MAKE | BUILD | HACK | CREATE

132 PAGES OF MAKING

HackSpace

TECHNOLOGY IN YOUR HANDS

hsmag.CC | March 2020 | Issue #28

Mar.2020 Issue #28 £6

9 772515 514006 28

Raspberry Pi PRESS

ELECTRONICS

BUILD BETTER CIRCUITS

Strong 3D prints

Finding the settings to make plastic last

Raspberry Pi Audio

Make your projects sing

TESTING COMPONENTS
TOOTHPICKS
CIRCUIT PYTHON
BELT DRIVES

PROJECTOR MASK

Foil facial recognition with 3D printing and a Raspberry Pi Zero

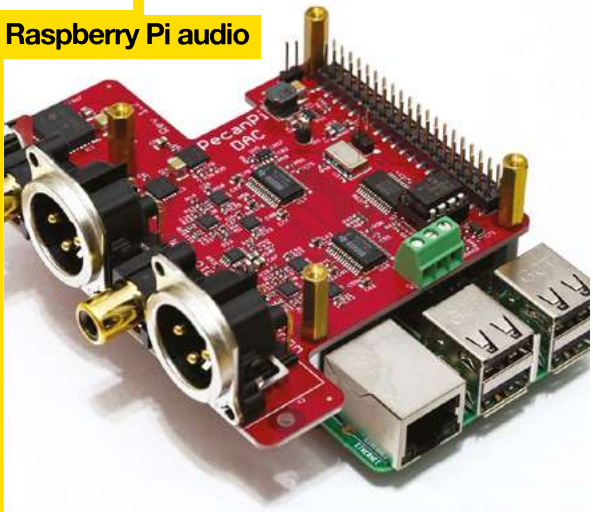
SLA PRINTING ARCADES STICK WELDING VASE

Contents

120

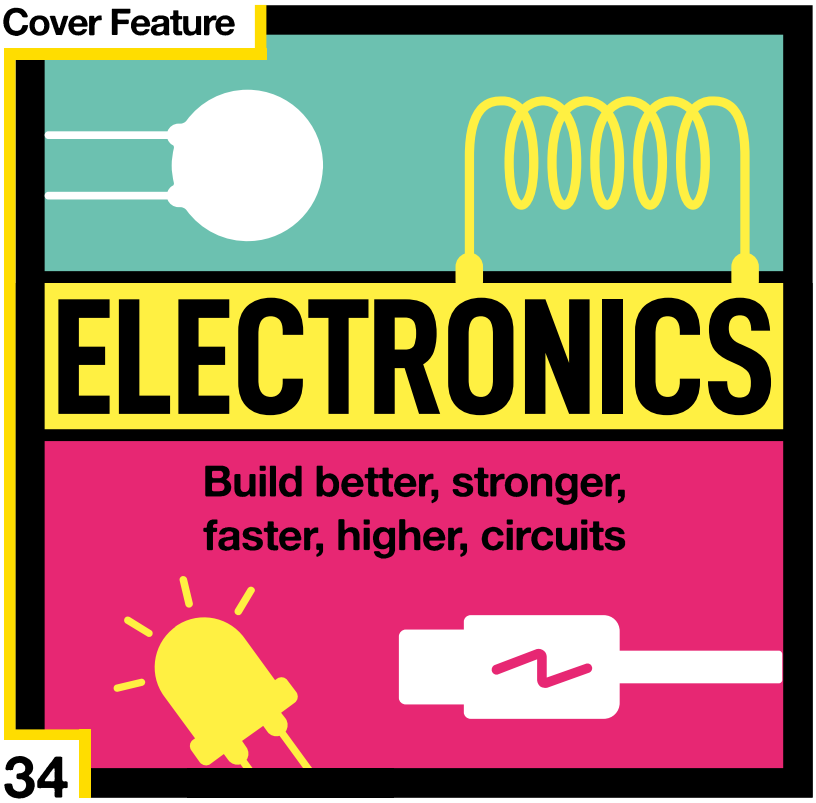
Tutorial

Raspberry Pi audio

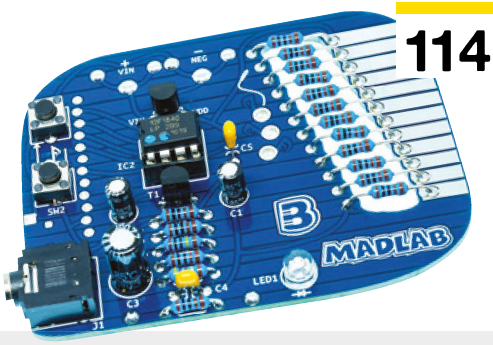


86 How to wrangle decent audio out of your Raspberry Pi

Cover Feature



34



114

TUTORIAL

Real-time Twitch gaming

Low-latency interfacing requires a kernel driver, right? Think again

Andrew Henderson

Dr. Andrew Henderson is an author, researcher, and software engineer. He has developed Linux embedded and desktop software for over 20 years. His website is **icculus.org/~hendersa**.

Figure 1 SNES gamepads have eight buttons and a directional-pad, making them a flexible input option for all sorts of projects (Credit: adafruit.com)

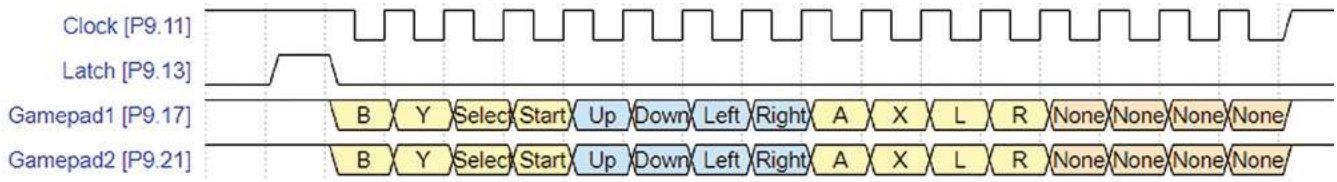
Linux is available for most ARM single-board computers, and the wide variety of open-source software and tools makes it easy to hack projects together. You don't need to worry about the low-level details of the system because the Linux kernel hides those away. You can concentrate on what you want to do, without worrying about every single detail of how you're going to do it.

It would be nice to get the performance and hardware access of a kernel driver without having to actually write one. You could treat your platform like a Linux system to do all of the hard stuff (video, audio, networking, memory management, etc.), and treat it like a microcontroller to do the bare-metal, time-sensitive GPIO interfacing. Is such a thing possible?

In this article, you'll see how to use the programmable real-time units (PRUs) of the BeagleBone Black (BBB) to read from and write to GPIOs directly. You'll use a PRU to natively interface the GPIOs to a pair of Super Nintendo Entertainment System (SNES) gamepads (**Figure 1**) for lightning-fast, low-latency input… all without using a kernel driver!

INS AND OUTS OF PRU INTERFACING

What exactly is a PRU? A PRU is an independent RISC coprocessor with its own memory and



registers. The TI AM3359 processor of the BBB contains a pair of PRUs. The PRU instruction set has arithmetic and logical operations (add, subtract, shifts), loads/stores for registers and memory, and control flow operations (compares, loops, jumps). Execution is very fast, taking only one 5 ns clock cycle to execute an instruction. PRU firmware size is limited to 8kB, but a program that toggles GPIOs will easily fit within only a few dozen bytes.

The BBB has 33 PRU-enhanced GPIOs available: 16 for output and 17 for input. These GPIOs can be accessed by a PRU in only a single PRU clock cycle! However, multiplexing enhanced GPIOs to the BBB's P8 and P9 headers often requires unmuxing other BBB features that you might want to use (like audio, video, and SPI channels). For applications where enhanced GPIOs create conflicts, the PRU can still interact with standard GPIOs. This is a bit slower, though, taking three to four PRU cycles per each read/write of a GPIO control register. For this project, you'll be using standard GPIOs for flexibility.

The Linux kernel uses two methods for communicating with PRUs: remote procedure calls (remoteproc, which is unrelated to POSIX RPC) and userspace I/O (UIO). Remoteproc provides communication between a custom Linux kernel driver and the PRU firmware. UIO skips the kernel driver and allows user space applications direct access to memory shared with the PRU.

Which PRU interface method should you use? This article uses UIO, which is very straightforward and simple. Remoteproc is far more powerful and efficient than UIO, but it requires that you write a kernel driver to use it. UIO is a good place to start for beginners. If you're doing complex projects in the future, consider using remoteproc.

BITS AND BYTES OF THE GAMEPAD PROTOCOL

SNES gamepad connectors have seven pins, though only five are used to interface with the gamepad. Two pins provide power (5 V and ground), and two input pins accept 'clock' and 'latch' control signals. The final pin is an output 'data' signal which provides a serial stream of data (1 bit per pulse of the clock signal) that represents the state of each gamepad button and the directional-pad (D-pad).

> "Useful data is only returned during **the first twelve of the clock signal pulses**"

Figure 2 shows the timing diagram for the serial communication protocol with two gamepads. Each time that you wish to sample the gamepad state, the latch signal is raised for 12 µs and then lowered to wake up the gamepad. After the latch pulse, a series of 16 pulses, each 12 µs in duration, are sent on the clock signal. During each clock pulse, the state of one gamepad button is returned on the data line as a bit of data. The bit for that button will be 0 (low) if the button is pressed and 1 (high) otherwise. There are only twelve bits (eight for buttons and four for the D-pad), so useful data is only returned during the first twelve of the clock signal pulses. During the final four clock pulses, a 1 is returned on the data signal. After these 16 pulses, the gamepad goes back to sleep until another latch signal is received.

Interfacing with gamepads using this protocol is an ideal PRU starter project because the protocol is →

Figure 2
For more details on the SNES gamepad protocol, visit hsmag.cc/JAQwz7

Below
Relive your youth with our nostalgic games controller

YOU'LL NEED

- **BeagleBone Black**
- **One or two Super Nintendo gamepads** (adafruit.com, Part #131)
- **One or two female Super Nintendo gamepad connectors** (raphnet-tech.com, Part #250025)
- **Four bidirectional BSS138 line-level converters** (adafruit.com, Part #757)
- **Breadboard and jumper wires**
- **Basic soldering equipment**

RECOMMENDED

- **4GB or larger microSD card**

QUICK TIP

For step-by-step instructions on updating your BBB, downloading and writing images to microSD cards, logging into your BBB, and configuring your BBB's network connection, visit **beagleboard.org/getting-started**

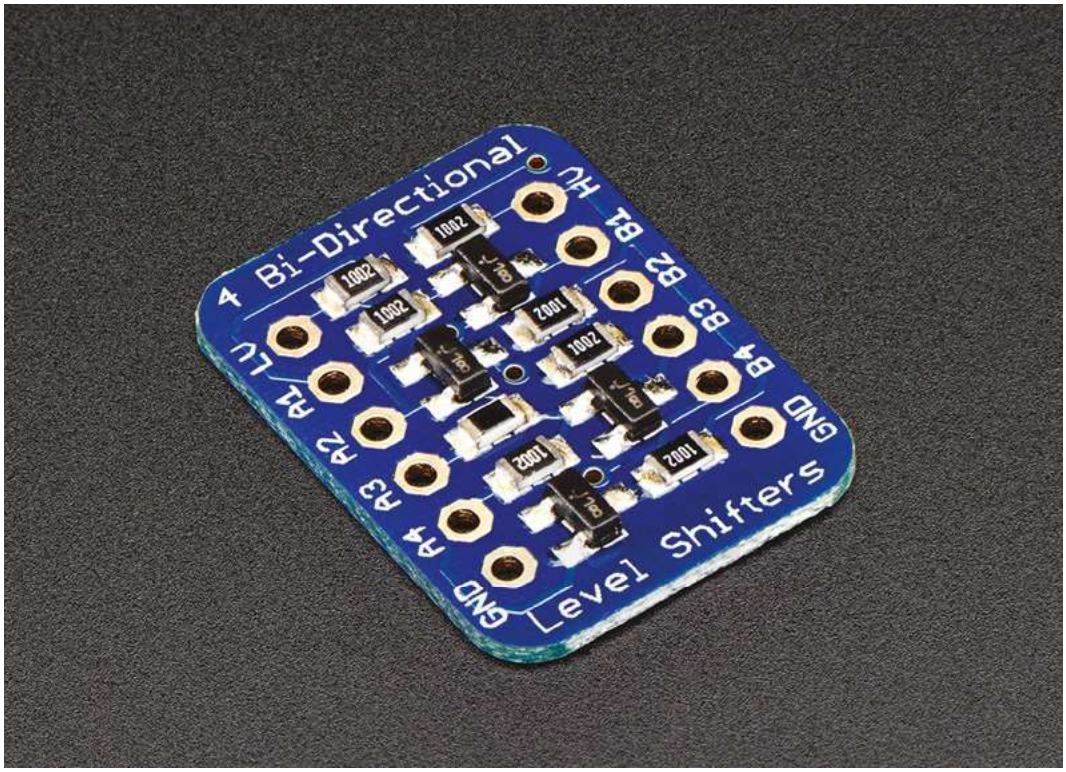


Figure 3 Double-check your breadboarding to ensure that your BBB's GPIO pins are only connected to the low-level side ('A'-side) of your logic-level converter PCB (Credit: adafruit.com)

simple, implemented solely using GPIOs, and has strict timing constraints (but not so strict that you are limited to using PRU-enhanced GPIOs).

PREPARING FOR PRUS

Working with PRUs requires a BBB system with PRU support in its kernel, and a toolchain for building PRU firmware. You should start experimenting with PRUs using a known-good BBB OS image with PRU support that runs from a microSD card. The recent Debian 9.9 Internet of Things (IoT) image available for the BBB provides a 4.14 Linux kernel and root file system that supports both the remoteproc and UIO methods of PRU interfacing. If you already have a BBB OS environment that you like, feel free to try it when following these instructions. However, troubleshooting the setup of the PRU can be difficult, so we recommend using the suggested IoT image to make things easier.

By default, the IoT image is configured to use the remoteproc PRU interface. Since you'll be using the UIO interface instead, you'll need to make a small change to the image. The U-Boot bootloader configuration script (**/boot/uEnv.txt**) on your microSD must be changed to enable UIO. The image comes with the Nano, Pico, and Vim text editors already installed, so you should be able to find one that you like to edit **uEnv.txt** on your running BBB system.

Log into your BBB system and change the `uboot_overlay_pru` setting of **uEnv.txt** to use the UIO device tree overlay, rather than the remoteproc one. To make the edit, simply insert a hash (#) at the start of line 38 to comment out the `pru_rproc` overlay option for (`4.14.x-ti kernel`) and remove the starting hash on line 42 to comment in the `pru_uio` overlay option. After the change, the file should look like this (starting at line 37):

```
###pru_rproc (4.14.x-ti kernel)
#uboot_overlay_pru=/lib/firmware/AM335X-PRU-RPROC-4-14-TI-00A0.dtbo
###pru_rproc (4.19.x-ti kernel)
#uboot_overlay_pru=/lib/firmware/AM335X-PRU-RPROC-4-19-TI-00A0.dtbo
###pru_uio (4.4.x-ti, 4.14.x-ti, 4.19.x-ti & mainline/bone kernel)
uboot_overlay_pru=/lib/firmware/AM335X-PRU-UIO-00A0.dtbo
```

Once **uEnv.txt** has been modified, save your changes, sync the file system, and reboot your BBB:

```
$ sync
$ sudo shutdown -r now
```

With the system rebooted, log into your BBB once more and ensure that the UIO PRU kernel module is properly loaded on your system. The easiest way to do this is to check for the presence of the **/dev/uio0** file, which is created when the UIO driver is loaded:

```
$ ls -l /dev/uio0
crw-rw---- 1 root users 246, 0 Sep  3 02:28 /dev/
uio0
```

If you do not see the **/dev/uio0** file, double-check your edit to the **uEnv.txt** file to ensure that the UIO overlay is commented in and that all RPROC overlays are commented out. With the UIO PRU driver loaded, you are now ready to begin experimenting with the PRU via the UIO interface.

The PRU development package from **beagleboard.org** contains the PRU assembler (PASM), which compiles PRU assembly code into binary PRU firmware images. Download and build the source code for PASM from GitHub onto your BBB:

```
$ cd ~
$ git clone https://github.com/beagleboard/
am335x_pru_package
$ cd am335x_pru_package/pru_sw/utils/pasm_source
$ ./linuxbuild
$ sudo cp ../pasm /usr/local/bin
```

Now that you have a working system containing the tools needed to begin working with the PRU, you can start interfacing the BBB with some hardware.

CREATING THE GAMEPAD INTERFACE CIRCUIT

Unfortunately, SNES gamepads use 5V logic-level control signals, and BBB GPIOs use 3.3V. You need line-level converters to translate between the two. Because this project has two output signals (latch and clock) and two input signals (the data line from each gamepad), it is simplest to use bidirectional converters when constructing your interface circuit. Adafruit sells a PCB with four BSS138 bidirectional converters mounted on it (**Figure 3**) which is perfect for this project. Adafruit's PCB has two sides: one is low voltage (the 'A'-side), and one is high voltage (the 'B'-side). Any low-voltage signal connected to pin A1 will be translated to the high-voltage signal connected to pin B1 and vice versa. A2 maps to B2, A3 to B3, and A4 to B4.

The SNES female connectors have seven thick pins that are designed to anchor the connector in place when soldered to a PCB. These pins are much wider than the diameter of the holes in a breadboard. Solder a breadboard wire to each pin (skipping the unused pins 5 and 6) to make it easy to wire each connector into the breadboard (**Figure 4**).

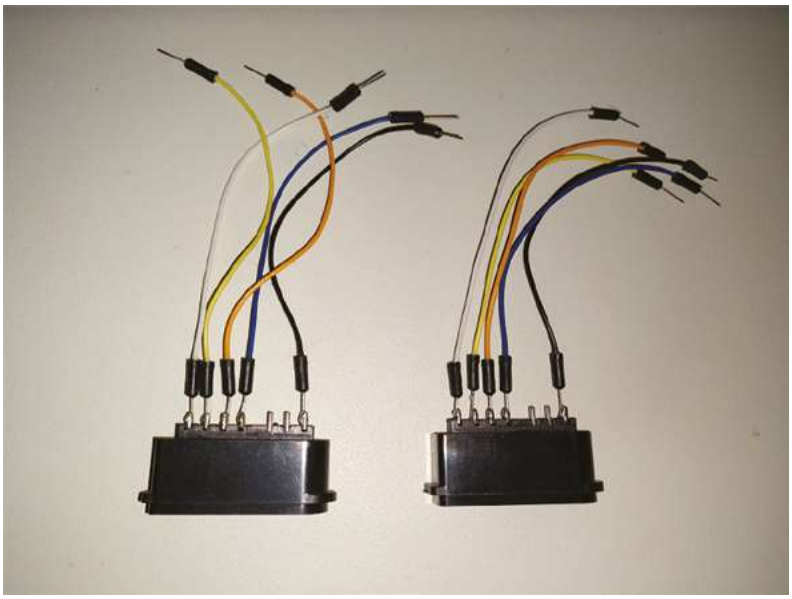


Figure 4 Adding the breadboarding wires makes it easy to reuse the connectors in multiple projects

This firmware uses four GPIO pins that are muxed to the BBB's P9 connector: latch is assigned to pin P9.11, clock to pin P9.13, gamepad one data to pin P9.17, and gamepad two data to pin P9.21. These four GPIOs were selected because they are muxed as GPIOs by default in BBB OS images. **Figure 5** (overleaf) shows the completed interfacing circuit.

> "It is simplest to use bidirectional converters **when constructing your interface circuit**"

Wiring the circuit is easy when using a breadboard. Both the gamepads and the line-level →

THE SIGNALS ON THE PINS OF A SNES CONNECTOR

Pin 1: +5V	Pin 2: Latch
Pin 3: Clock	Pin 4: Data
Pin 5: Unused	Pin 6: Unused
Pin 7: GND	

QUICK TIP

For more information on the PRU instruction set, and many other details of PRU interfacing, visit **hsmag.cc/eqGJXf.**

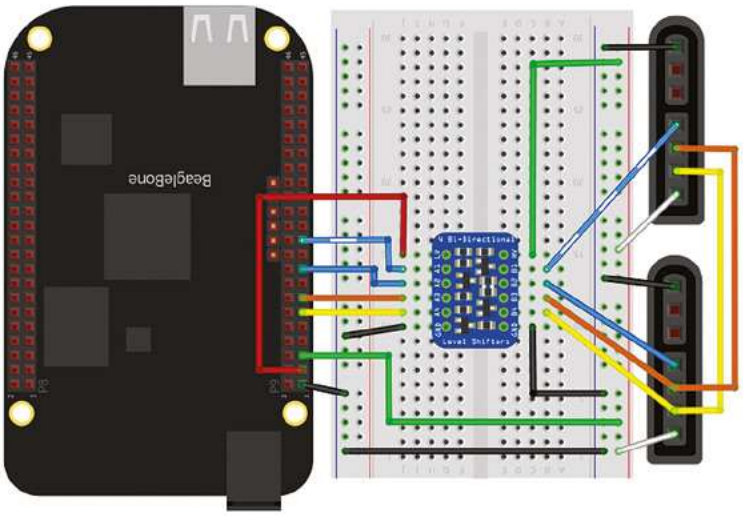


Figure 5 The interfacing circuit, complete with line-level conversion between the gamepads and the BBB

converter PCB use the 5V and GND signals from the BBB, so wire your BBB's P9.5 (5V) to the breadboard VCC bus and P9.1 (GND) to the breadboard ground bus. This way, both the PCB and the controllers have easy access to 5V and GND.

Wire the BBB to the line-level converter PCB by connecting P9.3 (3.3V) to LV, P9.11 (latch) to A1, P9.13 (clock) to A2, P9.17 (gamepad one data) to A3, P9.21 (gamepad two data) to A4, and your breadboard's ground bus to GND. On the high voltage side of your PCB, connect HV to the 5V VCC bus of your breadboard and GND to the ground bus. Wire the PCB to the gamepad connectors by connecting B1 (latch) to pin 2 of both connectors, B2 (clock) to pin 3 of both connectors, B3 (gamepad one data) to pin 4 of one gamepad connector, and B4 (gamepad two data) to pin 4 of the second gamepad connector.

FIDDLING WITH FIRMWARE

Now that you have hardware connected to the BBB, it is time to get some PRU firmware talking to it. Fetch the prugamepad project from GitHub onto your BBB. It contains the example code used in this article:

```
$ cd ~
$ git clone https://github.com/hendersa/
prugamepad
$ cd prugamepad
$ make
```

The PRU code implementing the gamepad protocol is in the source file **gamepad.p**. It uses only 61 assembly instructions, and the assembled firmware binary is only 244 bytes in size! There isn't much variety in the instructions being used, which shows that you don't have to understand every single PRU instruction to make useful firmware programs.

gamepad.p is divided into four main pieces: defining useful constants (lines 14–27), initialising shared memory (lines 30–47), polling gamepad button states (lines 53–110), and busy-wait delays for timing (lines 112–126). Don't worry about the details of shared memory initialisation, as this is boilerplate code provided by TI. It configures the PRU to share a memory location with Linux user space.

After PRU initialisation, the firmware's main loop begins and runs forever. This snippet of the main loop shows the basic mechanics of how the PRU handles setting GPIO outputs:

```
// Set latch
LBBO r0, GPIO_OUT, #0, #4
SET  r0.LATCH_BIT
SBBO r0, GPIO_OUT, #0, #4
```

First, the value held in a memory-mapped 32-bit GPIO output control register (**GPIO_OUT**) is loaded into a PRU register via the **LBBO** instruction. Each bit in this control register represents the output state of one GPIO pin, and these bits are either set to 1 using the **SET** instruction or cleared to 0 using the **CLR** instruction. Once the bits have been manipulated, the changed value is stored back into the control register via the **SBBO** instruction. When new state data is written into the control register, the physical states of each GPIO change immediately.

It is a similar process for input GPIOs. Each bit in a 32-bit GPIO input control register (**GPIO_IN**) contains bits representing the current state of input GPIOs. If a GPIO pin is high (connected to 3.3V), that GPIO's bit in the control register is set to 1. If that GPIO is low (connected to GND), the corresponding bit in the control register is cleared to 0. This snippet of the main loop shows the reading of GPIO input values.

```
// Read bit from each gamepad
LBBO r0, GPIO_IN, #0, #4
LSR  GP0_REG, r0, GP0_BIT
```

QUICK TIP

The **am335x_pru_** package contains the source code to prussdrv. The prugamepad Git repo also includes a copy.

CODE MADE CLEARER

There aren't PRU registers with names like 'GPIO_IN' and 'GPIO_OUT'. PASM provides a '#define' preprocessor directive for you to give convenient, easy-to-understand names to registers and constants. Use this to make your assembly code easier to read and understand.

```
AND  GP0_REG, GP0_REG, #1
LSR  GP1_REG, r0, GP1_BIT
AND  GP1_REG, GP1_REG, #1
```

The **LBB0** instruction reads all four bytes of **GPIO_IN** and copies it into a PRU register. Logically shifting right and masking (using the **LSR** and **AND** instructions) isolates a specific GPIO's bit. Once the bit is isolated, it is tested to see whether it is 1 or 0. By repeating this process of sampling and shifting, the state of all gamepad buttons and the D-pad are collected and then written out to a shared memory location.

TALKING TO THE PRU FROM USER SPACE

Now that you have firmware that talks to hardware, you'll need user space code that talks to the firmware. Such user space code has two important jobs. First, it must manage the PRU program life cycle: load firmware into a PRU, begin PRU execution, and stop PRU execution (if needed). Second, it must access memory shared with a PRU to collect any data that the PRU generates.

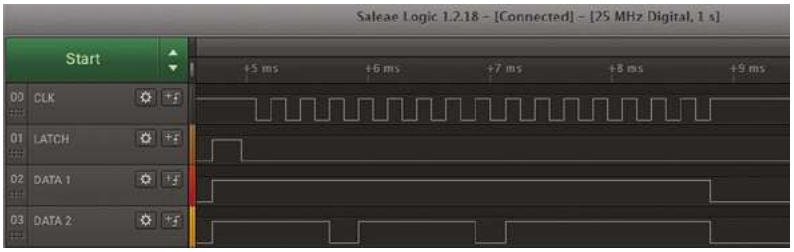
A very simple PRU interfacing program looks like this:

```
int main(void) {
  PRUSetup("./firmware.bin");
  do {
    printf("Shared memory: 0x%08x\n",
PRUState());
  } while (1);
PRUShutdown();
}
```

PRUSetup() loads the firmware onto a PRU and tells the PRU to begin execution. Likewise, **PRUShutdown()** stops the PRU's execution. **PRUState()** is called to repeatedly poll memory shared with the firmware. As the firmware runs, it continually updates the value held in shared memory.

It may seem like a lot of complex details are hidden inside those PRU functions, but there really aren't. TI provides a small, BSD-licensed C library called 'prussdrv' that performs PRU life cycle management. This makes implementing these PRU* functions simple since they are largely wrappers around calls to prussdrv code.

prugamepad implements a complete **PRUSetup()** and **PRUShutdown()**. It has a more complex sampling loop that interprets the value of each bit retrieved from shared memory. Any change in individual bits between two successive samples indicates that a gamepad button or D-pad has changed state. Try running it to see it in action:




```
$ cd ~/prugamepad
$ sudo ./setup.sh
$ sudo ./prugamepad
```

As prugamepad runs, the current state of both gamepads is dumped to the screen every time that the value held in shared memory changes. You can use a logic analyser or an oscilloscope to watch the gamepad protocol signal activity on the BBB's pins (**Figure 6**).

WHAT'S NEXT?

Now you've seen some of the basics of what PRUs can do, it is time to start experimenting! PRU-based projects include flying drones, driving 3D printers, acting as digital logic analysers, and controlling robots! Visit **hsmag.cc/CmQxjd** to learn more about how to use PRUs in your own projects.

Figure 6
In any digital interfacing project, a logic analyser is invaluable when troubleshooting and debugging your software and firmware

Below
The SNES is the greated games console of all time [citation needed]