iioinput-elce2019

Introduction to IIO and Input Drivers

Matt Porter <mporter@konsulko.com>

e-ale

The E-ALE (Embedded Apprentice Linux Engineer) is a series of seminars held at existing conferences covering topics which are fundamental to a Linux professional in the field of Embedded Linux.

This seminar will spend equal time on lecture and hands on labs at the end of each seminar which allow you to practice the material you've learned.

> This material makes the assumption that you have minimal experience with using Linux in general, and a basic understanding of general industry terms. The assumption is also made that you have access to your own computers upon which to practice this material.

More information can be found at **https://e-ale.org/**

Contents

Chapter 1

Preliminaries

1.1 Introductions

About Me

- CTO at Konsulko Group
- Using Linux since 1992
- Professional embedded Linux engineer since 1998
- Previously maintained kernel support for embedded PPC platforms, RapidIO subsystem, and Broadcom Mobile SoCs
- Various contributions around the kernel

About Konsulko Group

- Konsulko Group is a services company founded by embedded Linux veterans
- Community and commercial embedded, Linux, and Open Source Software development
- Linux Foundation training partners
- See https://www.konsulko.com for more information

1.2 Project Plan

Slides

- Download the slides for local reference
- https://cm.e-ale.org/2019/ELCE2019/iioinput/iioinput-elce2019-SLIDES.pdf

What To Do?

- The TechLab Cape has an accelerometer and a couple buttons.
- We'll learn how to read the position of the accelerometer axes.
- We'll also learn how to read the state of the buttons
- With that data, we have the foundation for a joystick driver.

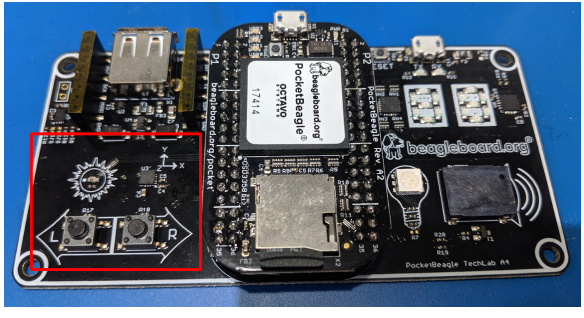

Figure 1.1: **TechLab Cape**

Exact Steps

- Understand how the accelerometer and buttons are interfaced in hardware
- Understand IIO and its purpose in the universe
- Learn how IIO exposes the accelerometer to the kernel and userspace
- Learn how to use libiio for userspace IIO interaction
- Write a accelerometer-based joystick driver and test it

Chapter 2

Hardware

2.1 TechLab Hardware

TechLab Schematic Overview

- https://github.com/beagleboard/capes/raw/master/pocketbeagle/TechLab/TechLab_Cape_sch.pdf

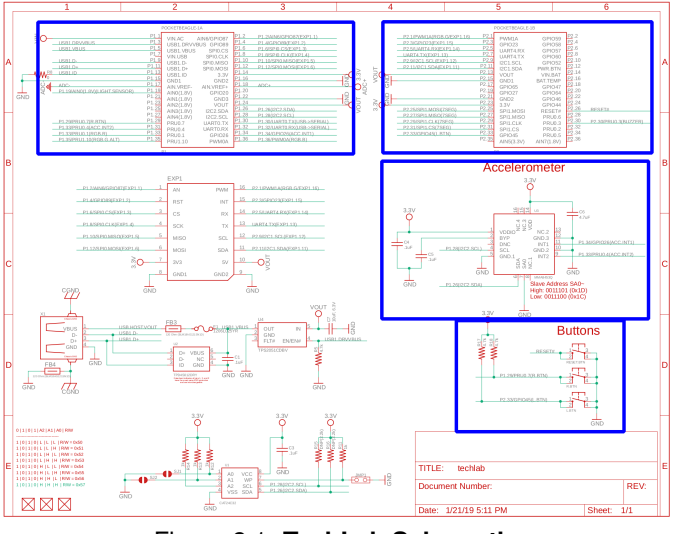

Figure 2.1: **TechLab Schematic**

TechLab Accelerometer

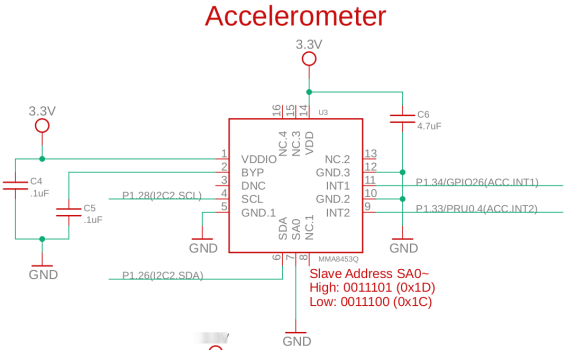

Figure 2.2: **TechLab Accelerometer**

- Signals:
  - **P1.28(I2C2.SCL)**
  - **P1.26(I2C2.SDA)**
- I2C address **0x1C**

TechLab Buttons

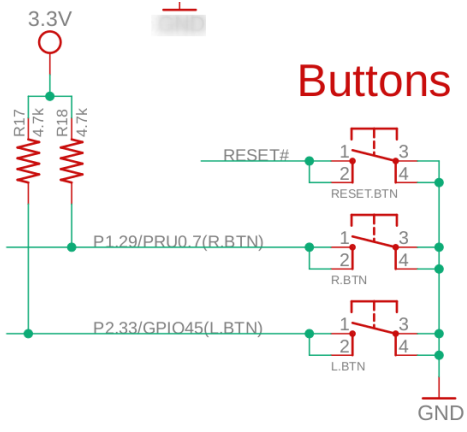

Figure 2.3: **TechLab Buttons**

- Signals:
  - **P1.29/PRU0.7(R.BTN)**
  - **P2.33/GPIO45(L.BTN)**

2.2 PocketBeagle Hardware

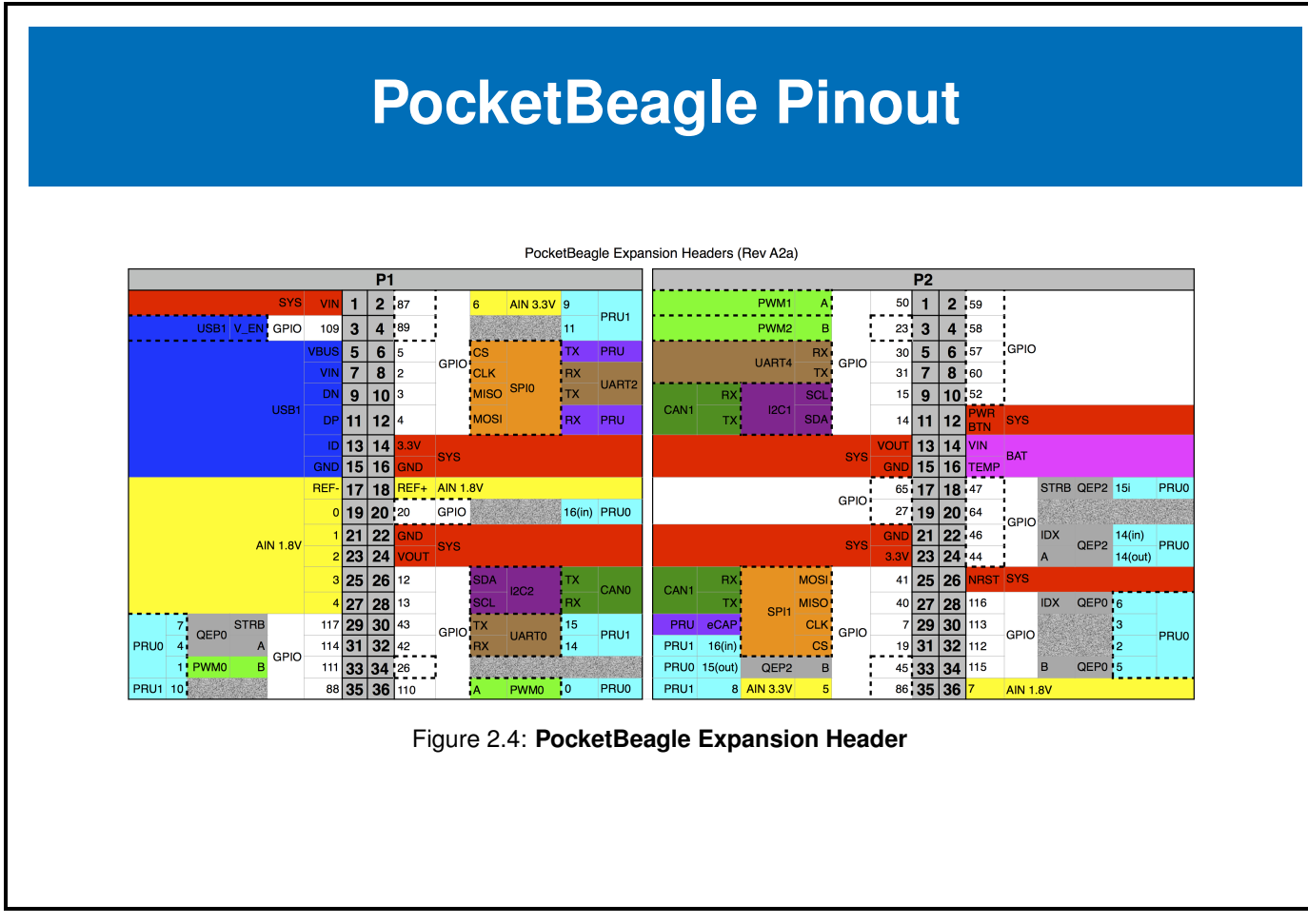

Figure 2.4: **PocketBeagle Expansion Header**

2.3 MMA8453

MMA8453 Resolution

https://www.nxp.com/docs/en/data-sheet/MMA8453Q.pdf

5.2 8-bit or 10-bit data

The measured acceleration data is stored in the OUT_X_MSB, OUT_X_LSB, OUT_Y_MSB, OUT_Y_LSB, OUT_Z_MSB, and OUT_Z_LSB registers as 2's complement 10-bit numbers. The most significant 8-bits of each axis are stored in OUT_X (Y, Z)_MSB, so applications needing only 8-bit results can use these three registers and ignore OUT_X,Y, Z_LSB. To do this, the F_READ bit in CTRL_REG1 must be set. When the F_READ bit is cleared, the fast read mode is disabled.

When the full-scale is set to 2 *g*, the measurement range is –2 *g* to +1.9961 *g*, and each count corresponds to 1 *g*/256 (3.9 mg) at 10-bits resolution. When the full-scale is set to 8 *g*, the measurement range is 8 *g* to +7.9844 *g*, and each count corresponds to 1 *g*/64 (15.6 mg) at 10-bits resolution. The resolution is reduced by a factor of 4 if only the 8-bit results are used. For more information on the data manipulation between data formats and modes, refer to NXP application note AN4076. There is a device driver available that can be used with the Sensor Toolbox demo board (LFSTBEB8451, 2, 3Q).

Figure 2.5: **MMA8453 Resolution**

- 10-bit samples (verified by inspection of mma8452.c kernel driver)
- In full resolution 2g mode, this means a range of -255 to 256

2.4 Summary

Hardware Investigation Results

- Accelerometer:
  - I2C SCL (**P1.28(I2C2.SCL)**)
  - I2C SDA (**P1.26(I2C2.SDA)**)
- Buttons:
  - Left (GPIO pull-up) (**P2.33/GPIO45(L.BTN)**)
  - Right (GPIO pull-up) (**P1.29/PRU0.7(R.BTN)**)

2.5 Device Tree

Overview

A Device Tree defines the hardware configuration for a platform. The following resources provide details on DT.

- Device Tree Specification https://github.com/devicetree-org/devicetree-specification/releases/download/v0.2/devicetree-specification-v0.2.pdf Register input devices with the kernel
- Kernel Device Tree Documentation https://www.kernel.org/doc/Documentation/devicetree/
- Device Tree for Dummies https://elinux.org/images/f/f9/Petazzoni-device-tree-dummies_0.pdf

IIO provider binding

Documentation/devicetree/bindings/iio/iio-bindings.txt:

```
==IIO providers==

Required properties:
#io-channel-cells: Number of cells in an IIO specifier; Typically 0 for nodes
                   with a single IIO output and 1 for nodes with multiple
                   IIO outputs.

Example for a simple configuration with no trigger:

        adc: voltage-sensor@35 {
                compatible = "maxim,max1139";
                reg = <0x35>;
                #io-channel-cells = <1>;
        };
        .
        .
        .
```

IIO consumer binding

Documentation/devicetree/bindings/iio/iio-bindings.txt:

```
==IIO consumers==

Required properties:
io-channels:    List of phandle and IIO specifier pairs, one pair
                for each IIO input to the device. Note: if the
                IIO provider specifies '0' for #io-channel-cells,
                then only the phandle portion of the pair will appear.

Optional properties:
io-channel-names:
                List of IIO input name strings sorted in the same
                order as the io-channels property. Consumers drivers
                will use io-channel-names to match IIO input names
                with IIO specifiers.

For example:

        device {
                io-channels = <&adc 1>, <&ref 0>;
                io-channel-names = "vcc", "vdd";
        };
```

MMA8453 binding

Documentation/devicetree/bindings/iio/accel/mma8452.txt:

```
Freescale MMA8451Q, MMA8452Q, MMA8453Q, MMA8652FC, MMA8653FC or FXLS8471Q
triaxial accelerometer

Required properties:

  - compatible: should contain one of
    * "fsl,mma8451"
    * "fsl,mma8452"
    * "fsl,mma8453"
    * "fsl,mma8652"
    * "fsl,mma8653"
    * "fsl,fxls8471"

  - reg: the I2C address of the chip

Optional properties:

  - interrupts: interrupt mapping for GPIO IRQ

  - interrupt-names: should contain "INT1" and/or "INT2", the accelerometer's
                     interrupt line in use.

Example:

        mma8453fc@1d {
                compatible = "fsl,mma8453";
                reg = <0x1d>;
                interrupt-parent = <&gpio1>;
                interrupts = <5 0>;
                interrupt-names = "INT2";
```

Pinctrl client binding

Documentation/devicetree/bindings/pinctrl/pinctrl-bindings.txt:

```
Required properties:
pinctrl-0:      List of phandles, each pointing at a pin configuration
                node. These referenced pin configuration nodes must be child
                nodes of the pin controller that they configure.
.
.
.
Optional properties:
pinctrl-1:      List of phandles, each pointing at a pin configuration
                node within a pin controller.
.
.
.
For example:

        /* For a client device requiring named states */
        device {
                pinctrl-names = "active", "idle";
                pinctrl-0 = <&state_0_node_a>;
                pinctrl-1 = <&state_1_node_a &state_1_node_b>;
        };
```

GPIO consumer binding

Documentation/devicetree/bindings/gpio/gpio.txt:

```
.
.
.
GPIO properties should be named "[<name>-]gpios", with <name> being the purpose
of this GPIO for the device.
.
.
.
Example of a node using GPIOs:

        node {
                enable-gpios = <&qe_pio_e 18 GPIO_ACTIVE_HIGH>;
        };

GPIO_ACTIVE_HIGH is 0, so in this example gpio-specifier is "18 0" and encodes
GPIO pin number, and GPIO flags as accepted by the "qe_pio_e" gpio-controller.
.
.
.
```

Chapter 3

Linux Subsystems

3.1 IIO Subsystem

Overview

- The Linux IIO subsystem is a framework to support sensors and any type of device with ADCs or DACs
- https://www.kernel.org/doc/html/v5.0/driver-api/iio/intro.html
- IIO provides a core framework to support device drivers in a common manner
- IIO also provides an interface for in-kernel users of IIO devices
- IIO provides a high-latency userspace interface via sysfs
- IIO provides an efficient buffered interface via character device

IIO architecture

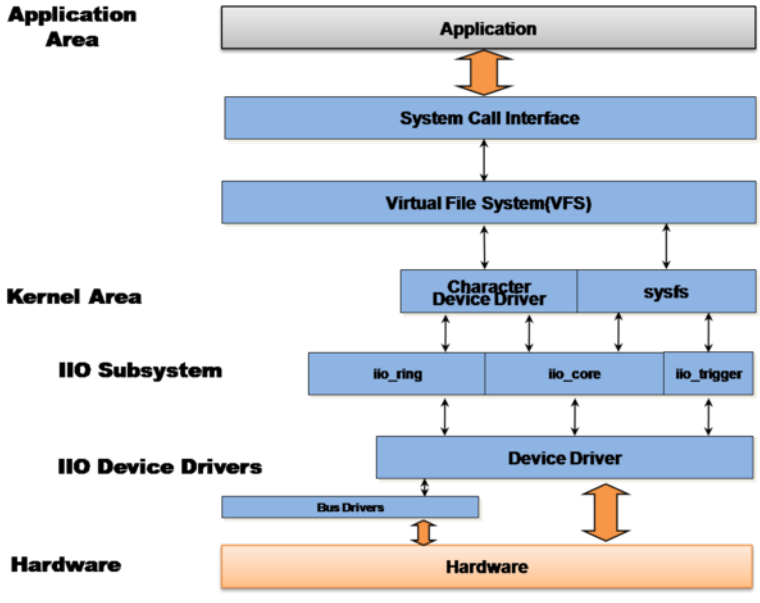

Figure 3.1: **IIO architecture**

IIO concepts

- Producer/Consumer model
  - Sensor/ADC drivers are producers of samples.
  - In-kernel drivers or userspace clients are consumers of samples.
- Ring buffer (efficient buffered access to samples)
  - Hardware and software circular buffer support for producers to place samples.
- Triggers (external events to **trigger** capture of a sample)
  - GPIO
  - RTC
  - sysfs file
- Scaled samples
  - Ability to provide both raw samples and scaled (in units relevant to the sensor e.g. mA, mV, lumens, foot pounds per fortnight, etc.)

IIO userspace ABI

The canonical ABI documentation is here https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/ABI/testing/sysfs-bus-iio

Focusing on the polled sysfs interface:

- IIO device nodes
  - /sys/bus/iio/devices/iio:deviceN
- IIO device attribute nodes
  - /sys/bus/iio/devices/iio:deviceN/in_*_raw
  - /sys/bus/iio/devices/iio:deviceN/in_*_scale
    - Variations for each type of sample (voltage, current, power, temperature

IIO temp sensor example

Example graciously provided by Matt Ranostay from his https://elinux.org/images/b/ba/ELC_2017_-_Industrial_IO_and_You-_Nonsense_Hacks!.pdf presentation.

```
$ cd /sys/bus/iio/devices/iio:device0
$ cat in_temp_raw
98
$ cat in_temp_ambient_raw
416
$ cat in_temp_scale
250
$ cat in_temp_ambient_scale
62.500000
```

libiio

- **libiio** is a higher level userspace library for working with IIO devices. The official documentation is at http://analogdevicesinc.github.io/libiio/. It was created in response to the difficulty of working with the low-level IIO userspace ABI for buffered I/O management.
- **libiio** provides high level APIs to manage
  - polled sysfs read/write attributes
  - trigger events
  - buffered samples
- **libiio** is used by the **iiod** server to provide networked access to IIO data and provides several utilities like **iio_info** for retrieving data from IIO.
- **python-libiio** provides Python bindings for libiio.

Kernel consumer API

- Used by other kernel drivers to build functionality on top of an IIO hardware device driver
- https://elixir.bootlin.com/linux/latest/source/include/linux/iio/consumer.h
- Get an IIO device channel

```
struct iio_channel *devm_iio_channel_get(struct device *dev,
                                         const char *consumer_channel);
```

- Get a channel type from a device channel

```
int iio_get_channel_type(struct iio_channel *channel,
                         enum iio_chan_type *type);
```

- Read a raw (unprocessed) value from a device channel

```
int iio_read_channel_raw(struct iio_channel *chan,
                         int *val);
```

3.2 GPIO Subsystem

Overview

- The Linux GPIO subsystem is a framework to support control of General Purpose Input/Output pins
- https://www.kernel.org/doc/Documentation/gpio/gpio.txt
- No longer using the legacy GPIO APIs.
  - https://www.kernel.org/doc/Documentation/gpio/gpio-legacy.txt
- Prefer the descriptor-based consumer GPIO APIs.
  - https://www.kernel.org/doc/Documentation/gpio/consumer.txt

Consumer

- Get a GPIO descriptor

```
struct gpio_desc *devm_gpiod_get_index(struct device *dev,
                                       const char *con_id,
                                       enum gpiod_flags flags)
```

  - **con_id** is typically the prefix of a **Device Tree gpio(s)** property. e.g. a **power-gpio** property would require **power** for **con_id**
    - https://www.kernel.org/doc/Documentation/gpio/board.txt
  - **flags** are optional and can include direction and/or initial value for a GPIO. e.g. **GPIOD_IN** for an input

- Get a GPIO value (0 for low, nonzero for high)

```
int gpiod_get_value(const struct gpio_desc *desc);
```

3.3 Input Subsystem

Overview

- The Linux Input subsystem is a framework to support all types of input devices
- https://www.kernel.org/doc/html/v5.0/input/input.html
- Consists of the core **input module**, **device drivers**, and **event handlers**

Device Drivers

- **Device drivers** interface with hardware and provide events to the **input module**
- Examples are:
  - **gpio_keys**
  - **hid-generic**
  - **usbmouse**

Event Handlers

- **Event handlers** interface with the **input module** and pass events to other kernel subsystems or userspace
  - **evdev** passes generic input events to userspace. Devices are in **/dev/input**:

```
crw-r--r--   1 root     root      13,  64 Apr  1 10:49 event0
crw-r--r--   1 root     root      13,  65 Apr  1 10:50 event1
crw-r--r--   1 root     root      13,  66 Apr  1 10:50 event2
crw-r--r--   1 root     root      13,  67 Apr  1 10:50 event3
```

  - **joydev** passes joystick events to userspace. Devices are in **/dev/input**:

```
crw-r--r--   1 root     root      13,   0 Apr  1 10:50 js0
crw-r--r--   1 root     root      13,   1 Apr  1 10:50 js1
crw-r--r--   1 root     root      13,   2 Apr  1 10:50 js2
crw-r--r--   1 root     root      13,   3 Apr  1 10:50 js3
```

evdev

- **evdev** nodes support blocking/non-blocking **read** and **select**
- Reading an **evdev** node returns a **struct input_event**:

```
struct input_event
{
        struct timeval time;
        unsigned short type;
        unsigned short code;
        unsigned int value;
};
```

evtest

- **evtest** can be used to test evdev events at the command line
- Example:

```
$ evtest /dev/input/event0
Input driver version is 1.0.1
Input device ID: bus 0x3 vendor 0x46d product 0x1028 version 0x111
Input device name: "Logitech M570"
Supported events:
Event type 0 (EV_SYN)
Event type 1 (EV_KEY)
Event code 272 (BTN_LEFT)
...
Event type 4 (EV_MSC)
Event code 4 (MSC_SCAN)
Properties:
Testing ... (interrupt to exit)
Event: time 1540159516.312712, type 4 (EV_MSC), code 4 (MSC_SCAN), value 90001
Event: time 1540159516.312712, type 1 (EV_KEY), code 272 (BTN_LEFT), value 1
Event: time 1540159516.312712, ------------- SYN_REPORT ------------
```

jstest

- **jstest** can be used to test joystick events at the command line
- Example:

```
# jstest /dev/input/js0
Driver version is 2.1.0.
Joystick (python-uinput) has 2 axes (X, Y)
and 2 buttons (BtnA, BtnB).
Testing ... (interrupt to exit)
Axes:  0:  1028  1:  6425 Buttons:  0:off  1:off
```

Input device driver API

- Input devices described by **struct input_dev**:

```
struct input_dev {
        const char *name;
        ...
        unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
        unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];
        ...
        int (*event)(struct input_dev *dev, unsigned int type, unsigned int code, int value);
        ...
};
```

- A **struct input_dev** is allocated using **devm_input_allocate_device** before event types and codes are configured and handlers are filled in.
- **input_register_device** and **input_unregister_device** are used to register and unregister the device, respectively

Input polled device driver API

- Simple devices that can to be polled on a timer basis can be implemented using the simpler **struct input_polled_dev**:

```
struct input_polled_dev {
        ...
        void (*poll)(struct input_polled_dev *dev);
        unsigned int poll_interval; /* msec */
        ...
        struct input_dev *input;
};
```

- Allocate a **struct input_polled_dev** with **devm_input_allocate_polled_device**
- It is necessary only to fill in the **poll** handler, **poll_interval**, and fill in the **input** device configuration (**name, id, evkey, keybit** fields and use **input_set_abs_params** if absolute events are supported)
- Register the polled device with **input_register_polled_device**
- In the **poll** handler, read the hardware and use **input_report_*** to queue events to be reported and **input_sync** to flush the queued events

Chapter 4

Labs

4.1 Preparation

Install TechLab image

- Download image from https://debian.beagleboard.org/images/techlab-workshop-2019-01-24.img.xz
- Write image to uSD:

```
$ xzcat techlab-workshop-2019-01-24.img.xz | sudo dd status=progress of=/dev/sdX
```

Install dependencies

- Download lab preparation tarball from https://cm.e-ale.org/2019/ELCE2019/iioinput/iioinput.tar.bz2
- Copy tarball from host to target

```
$ scp iioinput.tar.bz2 debian@192.168.7.2:~
```

- Set up passwordless root login:

```
$ sudo tar jxf iioinput.tar.bz2 -C /root
$ sudo /root/bin/pwless-root.sh
$ sudo reboot
```

  Login as **root**

- Run dependency install script:

```
root@beaglebone:~# bin/install.sh
```

Note: Passwordless root is optional. One can also execute all lab commands as the default debian user by using sudo

4.2 Lab 1

IIO sysfs interface

In Lab1, we will accomplish the following:

- Within sysfs, locate the IIO device corresponding to the accelerometer
- Read the X and Y axes using a sysfs file

Find the accelerometer IIO device

Confirm that the IIO subsystem has exported devices to sysfs:

```
root@beaglebone:~# ls /sys/bus/iio/devices/
iio:device0  iio:device1
```

Which is our device?

```
root@beaglebone:~# ls /sys/bus/iio/devices/iio\:device1
buffer                                          in_accel_y_calibbias
current_timestamp_clock                          in_accel_y_raw
dev                                           in_accel_z_calibbias
events                                           in_accel_z_raw
in_accel_filter_high_pass_3db_frequency           name
in_accel_filter_high_pass_3db_frequency_available  of_node
in_accel_oversampling_ratio                      power
in_accel_oversampling_ratio_available              sampling_frequency_available
in_accel_sampling_frequency                      scan_elements
in_accel_scale                                   subsystem
in_accel_scale_available                       trigger
in_accel_x_calibbias                             uevent
in_accel_x_raw
```

That's definitely an accelerometer.

Read the X and Y axes value

Read the raw X axis value:

```
root@beaglebone:~# cat /sys/bus/iio/devices/iio\:device1/in_accel_x_raw
-10
```

Read the raw Y axis value:

```
root@beaglebone:~# cat /sys/bus/iio/devices/iio\:device1/in_accel_y_raw
-7
```

Try moving the board and reading the values again, what are the results?

4.3 Lab 2

Using libiio utilities

libiio is a helper library that abstracts away some of the complications of the raw IIO userspace ABI.

In Lab2, we will accomplish the following:

- Using libiio, discover all devices

Find our accelerometer device

Example which devices the IIO subsystem has exported:

```
root@beaglebone:/# iio_info
                Library version: 0.16 (git tag: v0.16)
                Compiled with backends: local xml ip usb serial
                IIO context created with local backend.
                Backend version: 0.16 (git tag: v0.16)
                Backend description string: Linux beaglebone 4.14.71-ti-r80 #1 SMP PREEMPT Fri Oct 5 23:50:11 UTC 2018 armv7l
                IIO context has 1 attributes:
                        local,kernel: 4.14.71-ti-r80
                IIO context has 2 devices:
                .
                .
                .
                        iio:device1: mma8453 (buffer capable)
                4 channels found:
                        accel_x:  (input, index: 0, format: be:s10/16>>6)
                        9 channel-specific attributes found:
                                attr  0: calibbias value: 0
                                attr  1: filter_high_pass_3db_frequency value: 0.000000
                                attr  2: filter_high_pass_3db_frequency_available value: 0.250000 0.500000 1.000000 2.000000
                                attr  3: oversampling_ratio value: 4
                                attr  4: oversampling_ratio_available value: 4 32 2
                                attr  5: raw value: -7
                                attr  6: sampling_frequency value: 50.000000
                                attr  7: scale value: 0.038307
                                attr  8: scale_available value: 0.153228 0.076614 0.038307
                        accel_y:  (input, index: 1, format: be:s10/16>>6)
                .
                .
                .
```

4.4 Lab 3

Device Tree

In Lab3, we will accomplish the following:

- Learn Pocketbeagle DT specifics
- Configure the platform DT for our joystick device
- Overlay the base DT with the joystick device

pocketbeagle.dts P1.29 pinctrl node

```
/* P1_29 (ZCZ ball A14) pruin */
P1_29_pinmux {
        compatible = "bone-pinmux-helper";
        status = "okay";
        pinctrl-names = "default", "gpio", "gpio_pu", "gpio_pd",
                        "gpio_input", "qep", "pruout", "pruin";
        pinctrl-0 = <&P1_29_default_pin>;
        pinctrl-1 = <&P1_29_gpio_pin>;
        pinctrl-2 = <&P1_29_gpio_pu_pin>;
        pinctrl-3 = <&P1_29_gpio_pd_pin>;
        pinctrl-4 = <&P1_29_gpio_input_pin>;
        pinctrl-5 = <&P1_29_qep_pin>;
        pinctrl-6 = <&P1_29_pruout_pin>;
        pinctrl-7 = <&P1_29_pruin_pin>;
};
```

pocketbeagle.dts P2.33 pinctrl node

```
/* P2_33 (ZCZ ball R12) */
        P2_33_pinmux {
                compatible = "bone-pinmux-helper";
                status = "okay";
                pinctrl-names = "default", "gpio", "gpio_pu", "gpio_pd",
                                "gpio_input", "qep", "pruout";
                pinctrl-0 = <&P2_33_default_pin>;
                pinctrl-1 = <&P2_33_gpio_pin>;
                pinctrl-2 = <&P2_33_gpio_pu_pin>;
                pinctrl-3 = <&P2_33_gpio_pd_pin>;
                pinctrl-4 = <&P2_33_gpio_input_pin>;
                pinctrl-5 = <&P2_33_qep_pin>;
                pinctrl-6 = <&P2_33_pruout_pin>;
        };
```

P1.29 GPIO resource

http://www.ti.com/lit/ds/symlink/am3352.pdf

ZCZ BALL NUMBER [1]	PIN NAME [2]	SIGNAL NAME [3]	MODE [4]
A14	MCASP0_AHCLKX	mcasp0_ahclkx	0
		eQEP0_strobe	1
		mcasp0_axr3	2
		mcasp1_axr1	3
		EMU4	4
		pr1_pru0_pru_r30_7	5
		pr1_pru0_pru_r31_7	6
		gpio3_21	7

Figure 4.1: **P1.29 ZCZ A14 (gpio3_21)**

P2.33 GPIO resource

http://www.ti.com/lit/ds/symlink/am3352.pdf

R12	GPMC_AD13	gpmc_ad13	0
		lcd_data18	1
		mmc1_dat5	2
		mmc2_dat1	3
		eQEP2B_in	4
		pr1_mii0_txd1	5
		pr1_pru0_pru_r30_15	6
		gpio1_13	7

Figure 4.2: **P2.33 ZCZ R12 (gpio1_13)**

Describing the Joystick Hardware

- Specify the MMA8453 accelerometer
- Specify the Left and Right buttons used by joystick
- Specifiy the MMA8453 channels used by joystick
- Specify pinmux settings needed for buttons
- Specify GPIO resources needed for buttons

Joystick DT Overlay

```
/dts-v1/;
/plugin/;

#include <dt-bindings/gpio/gpio.h>

/ {
        fragment@0 {
                target = <&i2c2>;
                __overlay__ {
                        #address-cells = <1>;
                        #size-cells = <0>;
                        mma8453: mma8453@1c {
                                #io-channel-cells = <1>;
                                status = "okay";
                                compatible = "fsl,mma8453";
                                reg = <0x1c>;
                        };

                };
        };
```

Joystick DT Overlay (continued)

```
        fragment@1 {
                target = <&ocp>;
                __overlay__ {
                        P2_33_pinmux { status = "disabled"; };  /* Left - gpio3_21 */
                        P1_29_pinmux { status = "disabled"; };  /* Right - gpio1_13 *
                        cape-universal { status = "disabled"; };

                        joystick {
                                compatible = "bborg,techjoy";
                                pinctrl-0 = <&P2_33_gpio_input_pin>,
                                            <&P1_29_gpio_input_pin>;
                                io-channels = <&mma8453 0>, <&mma8453 1>;
                                io-channel-names = "accel_x", "accel_y";
                                button-gpios = <&gpio3 21 GPIO_ACTIVE_LOW>,
                                               <&gpio1 13 GPIO_ACTIVE_LOW>;
                        };
                };
        };
};
```

Build/Install joystick overlay

```
# cp dts/techjoy.dts /opt/source/bb.org-overlays/src/arm/
# cd /opt/source/bb.org-overlays
# make src/arm/techjoy.dtbo
# make install
```

Deploy joystick overlay

Edit /**boot/uEnv.txt** disabling/enabling overlays as follows:

```
#uboot_overlay_addr0=/lib/firmware/PB-I2C2-ACCEL-TECHLAB-CAPE.dtbo
#uboot_overlay_addr1=/lib/firmware/PB-PWM-RGB-TECHLAB-CAPE.dtbo
#uboot_overlay_addr2=/lib/firmware/PB-SPI1-7SEG-TECHLAB-CAPE.dtbo
#uboot_overlay_addr3=/lib/firmware/PB-SPI0-OLEDC-CLICK.dtbo
uboot_overlay_addr0=/lib/firmware/techjoy.dtbo
```

Reboot the board to activate the overlay:

```
# reboot
```

4.5 Lab 4

Userspace Input Driver

In Lab 4, we will accomplish the following:

- Learn about the kernel uinput module, python-uinput, python-libiio, and python-periphery
- Create a userspace joystick driver
- Test the joystick driver

uinput drivers

Linux userspace input driver provides methods to:

- Register input devices with the kernel
- Register types of events the userspace driver may generate (and ranges if applicable. e.g. BTN, KEY, ABS
- Trigger events from userspace

  The uinput driver interface and C APIs are described in detail at https://www.kernel.org/doc/html/v5.0/input/uinput.html

Write a uinput joystick driver in Python

- **python-uinput** provides Python support for uinput drivers.
- **python-libiio** provides Python support for IIO devices.
- **python-periphery** provides access to various embedded I/O facilities including GPIO.

Python joystick driver

```
#!/usr/bin/env python3

from periphery import GPIO
import iio
import time
import uinput

events = (
    uinput.ABS_X + (-255,256, 0, 0),
    uinput.ABS_Y + (-255,256, 0, 0),
    uinput.BTN_A,
    uinput.BTN_B,
    )

with uinput.Device(events) as device:
    contexts = iio.scan_contexts()
    uri = next(iter(contexts), None)
    ctx = iio.Context(uri)
```

Python joystick driver (continued)

```
mma8453 = ctx.find_device("mma8453")

x = mma8453.find_channel("accel_x", False)
y = mma8453.find_channel("accel_y", False)

a = GPIO(45, "in")
b = GPIO(117, "in")

while (1):
    device.emit(uinput.ABS_X, int(x.attrs["raw"].value))
    device.emit(uinput.ABS_Y, int(y.attrs["raw"].value))
    device.emit(uinput.BTN_A, not a.read())
    device.emit(uinput.BTN_B, not b.read())
    time.sleep(.1)
```

Start the driver:

```
root@beaglebone:~# ./techjoy.py &
```

Test the joystick driver (evtest)

```
root@beaglebone:~# evtest /dev/input/event1
Input driver version is 1.0.1
Input device ID: bus 0x19 vendor 0x0 product 0x0 version 0x0
Input device name: "ocp:joystick"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
    Event code 304 (BTN_SOUTH)
    Event code 305 (BTN_EAST)
  Event type 3 (EV_ABS)
    Event code 0 (ABS_X)
      Value    -31
      Min     -255
      Max      256
    Event code 1 (ABS_Y)
      Value     40
      Min     -255
      Max      256
Properties:
Testing ... (interrupt to exit)
Event: time 1566259060.295970, type 3 (EV_ABS), code 0 (ABS_X), value -31
Event: time 1566259060.295970, ------------- SYN_REPORT ------------
Event: time 1566259060.547851, type 3 (EV_ABS), code 0 (ABS_X), value -29
Event: time 1566259060.547851, ------------- SYN_REPORT ------------
Event: time 1566259060.631933, type 3 (EV_ABS), code 0 (ABS_X), value -30
Event: time 1566259060.631933, type 3 (EV_ABS), code 1 (ABS_Y), value 41
.
.
.
```

Try moving the board around, what are the results?

Test the joystick driver (jstest)

```
root@beaglebone:~# jstest /dev/input/js0
Driver version is 2.1.0.
Joystick (python-uinput) has 2 axes (X, Y)
and 2 buttons (BtnA, BtnB).
Testing ... (interrupt to exit)
Axes:  0:   642  1:  2441 Buttons:  0:off  1:off ^C
```

Try moving the board around, what are the results?

Play Tetris

ssh to the PocketBeagle for better performance

```
root@beaglebone:~# tar zxf $HOME/test/vitetris-0.57.tar.gz
root@beaglebone:~# cd vitetris-0.57
root@beaglebone:~# ./configure
root@beaglebone:~# make
root@beaglebone:~# ./tetris
```

4.6 Lab 5

Write a kernel joystick driver

In Lab 5, we will accomplish the following:

- Create a kernel joystick driver
- Test the joystick driver

What is needed?

- Platform driver matching on DT compatible string
- Get the IIO accel channels corresponding to the X/Y axes
- Get the GPIOs corresponding to the buttons
- Register a polled input device
- Read current values from accelerometer and GPIOs
- Report the input events

Implementation: Skeleton

```
#include <linux/gpio/consumer.h>
#include <linux/iio/consumer.h>
#include <linux/iio/types.h>
#include <linux/input.h>
#include <linux/input-polldev.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/of.h>
#include <linux/platform_device.h>

...

#ifdef CONFIG_OF
static const struct of_device_id techjoy_of_match[] = {
        { .compatible = "bborg,techjoy", },
        { }
};
MODULE_DEVICE_TABLE(of, techjoy_of_match);
#endif

static struct platform_driver __refdata techjoy_driver = {
        .driver = {
                .name = "techjoy",
                .of_match_table = of_match_ptr(techjoy_of_match),
        },
        .probe = techjoy_probe,
        .remove = techjoy_remove,
};
module_platform_driver(techjoy_driver);

MODULE_AUTHOR("Matt Porter");
MODULE_DESCRIPTION("Techlab cape joystick input driver");
MODULE_LICENSE("GPL v2");
```

Implemention: probe() 1/4

```
struct techjoy {
        struct input_polled_dev *poll_dev;
        char phys[32];
        struct gpio_desc *btn_a, *btn_b;
        struct iio_channel *accel_x, *accel_y;
};

...

static int techjoy_probe(struct platform_device *pdev)
{
        struct device *dev = &pdev->dev;
        struct techjoy *p;
        struct input_polled_dev *poll_dev;
        struct input_dev *input;
        enum iio_chan_type type;
        int ret;

        p = devm_kzalloc(dev, sizeof(*p), GFP_KERNEL);
        if (!p)
                return -ENOMEM;

        p->btn_a = devm_gpiod_get_index(dev, "button", 0, GPIOD_IN);
        if (IS_ERR(p->btn_a)) {
                ret = PTR_ERR(p->btn_a);
                dev_err(dev, "failed to get btn_a GPIO: %d\n", ret);
                return ret;
        }
```

Implementation: probe() 2/4

```
p->btn_b = devm_gpiod_get_index(dev, "button", 1, GPIOD_IN);
if (IS_ERR(p->btn_b)) {
        ret = PTR_ERR(p->btn_b);
        dev_err(dev, "failed to get btn_b GPIO: %d\n", ret);
        return ret;
}

p->accel_x = devm_iio_channel_get(dev, "accel_x");
if (IS_ERR(p->accel_x))
        return PTR_ERR(p->accel_x);

if (!p->accel_x->indio_dev)
        return -ENXIO;

ret = iio_get_channel_type(p->accel_x, &type);
if (ret < 0)
        return ret;

if (type != IIO_ACCEL) {
        dev_err(dev, "not accelerometer channel %d\n", type);
        return -EINVAL;
}
```

Implementation:probe() 3/4

```
p->accel_y = devm_iio_channel_get(dev, "accel_y");
if (IS_ERR(p->accel_y))
        return PTR_ERR(p->accel_y);

if (!p->accel_y->indio_dev)
        return -ENXIO;

ret = iio_get_channel_type(p->accel_y, &type);
if (ret < 0)
        return ret;

if (type != IIO_ACCEL) {
        dev_err(dev, "not accel channel %d\n", type);
        return -EINVAL;
}

poll_dev = devm_input_allocate_polled_device(dev);
if (!poll_dev) {
        dev_err(dev, "unable to allocate input device\n");
        return -ENOMEM;
}

poll_dev->poll_interval = 50;
poll_dev->poll = techjoy_poll;
poll_dev->private = p;

p->poll_dev = poll_dev;
platform_set_drvdata(pdev, p);
```

Implementation: probe() 4/4

```
        input = poll_dev->input;
        input->name = pdev->name;
        sprintf(p->phys, "techjoy/%s", input->dev.kobj.name);
        input->phys = p->phys;
        input->id.bustype = BUS_HOST;

        __set_bit(EV_KEY, input->evbit);
        __set_bit(BTN_A, input->keybit);
        __set_bit(BTN_B, input->keybit);
        __set_bit(EV_ABS, input->evbit);
        /* Hardcode min/max to the resolution of the 10-bit accelerometer w/ 2g scaling */
        input_set_abs_params(input, ABS_X, -255, 256, 0, 0);
        input_set_abs_params(input, ABS_Y, -255, 256, 0, 0);

        ret = input_register_polled_device(poll_dev);
        if (ret) {
                dev_err(dev, "unable to register input device: %d\n", ret);
                return ret;
        };

        return 0;
}
```

Implementation: remove()

```
static int techjoy_remove(struct platform_device *pdev)
{
        struct techjoy *p = platform_get_drvdata(pdev);

        input_unregister_polled_device(p->poll_dev);

        return 0;
}
```

Implementation: poll()

```
static void techjoy_poll(struct input_polled_dev *dev)
{
        struct techjoy *p = dev->private;
        int ret, a, b, x, y;

        a = gpiod_get_value(p->btn_a);
        input_report_key(dev->input, BTN_A, a);

        b = gpiod_get_value(p->btn_b);
        input_report_key(dev->input, BTN_B, b);

        ret = iio_read_channel_raw(p->accel_x, &x);
        if (unlikely(ret < 0))
                return;
        input_report_abs(dev->input, ABS_X, x);

        ret = iio_read_channel_raw(p->accel_y, &y);
        if (unlikely(ret < 0))
                return;
        input_report_abs(dev->input, ABS_Y, y);

        input_sync(dev->input);
}
```

Implementation: Makefile

```
obj-m := techjoy.o
```

Build and deploy joystick module

```
root@beaglebone:~# make -C /lib/modules/$(uname -r)/build M=$PWD
root@beaglebone:~# make -C /lib/modules/$(uname -r)/build M=$PWD modules_install
root@beaglebone:~# depmod -a
root@beaglebone:~# pgrep python | xargs kill
root@beaglebone:~# modprobe techjoy
```

Test the joystick driver

Repeat **evtest** and **vitetris** tests from Lab 4.

Reference Implementation

https://github.com/konsulko/techjoy