# make
ca65 -l pokechar.lst pokechar.s
ld65 -o pokechar.bin -C b6502.cfg pokechar.o
root@bone2:~/b6502/pokechar# cp pokechar.bin /lib/firmware/bone-6502-fw.bin
```

```
root@bone2:~/b6502/pokechar# b6502_pruss
root@bone2:~/b6502/pokechar# echo 1 > /sys/class/rproc/remoteproc0/on_off
```

```
root@bone2:~/b6502/pokechar# devmem2 0x4a31fff0 w
/dev/mem opened.
Memory mapped at address 0xb6f92000.
Read at address  0x4A311FF0 (0xb6f92ff0): 0x00000043
```

Texas Instruments

Examining Hello World

```
.include "b6502_hdr.inc"

.segment  "SRAM1"     ; place our code in the SRAM1 segment (0xf000)

start:
        lda #$43     ; load 'C' to accumulator
        sta $fff0    ; store accumulator to magic address

.segment "VECTORS" ; place the required vectors

.word start          ; NMI
.word start          ; RST
.word start          ; IRQ
```

Hello World Linker Script

```
# B6502 Linker script
#
# Memory Map:
# $0000-$0fff     RAM0     4KiB
# $f000-$ffff     RAM1     4KiB
#
# B6502 remoteproc compliant header is generated
# and prepended to make a ready to load image containing
# 8KiB total of memory contents for SRAM.

MEMORY {
        HEADER:  start = $0000, size = $0028, file = %O;
        RAM0:    start = %S, size = $1000, fill = yes;
        RAM1:    start = $f000, size = $1000, fill = yes;
}

SEGMENTS {
        B6502HDR:    load = HEADER, type = ro;
        SRAM0:   load = RAM0, type = rw, optional = yes;
        SRAM1:   load = RAM1, type = rw, optional = yes;
        VECTORS:load = RAM1, type = ro, optional = yes, offset = $0ffa;
}

FILES {
        %O: format = bin;
}
```

B6502 RemoteProc Image Header

```
; B6502 compliant remoteproc header
;
; Header is prepended to the 8KiB SRAM image contents
;
.segment "B6502HDR"
.word $beef     ; magic
.word $dead
.word $0028     ; image offset
.word $0000
.word $2000     ; image size
.word $0000
.word $0014     ; resource table offset
.word $0000
.word $0014     ; resource table size
.word $0000
.word $0001     ; resource table - version
.word $0000
.word $0000     ; resource table - num entries
.word $0000
.word $0000     ; resource table - reserved 0
.word $0000
.word $0000     ; resource table - reserved 1
.word $0000
.word $0000     ; resource table - offset 0
.word $0000
```

Texas Instruments

BVUART – Talking to the 6502

- As mentioned, making use of the Virtio framework and virtio-console driver is not possible without some changes to adapt it to working with very small buffers that are not in system memory.
- The workaround is the creation of a simple kernel serial driver, bvuart.c, which is based on Greg KH's tinyserial example.
  - The driver is kernel timer polling driven
  - It relies on the 6502 side to also implement the same communication scheme
  - Using 4 bytes within the 8K SRAM space, a set of TX, RX, TXSTS, and RXSTS registers are created. Each side of the comm channel marks their respective RX channel as ready on start. The other side marks that channel as valid after placing a byte of data in. The TX side only places a byte in the TX register if the status shows ready.
- With this driver, /dev/bvuart can be opened from userspace to provide a direct serial console to the 6502 processor.

BVUART – Echo Example

```
.include "b6502_hdr.inc"

.segment  "SRAM1"       ; place our code in the SRAM1 segment (0xf000)

BVUART          = $fff0
BVUART_RX       = VUART
BVUART_RXSTS    = VUART + 1
BVUART_TX       = VUART + 2
BVUART_TXSTS    = VUART + 3
BVUART_STS_RDY = $40
BVUART_STS_VLD = $20
```

```
start:
        lda #BVUART_STS_RDY
        sta VUART_RXSTS ; we're ready for a character
        lda #BVUART_STS_VLD
rxtst:  cmp VUART_RXSTS ; character available?
        bne rxtst
        ldx VUART_RX    ; load it in X

        lda #BVUART_STS_RDY
txtst:  cmp VUART_TXSTS ; ready for tx?
        bne txtst
        txa             ; fetch character to echo
        sta VUART_TX    ; store accumulator to vuart tx register
        lda #BVUART_STS_VLD
        sta VUART_TXSTS ; tx content is valid

        jmp start       ; do it again

.segment "VECTORS"      ; place the required vectors

.word start             ; NMI
.word start             ; RST
.word start             ; IRQ
```

Texas Instruments

Lessons Learned

- DT can help...if you have infinite time to implement everything that you need.
  - As long as all our drivers we leverage are DT-enabled, it does make it clear and concise as to what we are instantiating and what resources it uses.
  - Instead of leaving the implicit message unwritten, it should be pointed out that a huge amount of effort went into getting uio_pruss, in particular, DT-enabled and working with the current hwmod external reset scheme. It's still not upstream ready but it works for me. There was a lot of work by many different people just to have all the AM335x drivers needed here to be DT-enabled and working against 3.6+.
  - Things are still immature with doing production work on a DT-only kernel as many drivers don't have DT bindings...YMMV.
- When in doubt, your pinmux is wrong
  - Always always always, quadruple check it, maybe even a fifth time.

Lessons Learned

- When you are both the hardware developer and the software developer, you have nobody to blame but yourself.
  - Don't forget the pullup resistors on the 74HC245 output enable pins that are asserted low...especially when the pins left floating double as SYSBOOT pins at power up!
  - Remember to read the processor manual more times than you would expect before it really gets through to you that the clock has to be active 3 cycles before reset goes high.
  - Large breadboards should be banned by treaty worldwide. Debugging large address/data bus muxed nest of wires breadboard prototypes requires a particular level of patience.
  - Reread your datasheet for all part timings. Don't ever miss the fact that a propagation delay and output enable activation time can be quite different at 3.3V than at 5V, be sure to account for that in your PRUSS 65xx bitbanging code as you really want the delays correct so the address/data lines are actually valid when you read them on the 74HC245 outputs.

Lessons Learned

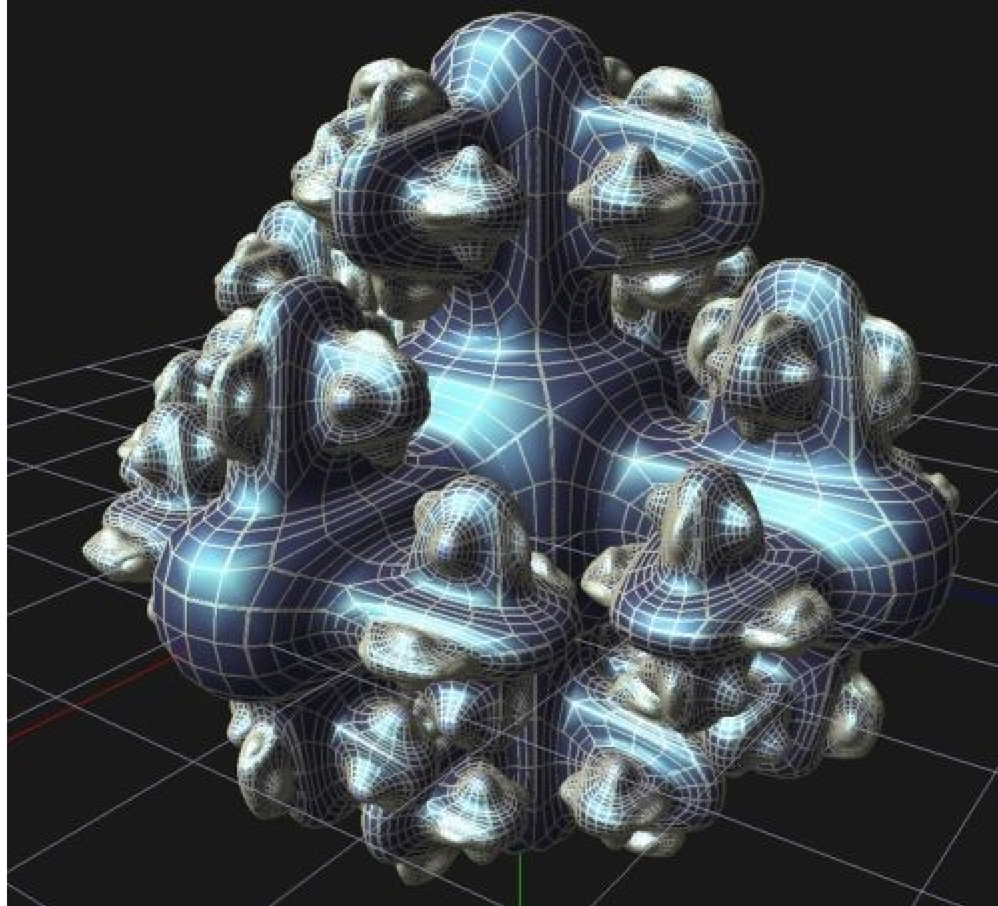
- More of an affirmation of what I tell people on a daily basis.
  - **More than likely, everything you're about to do has been done before**.
  - I leveraged the abx PRU project to save time figuring out a PRU pin scheme, simplified the code
  - Remoteproc already has everything figured out for remote processor management , messaging, and task offload
    - The sysfs interface I needed was a trivial improvement
  - Retro community had a finished product toolchain that even was robust enough on the linker side to produce the firmware header needed so no need to even write a custom .bin to b6502 remoteproc image creation tool.
  - In summary, there's just not much new code here to make this all work.
- Open Bench Logic Sniffer continues to pay for itself 100x over the 50USD price after a year of using it
  - #essentialtools for this project

Lessons Learned

- PRUSS is amazingly useful
  - After having mostly done blinkenlights tests and playing with PRUSS on AM180x and more recently on BeagleBone I hadn't considered what it could really do until I saw the abx project.
  - Initially, the design was to use a dual-port SRAM interfaced via GPMC. This was already prototyped using a less expensive single port SRAM when I was pointed at the abx project which changed my thinking. No need for external SRAM. We already paid for it inside the AM335x! Nice cost savings.
  - There's a lot that can be done with emulating different devices on the PRU, one drawback is the software story in that uio_pruss is not the ideal model for all PRU use cases. In fact, having the PRU supported as a remoteproc itself would be the ideal for many use cases. Certainly for those where the PRU implements a hardware device or system feature as that is precisely the model that remoteproc set out to solve.
    - But it's just software, we can fix that in time.

Demo Scene

- ...or a screencast if my prototype broke in transit.



? || /* */

- Bone-6502 RemoteProc Kernel
  - https://github.com/ohporter/linux/tree/WIP/bone-6502-remoteproc
- Bone-6502 Tools
  - https://github.com/ohporter/b6502

Texas Instruments