ECE381 – Lab 5 – Inter-Integrated Circuit (I^2C) Jolt Detector

Introduction:

Learn the I2C bus protocol and programming by making a jolt detector. Our CY8CKIT-044 has a few I2C components prewired. There is a non-volatile, ferroelectric memory (FM24V10) and also a 3-axis accelerometer (KXTJ2). This lab will utilize both to make a jolt detector. An acceleration threshold will be set on each of the three axes on the KXTJ2. When any of the acceleration thresholds is reached in either positive or negative direction, the KXTJ2 will trigger an interrupt. Once interrupted, the PSoC will read which of the 3-axes reached the threshold (stored in a byte in a register on the KXTJ2) and store this into the F-RAM. On a button press, the F-RAM will use the 3-color LED to replay any reached thresholds, mapping each LED to an axis. The lab will also introduce power saving, since the end goal will be a battery powered device to see if something was potentially damaged during transit.

Hardware Configuration:

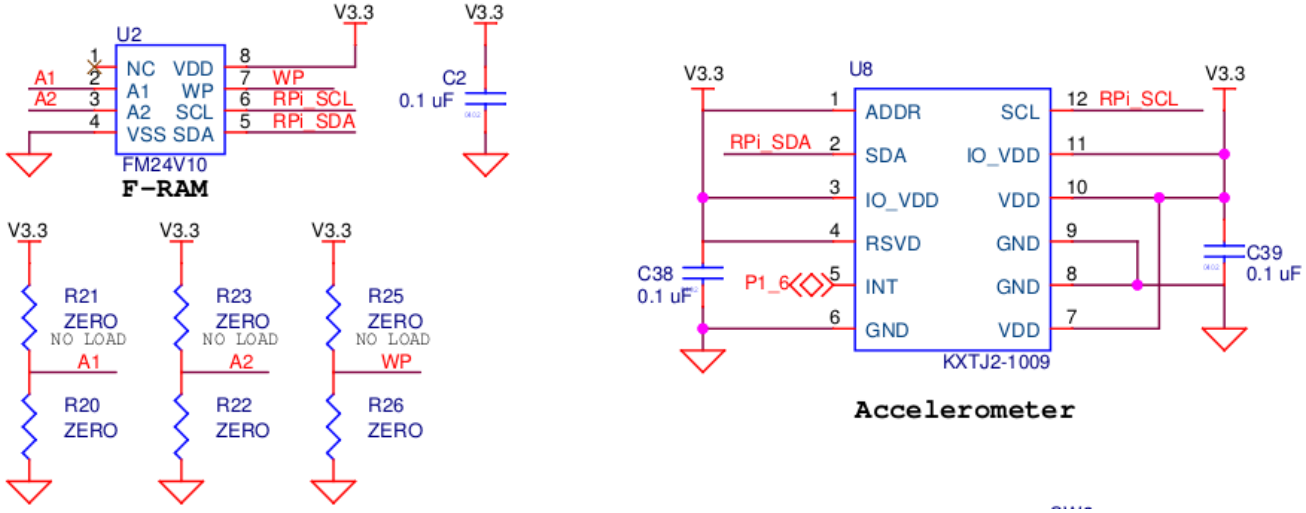


Figure 1: I2C Devices on CY8CKIT-044

Both the F-RAM and accelerometer are already soldered onto the CY8CKIT-044. The F-RAM is a memory device, so it has an I2C Group 1 address of 1010 (binary). The Cypress/Infineon engineers have already wired the two address pins to ground, thus giving the F-RAM an I2C address of 0x50 for using the lower 65536 bytes of storage, and 0x51 for accessing the upper 65536 bytes of storage. The accelerometer has an I2C address of 000111X, where X is the ADDR pin. As shown in Figure 1, the Cypress/Infineon engineers have wired this pin to VDD, so you can communicate with the accelerometer using I2C address 0x0F. The X, Y, and Z axes of the accelerometer are labeled on the

CY8CKIT-044 board in the lower right corner

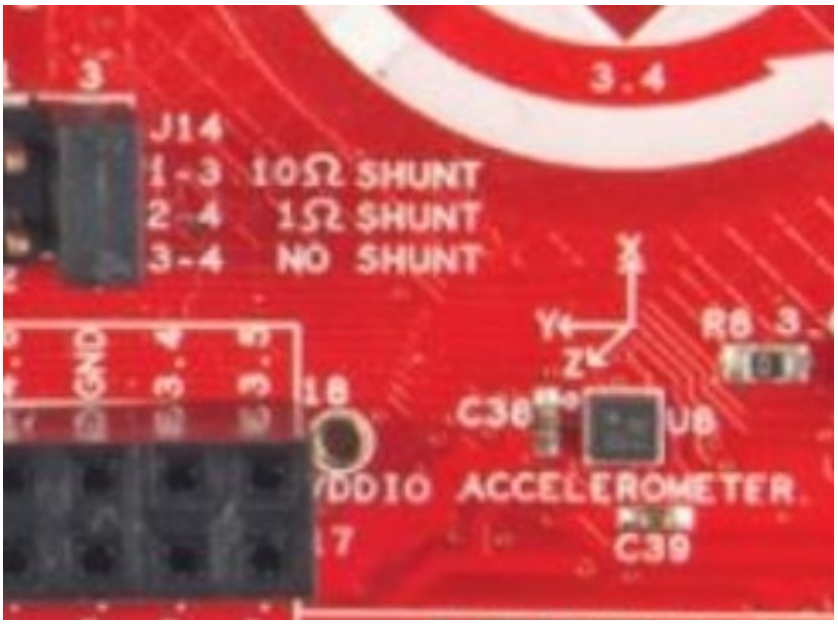


Figure 2: Accelerometer Position on CY8CKIT-044

The Y direction is parallel to the USB port, the X is perpendicular, and the Z axis goes through the board. Basically, if the CY8CKIT-044 is resting face-up, with the headers facing up, on the padded standoffs on a desk, it would read 0 in X, 0 in Y, and about +1g in Z. The accelerometer can be configured in a "wakeup" mode, whereby it mostly sits idle to save power. If significant motion is detected (acceleration higher than a specified g-force threshold), however, the accelerometer will wake up and cause an interrupt (low-to-high pulse) on the INT pin.

Low Power Modes

It turns out that quite often your microcontroller just sits around waiting for events to happen. Thus far, we have pretty much just done some kind of polling/infinite loop while waiting for events (polling a button, checking a flag, waiting for the UART to return non-zero, etc.). Unfortunately while doing so, we are basically wasting a bunch of CPU cycles, and thus power, while waiting. This hasn't mattered since we are using USB with an assumed infinite power source. But, quite often, batteries power microcontrollers, and hence burning power while waiting reduces their lifetime considerably. What we want is for the microcontroller to only really power the necessary systems to detect our event, turning the rest off to conserve energy.

Fortunately for us, the PSoC does allow us to come close to this goal. The PSoC4 does have quite a few low power modes available to us, each with various power saving/functionality tradeoffs. They range from the least power saving but most easily woken (Sleep Mode) all the way down to the most saving, but can only be woken by external reset (Stop Mode). See the tables below:

Table 1. Power Mode Specs

Power Mode	Current Range (typical) (V_{DD} = 3.3 V to 5.0 V)	Wakeup Time			
		PSoC 4100/4200	PSoC 4100M/4200M	PSoC 4200L	PSoC 4100PS
Active	1.3 mA to 14 mA	–	–	–	–
Sleep	1.0 mA to 3 mA	0	0	0	0
Deep Sleep	1.3 µA to 15 µA	25 µs	25 µs	25 µs	35 µs
Hibernate	150 nA to 1 µA	2 ms	0.7 ms	0.7 ms	Not Applicable
Stop	20 nA to 80 nA	2 ms	2 ms	1.9 ms	Not Applicable

and

Table 2. PSoC Power Modes—Resources Availability

Subsystem	Active	Sleep	Deep Sleep	Hibernate	Stop
CPU	ON	Retention[1]	Retention	OFF	OFF
SRAM	ON	ON	Retention	Retention	OFF
High-speed peripherals (SPI, UART, and so on)	ON	ON	Retention	OFF	OFF
Universal digital blocks (UDBs)	ON	ON	Retention[2]	OFF[3]	OFF
VDAC	ON	ON	Retention[2]	OFF	OFF
SPI slave and I^2C slave (SCB based)	ON	ON	ON	OFF	OFF
High-speed clock (IMO, ECO, and PLLs)	ON	ON	OFF	OFF	OFF
Low-speed clock (32 kHz) (ILO and WCO)	ON	ON	ON	OFF	OFF
Brown-out detection	ON	ON	ON	ON	OFF
CTBs/CTBms (opamp and comparators)	ON	ON	ON[4]	OFF	OFF
ADC	ON	ON	OFF	OFF	OFF
Low-power comparators	ON	ON	ON	ON	OFF
GPIO (output state)	ON	ON	ON	ON	Frozen[5]

Our use thus far has always been in Active mode with everything on, but as you can see from Table 1 other modes can offer significant savings. If all you are doing is waiting for a button (Hibernate) or having the WDT wake you (Deep Sleep) that's a few mA of current saved and thus for a standard AA sized alkaline with a lifetime around 2500 mAh can mean the difference between lasting a week (Active) and lasting over a year (Hibernate/Deep Sleep). The big tradeoff with power saving modes is that you save energy by going into a sleep mode, but you also require extra time to "wake up", as the various subsystems need to be re-powered and state restored (if the power saving mode supports it). Since most wakeup sources are interrupts, this means added latency to those interrupts, which may need to be considered if you have hard real-time targets. For example, waking up from Deep Sleep takes 25µs, which is considerably longer than the 12-24 clock cycles of latency in active mode.

The API call to enter a sleep mode is:

`CySysPmSleepMode()`

and the PSoC is assumed to enter that mode upon completion of the call. For example, calling `CySysPmDeepSleep()` would put the PSoC in DeepSleep mode. The PSoC will resume execution on the next line of code after a wakeup event, and will be operating in Active mode after the wakeup time has elapsed. Since most of the wakeup events are interrupts, more often than not, execution really resumes in that respective ISR. For programming with power saving, the `CySysPmSleepMode()` call would typically be at the start or end of your infinite loop. If it's at the start, then this means the PSoC goes to sleep waiting for the event(s). When an event wakes up, you either have all the code for that event in the ISR, or you set a flag in the ISR and check for that flag immediately after the `CySysPmSleepMode()` call in the infinite loop. If all the code is in the ISR, you would wake up, execute the ISR, then return from the ISR to the main infinite loop. In this case, the `CySysPmSleepMode()` call would likely be the only thing in the infinite loop, so you would just loop back around to the `CySysPmSleepMode()` call, which would put you back to sleep to wait for the next event! If you use a flag, the pattern is similar, just that after executing the code you want to execute for that event, you clear the flag and loop back around to the `CySysPmSleepMode()` call to go back to sleep. If your wakeup event sets a flag in the ISR, then on this event, you wakeup, set this flag, then immediately check it (and other flags) to take the appropriate action after the `CySysPmSleepMode()` call.

PSoC Configuration:

Create a new PSoC project/workspace.

- Place a Communications/I2C/I2C (SCB mode) module in the main schematic window. Right-click to configure, and set the mode to "Master", the data rate to "100 kbps" and click OK. Under pins, map I2C:scl to P4[0] and I2C:sda to P4[1].
- Place three Ports-and-Pins/Digital Output Pins, and name them for the Red, Green, and Blue LEDs and map them to P0[6], P2[6], and P6[5] respectively.
- Place a Ports-and-Pins/Digital Input Pin. Name it "ACCEL_INT". The mode should be "High Impedance Digital" since it is driven by the accelerometer. Configure it to interrupt on a rising edge. Map it to P1[6], as Figure 1 shows the accelerometer's physical interrupt pin connected there.
- Place a Ports-and-Pins/Digital Input Pin. Name it "TEST". If using P0[7] (recommended), put it in pull-up mode with an interrupt on a falling edge. If using another button, pick the correct pull-down/pull-up and edge for it.

Software

- On initialization, start the I2C module, then configure the accelerometer as below:
    - Set the wakeup function Output Data Rate to 12.5 or 6.25 Hz
    - Configure the accelerometer to create an active high interrupt, on the physical interrupt pin, that latches until INT_REL is read
    - Enable interrupts from all six possible directions (+/- x, +/- y, +/- z)
    - Set the output data rate so that the LPF roll-off matches your wakeup Output Data Rate
    - Set the wakeup timer to 10 counts for the threshold. NOTE: You might need to adjust this later for better operation!!!
    - Set the wakeup threshold to be 3g using Equation 1 on page 30 of the KXTJ2 datasheet

    - Finally, use CTRL_REG1 to put the accelerometer in operating mode, using 12-bit sampling resolution, with a range of +/-4g or +/- 8g, with DRDYE disabled (we only want to interrupt on threshold events) and WUFE enabled (so we can).
    - Wait for the appropriate amount of startup time, specified by Table 4 on pg. 7 of the KXTJ2 datasheet, for your chosen Output Data Rate
- The two ISRs should then be started (after the I2C setup to avoid any races), and they should just set flag variables that are checked in the infinite loop. This is due to the fact that the I2C module itself requires an interrupt, so ***YOU CAN'T DO I2C IN AN ISR WITHOUT MODIFYING INTERRUPT PRIORITIES!*** Using flag variables gets around this by keeping all I2C in the easily interruptible main loop.
- The first line of your main, infinite loop should be the `CySysPmDeepSleep()` call. We are using DeepSleep since both ACCEL_INT and TEST are GPIO interrupts, DeepSleep will save the most power while still retaining CPU/SRAM/Configuration Register state (see Table 2 above)
- The next lines should check for the respective flags from either the ACCEL interrupt or the TEST interrupt.
    - If TEST triggered the wakeup:
        - Make sure all 3 LEDs are off.
        - Enter a loop that keeps time on how long the user is holding the button (a simple counter and fixed delay will work here)
            - If the user holds the button for more than 5 seconds, turn on all 3 LEDs to make the multi-color LED white, and keep it white for as long as the user holds the button. This is to indicate that the user wishes to reset all stored "jolts" in the F-RAM.
                - Zeroing the F-RAM means storing the byte 0x00 in all 65536 word locations in F-RAM. **DO NOT DO THIS ONE BYTE AT A TIME, IT WILL TAKE TOO LONG!!!** Fortunately, the F-RAM will automatically increment the address internally until it sees a STOP bit on I2C, so you can write lots of zero bytes without having to constantly resend the target address.
                - You should play back some sequence of colors on the LEDs while the F-RAM is zeroing to show to the user that the F-RAM is being zeroed. The easiest way would be to just change the color every *N* bytes of zeros that were sent, as long as *N* is big enough to actually see it change.
            - If the user doesn't hold the button for more than 5 seconds, start at the first word address in F-RAM and read the byte stored there. If the byte is not zero, illuminate the appropriate LED(s) based on the stored value. For example, if the recorded byte has bits 5 or 4 set, that means the stored jolt happened in the x-direction, so illuminate the RED LED. If it has its 3 or 2 set, the jolt was in the y-direction, so illuminate the GREEN LED. If bits 1 or 0 are set, illuminate the BLUE LED. Keep the LED(s) on for

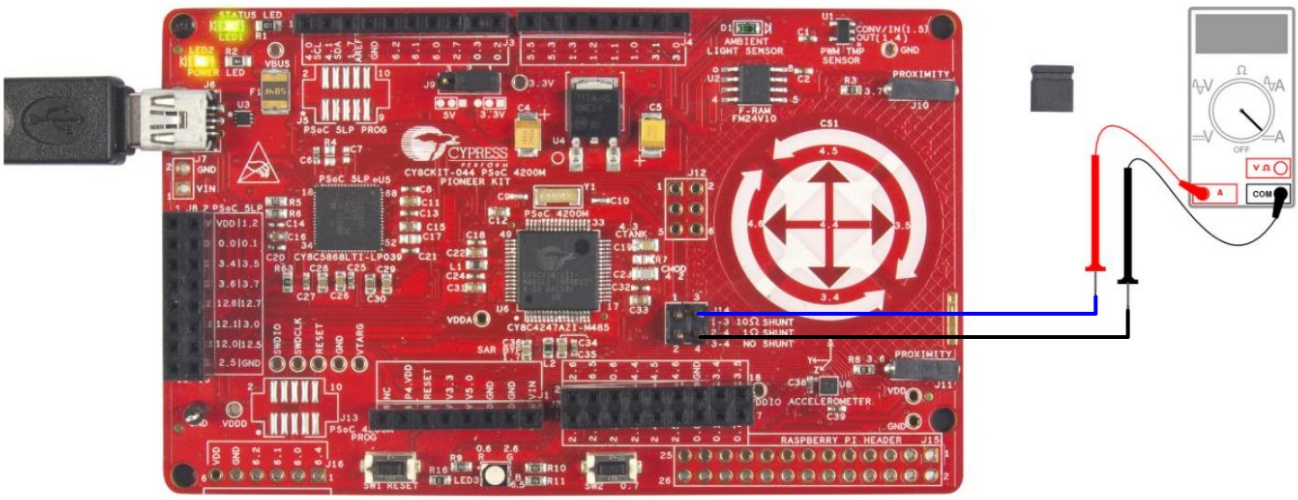
a long enough time so that the user can actually see what happened, then turn it off for the same amount of time
                - Keep doing this LED playback until the byte you read from the F-RAM is 0, indicating there are no more jolts stored
            - Reset the test flag variable to zero, and you should loop back around to `CySysPmDeepSleep()` to go back to sleep
    - If ACCEL caused the wakeup:
        - Read the INT_SOURCE2 register on the accelerometer. This register contains the information on which axis(axes) went beyond the threshold. Bits 5&4 are X-negative, X-positive, 3&2 are Y-negative, Y-positive, and Bits 1&0 are Z-negative, Z-positive, meaning a g-force beyond the value specified by the WAKEUP_THRESHOLD occurred in one (or more) of those directions
        - Start at the first word address (0x0000) of F-RAM, and read the data until the next non-zero byte is found. Store the value from INT_SRC into this location
        - Read the INT_REL register on the accelerometer. You don't have to care about the actual value read, the physical interrupt pin on P1[6] won't go low again until you read it, so we have to to get future interrupts from the accelerometer
        - Reset the accelerometer flag variable to zero, and you should loop back around to `CySysPmDeepSleep()` to go back to sleep

Programming Tips

- The F-RAM has an internal register that keeps the current word address of the F-RAM (NOT the I2C address, this is the fixed bus address). Thus, any read from a specified word address requires an I2C write of 2 bytes first to set this address (last bit of first byte on the I2C bus is 0 → Word Address) followed by an I2C read (last bit of first byte on the I2C bus is 1). The internal address automatically increments for each byte write or read.
- From the PSoC Example Code, the sequence to write is as follows:
    - `I2C_I2CMasterWriteBuf(`*`I2C_Address, pointer to buffer that has data, # of bytes to write, I2C_Mode);`*
    - `while(I2C_I2CMasterStatus() & I2C_I2C_MSTAT_WR_CMPLT);`
    - `I2C_I2CMasterClearStatus();`
- Using this sequence:
    - The *I2C_Mode* defines how the master handles ACKs and generate STOPs. See `I2CMasterWriteBuf` in the I2C module datasheet. `I2C_I2C_MODE_COMPLETE_XFER` is most common as it generates both the START and STOP bits. `I2C_I2C_MODE_NO_STOP` is most common as it generates start but no stop bit, and `I2C_I2C_MODE_REPEAT_START` repeats the start bit and also does the stop.
    - The while loop waits for the I2C module to write # of bytes. Since the I2C module is much slower than the CPU clock, this requires us to wait by polling the master status register until it indicates that the module is done sending data.
    - The `I2CMasterClearStatus()` then clears all bits in the I2C status register so we don't think the next write we do instantly completes, it will have to set the WR complete flag again.

- The I2C read sequence is similar:
  - `I2C_I2CMasterReadBuf(`*`I2C_Address, pointer to buffer that will store the data read in, # of bytes to read, I2C_Mode`*`)`
  - `while (!(I2C_I2CMasterStatus() & I2C_I2C_MSTAT_RD_CMPLT));`
  - `I2C_I2CMasterClearStatus();`
- The PSoC automatically shifts the *I2C_Address* in either function to the left by one bit and appends the 0 or 1 for the _Write or _Read functions accordingly.
- The first two bytes of any I2C Write to the F-RAM sets the internal F-RAM word address! Don't forget this when reading/writing!
- If no events take place no matter how hard you shake it, you might need to reduce the WAKEUP_THRESHOLD and WAKEUP_TIMER to something lower for testing.
- You are to measure the current consumption during operation and document it (phone picture of multi-meter is fine). The best way to do this is to remove the jumper on header J14 that normally shorts PSoC4 VDD to the board VDD. Place an ammeter in series on those pins (see figure below) and it will tell you the current consumed by the PSoC.

Figure A-9. Current Measurement When Powered From USB Connector



- In the report, you should estimate how long a standard AA 2500 mAh battery would last while powering your jolt detector. Unless it's jolted a whole bunch of times, the amount it consumes in DeepSleep should be pretty close to how much is consumed in total (the amount during active mode where it stores the log should be pretty minimal over a 24 hour period).