

ES6

**ADC**

---



**Rick van Melis & Simon Lit**

**10-05-2019**

---

---

<b>Part 1 - Exploration &amp; Implementation</b>	<b>3</b>
Parameters for request_irq	3
Freeing IRQ	4
Solve multiple interrupts on button press (level to edge triggered)	5
Turning on the ADC	6
Starting the ADC	6
Reading ADC Value	7
Reentrancy	8
<b>Part 2 - Verification</b>	<b>11</b>
Measuring GP Interrupted Time	11
Measuring A/D Conversion Time	13
<b>Part 3 - User manual</b>	<b>15</b>
<b>Sources</b>	<b>16</b>

---

## Part 1 - Exploration & Implementation

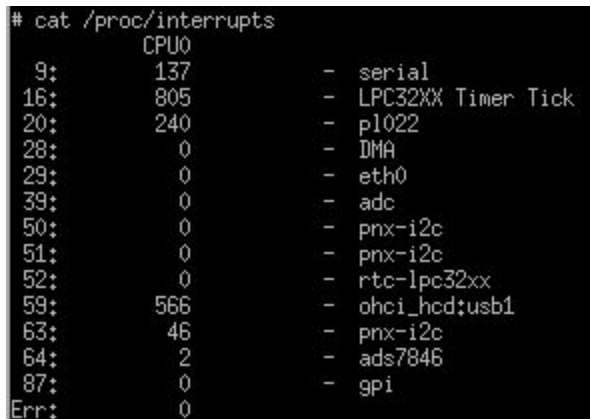
### Parameters for *request\_irq*

*Request\_irq* is an method for allocating a given interrupt line. It takes multiple arguments as specified in the below function definition.

```
int request_irq(unsigned int irq,
               irq_handler_t handler,
               unsigned long flags,
               const char *name,
               void *dev)
```

The fourth parameter is *name*, this is the name of the device associated with the interrupt. In our case for the *gp\_interrupt* the name *gpi* would make sense because it is a button that gives input. For *adc\_interrupt* the name *adc* would makes sense because that is the device that generates the interrupt.

This name is visible though using *cat* on */proc/interrupts*. As can be seen in the following screenshot.



```
# cat /proc/interrupts
          CPU0
 9:         137          - serial
16:         805          - LPC32XX Timer Tick
20:         240          - p1022
28:          0          - DMA
29:          0          - eth0
39:          0          - adc
50:          0          - pnx-i2c
51:          0          - pnx-i2c
52:          0          - rtc-lpc32xx
59:         566          - ohci_hcd:usb1
63:          46          - pnx-i2c
64:           2          - ads7846
87:          0          - gpi
Err:          0
```

The first parameter of *request\_irq* specifies an interrupt number for some devices this value is typically hard-coded. For most other devices, it is probed or otherwise determined dynamically. On the LPC32xx they are hardcoded.

The assignment instructs to search for *ADC\_INT* and *SERVICE\_N* and that gives respectively the following:

---

7	TS_IRQ (ADC_INT)	Touch screen irq interrupt
---	------------------	----------------------------

and

23	GPI_1	Interrupt from the GPI_1 (SERVICE_N) pin.
----	-------	---

Using *locate* the `<mach/irqs.h>` can be located, which contains defines for the interrupt numbers.

```
#define IRQ_LPC32XX_TS_IRQ      LPC32XX_SIC1_IRQ(7)
```

```
#define IRQ_LPC32XX_GPIO_01    LPC32XX_SIC2_IRQ(1)
```

These are the defines we need to use for a correct *request\_irq*.

## Freeing IRQ

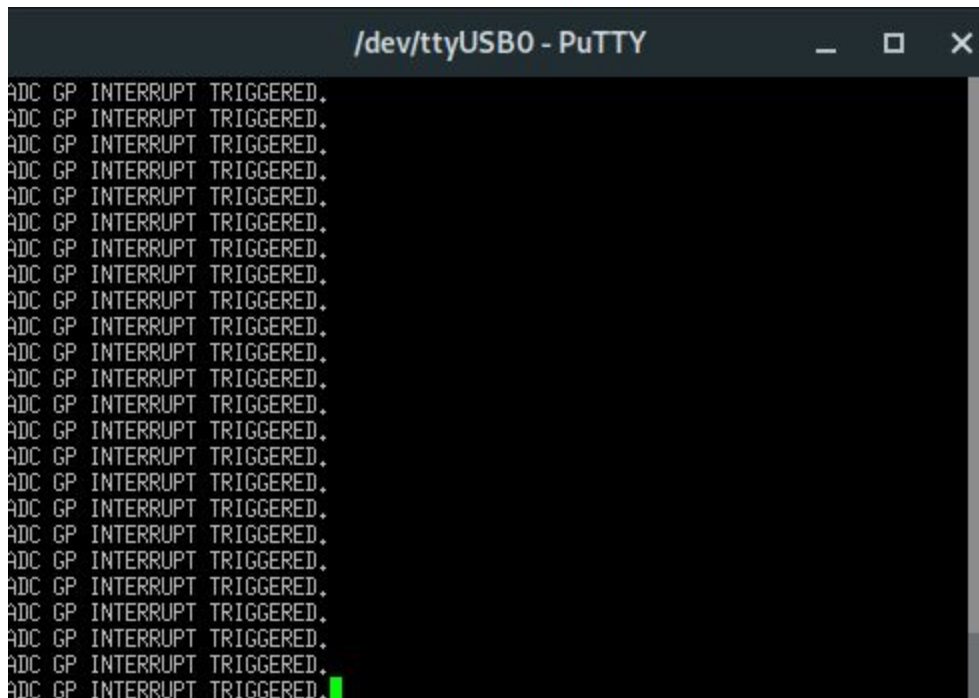
When the ADC driver unloads, the interrupt handler needs to be unregistered and the interrupt line needs to be disabled. This can be done by calling:

```
void free_irq(unsigned int irq, void *dev)
```

For *irq* we enter the same defines we found for registering an interrupt handler.

## Solve multiple interrupts on button press (level to edge triggered)

When pressing the EINT0 button we get the following output in the console. This is of course not correct, the interrupt is level triggered. In other words, interrupts are generated when the button is pressed. We want however to get an interrupt when there is a change in value, or better known as edge detection.



This can be done by setting bit 23 of the SIC2\_ATR register. The interrupt activation type determines whether each interrupt is level sensitive or edge sensitive. Setting this bit to a value of one makes the GPI\_1 work as an edge sensitive interrupt.

Proof: [https://youtu.be/43\\_8fUkHLvM](https://youtu.be/43_8fUkHLvM)

Table 65. Activation Type Register (SIC2\_ATR - 0x4001\_0010)

Bits	Name	Description	Operational value	Reset value
31	SYSCLOCK mux	Interrupt Activation Type, see bit 0 description		0
30:29	Reserved	Reserved, do not modify.	-	0
28	GPI_6	Interrupt Activation Type, see bit 0 description	user defined	0
27	GPI_5	Interrupt Activation Type, see bit 0 description	user defined	0
26	GPI_4	Interrupt Activation Type, see bit 0 description	user defined	0
25	GPI_3	Interrupt Activation Type, see bit 0 description	user defined	0
24	GPI_2	Interrupt Activation Type, see bit 0 description	user defined	0
23	GPI_1	Interrupt Activation Type, see bit 0 description	user defined	0
22	GPI_0	Interrupt Activation Type, see bit 0 description	user defined	0
21	Reserved	Reserved, do not modify.	-	0
20	SPI1_DATIN	Interrupt Activation Type, see bit 0 description		0
19	U5_RX	Interrupt Activation Type, see bit 0 description		0

## Turning on the ADC

Turning on the ADC is done by writing a 1 to bit 2 in the ADC\_CTRL register.

**Table 260. A/D Control Register (ADC\_CTRL - 0x4004 8008)**

Bits	Function	Description	Reset value
31:7	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	-
6:3	-	Internal A/D controls. Must be set to 0x0.	0x0
2	AD_PDN_CTRL (TS_ADC_PDN_CTRL)	0 = the ADC is in power down. 1 = the ADC is powered up and reset.	0
1	AD_STROBE (TS_ADC_STROBE)	Setting this bit to logic 1 will start an A/D conversion. The bit is reset by hardware when the A/D conversion has started.	0
0	-	Internal A/D control. Must be set to 0.	0

We then have to enable bit 7 in the Enable register of sub interrupt controller 1 (SIC1) to handle touchscreen(ADC) interrupts.

## Starting the ADC

Starting the ADC is done by first writing into ADC\_SELECT register which channel to read. After which the A/D conversion can be started by writing a one into the TS\_ADC\_STROBE bit of the ADC\_CTRL

register.

```
data = READ_REG(ADC_CTRL);  
data |= TS_ADC_STROBE;  
WRITE_REG (data, ADC_CTRL);
```

**Table 56. Interrupt Enable Register for Sub Interrupt Controller 1 (SIC1\_ER - 0x4000 C000)**

Bits	Name	Description	Reset value
31	USB_I2c_int	Interrupt from the USB I2C interface.	0
30	USB_dev_hp_int	USB high priority interrupt.	0
29	USB_dev_lp_int	USB low priority interrupt.	0
28	USB_dev_dma_int	USB DMA interrupt.	0
27	USB_host_int	USB host interrupt.	0
26	USB_otg_abx_int_n	External USB transceiver interrupt. Active LOW.	0
25	USB_otg_timer_int	USB timer interrupt.	0
24	SW_INT	Software interrupt (caused by bit 0 of the SW_INT register).	0
23	SPI1_INT	Interrupt from the SPI1 interface.	0
22	KEY_IRQ	Keyboard scanner interrupt.	0
21	Reserved	Reserved, do not modify.	0
20	RTC_INT	Match interrupt 0 or 1 from the RTC.	0
19	I2C_1_INT	Interrupt from the I2C1 interface. Active LOW.	0
18	I2C_2_INT	Interrupt from the I2C2 interface. Active LOW.	0
17	PLL397_INT	Lock interrupt from the 397x PLL.	0
16:15	Reserved	Reserved, do not modify.	0
14	PLLHCLK_INT	Lock interrupt from the HCLK PLL.	0
13	PLLUSB_INT	Lock interrupt from the USB PLL.	0
12	SPI2_INT	Interrupt from the SPI2 interface.	0
11:9	Reserved	Reserved, do not modify.	0
8	TS_AUX	Touch screen aux interrupt	0
7	TS_IRQ (ADC_INT)	Touch screen irq interrupt	0
6	TS_P	Touch screen pen down interrupt.	0
5	Reserved	Reserved, do not modify.	0
4	GPI_28	Interrupt from the GPI_28 pin.	0
3	Reserved	Reserved, do not modify.	0
2	JTAG_COMM_RX	RX full interrupt from the JTAG Communication Channel.	0
1	JTAG_COMM_TX	TX empty interrupt from the JTAG Communication Channel.	0
0	Reserved	Reserved, do not modify.	0

---

## Reading ADC Value

Reading the ADC value is done by simply reading the ADC\_VALUE register and applying a bitmask to read bit 9:0

**Table 261. A/D Data Register (ADC\_VALUE - 0x4004 8048)**

Bits	Function	Description	Reset value
31:10	Reserved	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	-
9:0	ADC_VALUE	The ADC value of the last conversion.	

After applying the following we can now read and print the ADC values as shown in the figure below.

```
# ADC GP INTERRUPT TRIGGERED.  
ADC(0)=584  
ADC(1)=585  
ADC(2)=747
```

---

## Reentrancy

Reentrancy can pose 2 problems for our driver:

Problems might occur if the same adc channel is activated multiple times before being ready. This could cause driver data to be accessed from multiple user contexts.

The biggest issue is the global data which is required for the operation of our driver.

```
struct state {
    int          adc_values[ADC_NUMCHANNELS];
    bool         interrupt_is_gpi;
    unsigned char adc_channel;
    struct completion completion;
    struct mutex  mlock;
};

static struct state st;
```

The first 3 variables of the struct are used for the proper operation of our driver, if this data is modified by more than 1 process at a time our logic will break down.

Finally, we have 2 points of entrancy which we need to cover:

device\_read and the gpi interrupt service routine (ISR).

In our device\_read function just before starting the adc and handling state data we use a mutex\_lock to prevent these problems. Furthermore we use the completion system to properly put our kernel driver to sleep until the required work is done.

```
mutex_lock(&st.mlock);

adc_start(channel);
wait_for_completion(&st.completion);
written = sprintf(retv_buffer, "%d", st.adc_values[st.adc_channel]);

mutex_unlock(&st.mlock);
```



---

Usually an ISR consists of 2 parts: The top-, and bottom-half.

The top is used for validating the IRQ, it is time critical since the hardware needs an acknowledge and because all interrupts are disabled since we set `IRQF_DISABLED` during the creation of the irq.

The bottom half is used for more time consuming work, it is not as time sensitive because interrupts have been acknowledged and are enabled again.

```
request_threaded_irq (IRQ_LPC32XX_GPI_01, NULL, gp_thread_interrupt, IRQF_DISABLED, "gpi", NULL) != 0)
```

The above line assigns our top- and bottom-half handler and creates a thread for the bottom half. This allows us to support reentrancy without increasing the top half ISR time. Since we do not need any validation we set our top half handler to `NULL`, resulting in a default handler which acknowledges the request and wakes our thread which executes the bottom-half handler.

In this function we use the same mutexes because we support only one entrance at a time. Again, we wait for completion because we cannot unlock the mutex until the adc conversion is done.

```
static irqreturn_t gp_thread_interrupt(int irq, void * dev_id)
{
    // Lock mainly because we cannot enter here and in our dev
    mutex_lock(&st.mlock);

    st.interrupt_is_gpi = true;
    adc_start (0);

    wait_for_completion(&st.completion); // We cant simply unlo

    mutex_unlock(&st.mlock);
    return (IRQ_HANDLED);
}
```

---

To come to this solution we have done general research on kernel reentrancy, specifically on topics such as mutexes, spinlocks, `wait_event_interruptible` and the completion system. Specifically mutexes for data safety and completion for waiting seemed the best solutions. Summarized the completion system is very suited for our needs because to paraphrase:

*"It results in readable, highly efficient code which uses waitqueues and the linux scheduler sleep/wakeup facilities"*

<https://www.kernel.org/doc/Documentation/scheduler/completion.txt>

Finally we researched Top/Bottom-half interrupt handling and the corresponding way to handle those with threads. After reading about softirq and tasklets we found the easiest and likely for simplicity sake best way to be the function: `request_threaded_irq`.

---

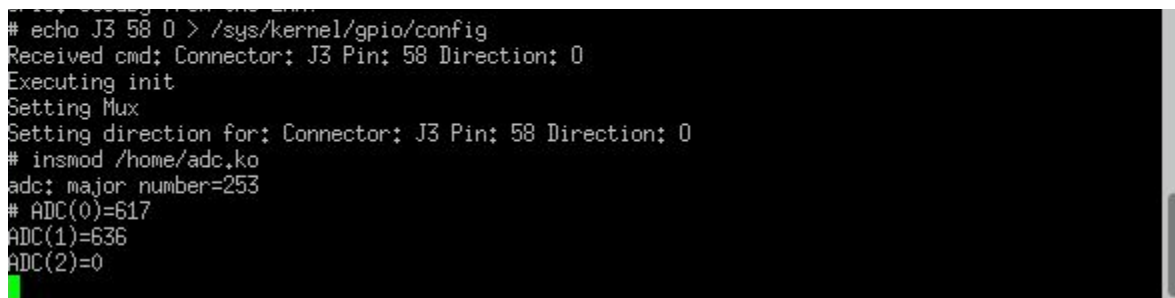
## Part 2 - Verification

### Measuring GP Interrupted Time

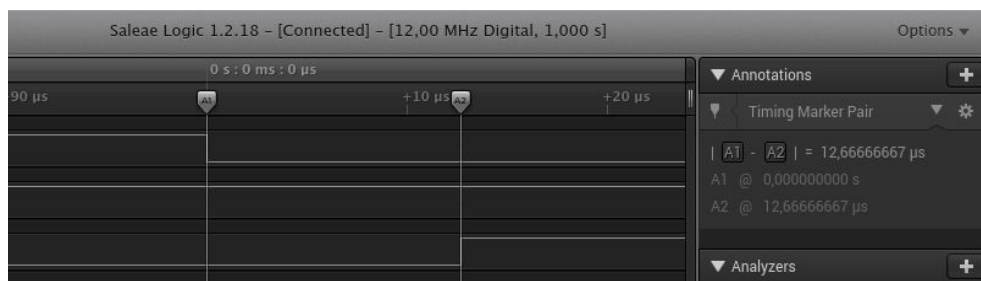
Measuring the time it took from button press to GP interrupt was 12,67 microseconds. Measuring was done by tapping on one channel of the logic analyzer the button and on another channel pin 58 of connector J3. The gpio sys driver that has been made previous assignment was used for setting the direction of the pin. In the ADC driver the pin is set to high when entering the GP interrupt handler. Done with the following code:

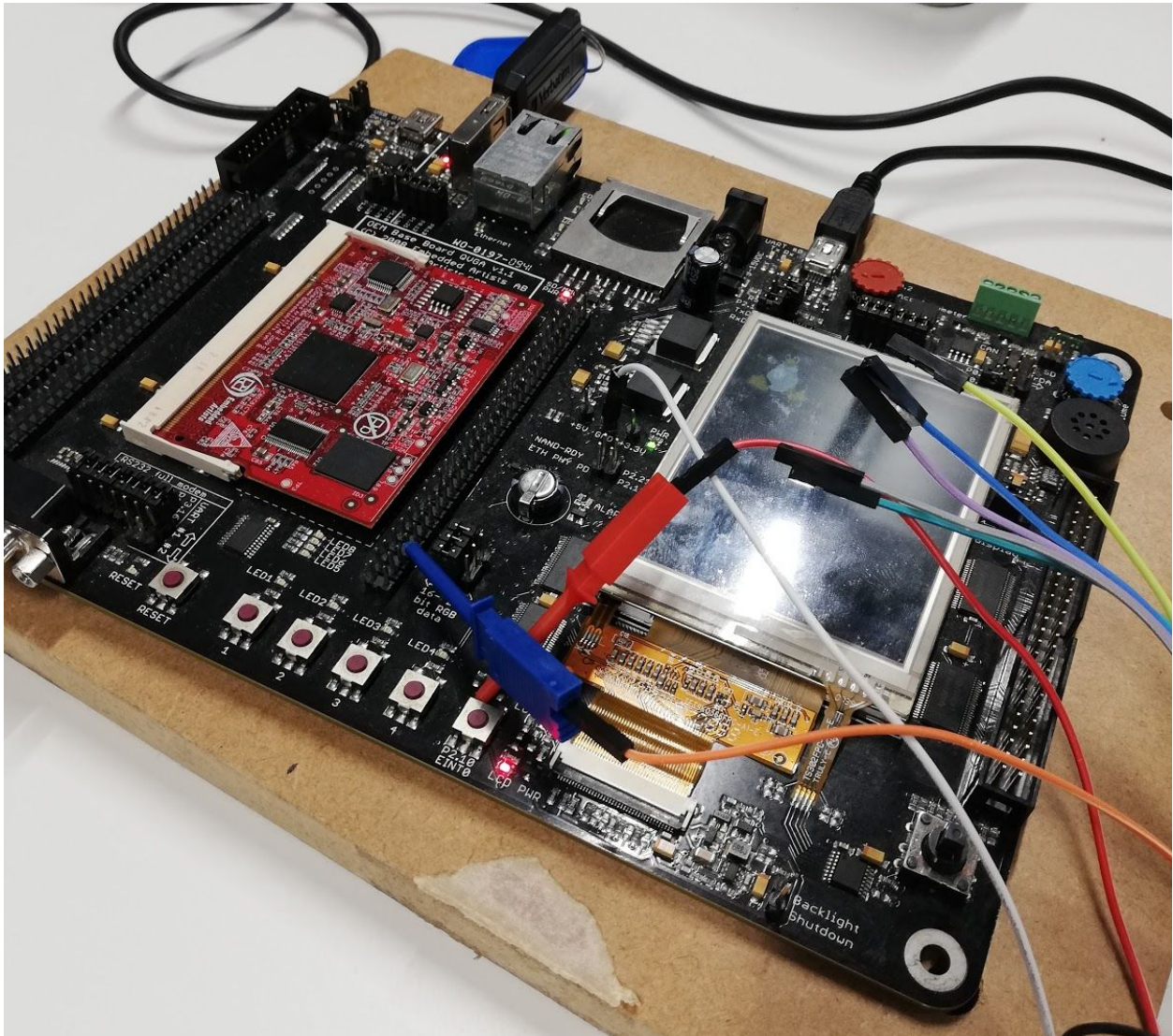
```
unsigned long* memAddr = 0;
memAddr = io_p2v(0x40028020);
*memAddr |= (1UL << 5);
```

Below is the terminal output, logic analyzer output and image of the setup.



```
# echo J3 58 0 > /sys/kernel/gpio/config
Received cmd: Connector: J3 Pin: 58 Direction: 0
Executing init
Setting Mux
Setting direction for: Connector: J3 Pin: 58 Direction: 0
# insmod /home/adc.ko
adc: major number=253
# ADC(0)=617
ADC(1)=636
ADC(2)=0
```

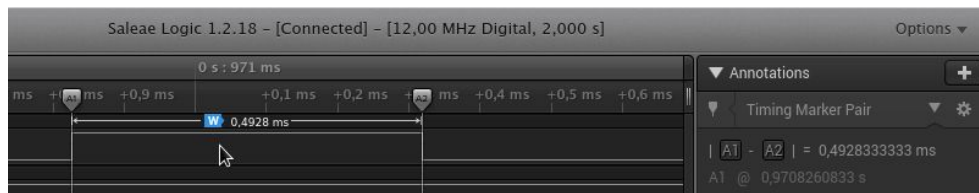




---

## Measuring A/D Conversion Time

Measuring the time it took from start of A/D conversion to ADC interrupt was almost half a millisecond. (492 microseconds) Measuring was done by having one channel of the logic analyzer connected to pin 58 of connector J3. The gpio sys driver that has been made previous assignment was used for setting the direction of the pin. In the ADC driver the pin is set to high when at the end of the `adc_start` function. After entering the ADC interrupt handler the pin was set to low again. Giving us the following measurement in *Saleae Logic* software.



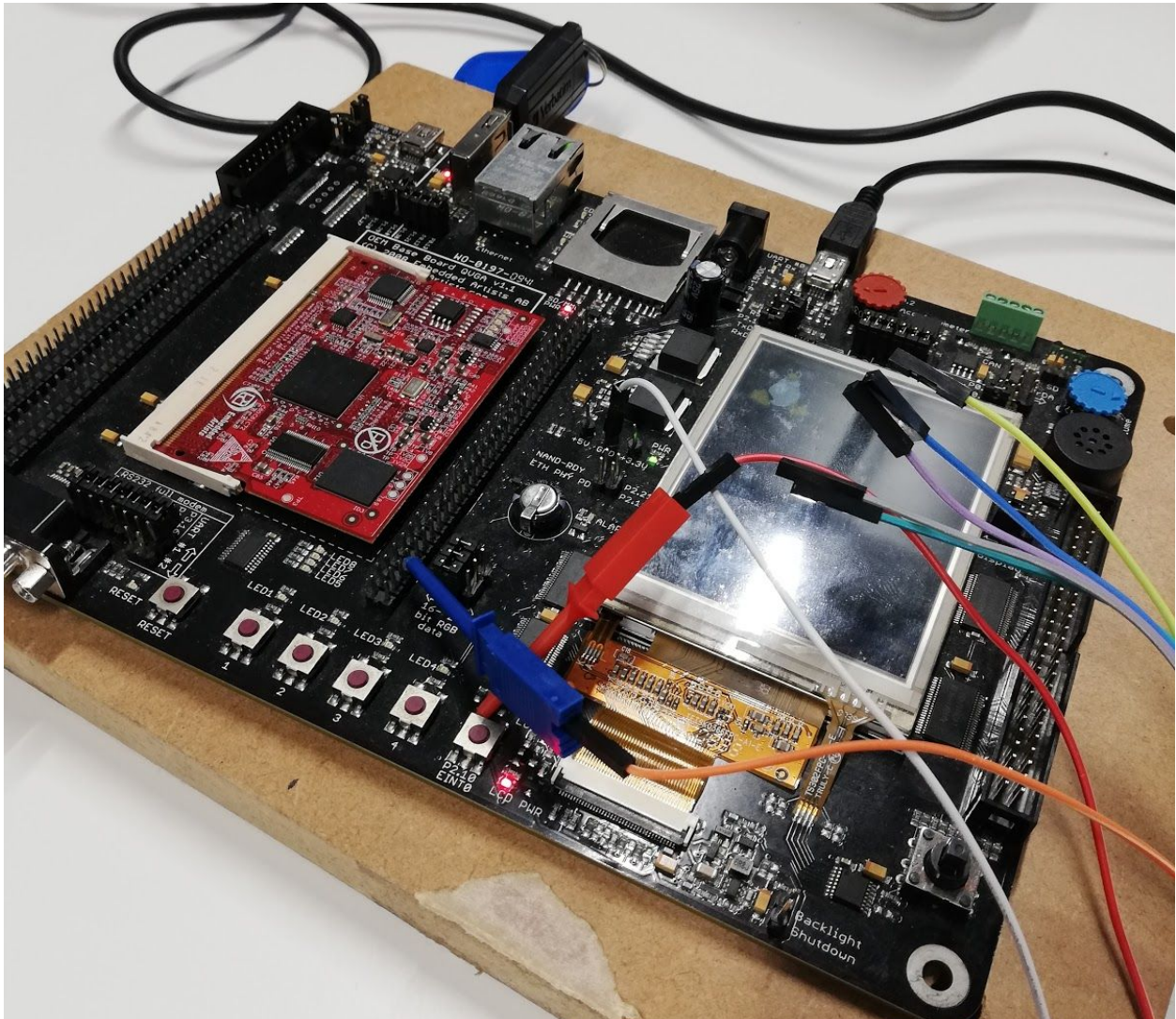
Setting the pin to high was done with the same code as the GP interrupt, setting in to low was done with the following code:

```
unsigned long* memAddr = 0;
memAddr = io_p2v(0x40028024);
*memAddr |= (1UL << 5);
```

Below the output of the terminal and an image of the setup.

```
# insmod /home/gpio-sys.ko
/sys/kernel/gpio/ created
# insmod /home/adc.ko
adc: major number=253
# echo J3 58 0 > /sys/kernel/gpio/config
Received cmd: Connector: J3 Pin: 58 Direction: 0
Executing init
Setting Mux
Setting direction for: Connector: J3 Pin: 58 Direction: 0
# cat /dev/adc0
adc:device_read(0)
ADC(0)=622
622adc:device_read(0)
adc: device_release()
#
```





---

## Part 3 - User manual

Loading the driver:

```
$ insmod adc.ko
$ mknod /dev/adc0 c 253 0
$ mknod /dev/adc1 c 253 1
$ mknod /dev/adc2 c 253 2
```

Usage:

```
$ cat /dev/adcX
```

Where x stands for the three different ADC channels.

Channel	Description
0	Accelerometer X axis
1	Accelerometer Y axis
2	Red potmeter on the LCP32xx dev board

### Reading:

Reading from one of the device nodes will return a value between 0 and 1023, this is a 10 bit nummer.

*Note: It will take a small amount of time before a value is returned due to the conversion taking up several clock cycles (11).*

### Writing:

Writing to the devices nodes will not have any effect.

### Note:

The driver is capable of handling reentrancy, any subsequent request for the adc is queued if a request is already being handled. The queue is handled in chronological order.

---

## Sources

<https://notes.shichao.io/lkd/ch7/>

<https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch05.html>

<https://github.com/torvalds/linux/blob/master/arch/arm/mach-lpc32xx/include/mach/platform.h>

[https://github.com/torvalds/linux/blob/master/drivers/iio/adc/lpc32xx\\_adc.c](https://github.com/torvalds/linux/blob/master/drivers/iio/adc/lpc32xx_adc.c)

<https://renenyffenegger.ch/notes/Linux/kernel/locks/mutex>

<https://www.kernel.org/doc/Documentation/scheduler/completion.txt>

[https://manpages.debian.org/jessie/linux-manual-3.16/wait\\_event\\_interruptible.9.en.html](https://manpages.debian.org/jessie/linux-manual-3.16/wait_event_interruptible.9.en.html)

<http://rahulonblog.blogspot.com/2014/03/why-to-use-requestthreadedirq.html>

<https://lwn.net/Articles/302043/>

<https://www.kernel.org/doc/html/docs/kernel-api/API-request-threaded-irq.html>

<https://kernel.readthedocs.io/en/sphinx-samples/kernel-locking.html>

<https://stackoverflow.com/questions/35011322/why-not-to-use-mutex-inside-an-interrupt>

*LPC32x0 User manual*

*LPC32x0 Base board design*