**EE414 Embedded System**

**20180745 – Tung-Duong Mai**

**Lab 4. Interrupt**

1. Purpose:

Understand how to program the interrupt and timer via signal handler on the AM3359 processor in Beaglebone with Linux.

The application program will work with interrupt from high-resolution timer. The main program waits for the user input (tempo, time-signature, start/stop) with menus given in the Lab 3. When the run state become 1, it should arm the high resolution timer to activate the signal handler. When the run state become 0, it should dis-arm the highresolution timer and deactivate the signal handler.

The application gets user commands – tempo, time-signature, start, and stop - from the PC keyboard via USB serial cable.

Target board: Beaglebone (containing 4 GPIO LEDs) with Linux.

Host computer: PC with Linux, cross compiler, NFS, and minicom.

1. Experiment sequence:
2. Setup cross development environment:
3. Connection

Connect Beaglebone with Ethernet cable and Power adapter.

Connect Beaglebone with Linux PC by USB cable.

By this, the PC can access the Beaglebone via the network or USB.

1. Start PC NFS server:

After this step, PC can act as NFS server and it waits for Beaglebone to mount as an NFS client.

1. Connect to Beagle bone:

Using minicom or ssh

I prefer ssh connection method since it is more reliable (I was worried that the stale lock file might happen again) and more intuitive (I can execute directly from the PC terminal without switching to minicom interface). The result I showed is from ssh connection.

However, for complete experiment and item 1 in the discussion, I also retest all experiments with minicom.

1. Start Beaglebone nfs client

I switch to root user to have enough permission to run sysfs command (I tried give gpio permission to john (my user) but there is some incompatibility, so I think it is the best to just run in root)

Now, the Beaglebone can access the directory on PC that we setup in the previous lab. We are ready for cross development.

NFS or scp both work correctly. However, I think NFS is more intuitive since it gives me the sentiment that the data are shared and I can work as if the data is locally stored (for scp, I have to explicitly transfer the data).

1. Test timer\_hrt.c
2. Detect if timer system supports high resolution:

Purpose: Since this lab use high resolution timer interrupt to improve the quality of the metronome, we must first check that the system support the high resolution timer.

Methods: We experiment 2 methods to check the timer’s resolution

a. Kernel startup message

b. Examine proc file system

(2) Check test\_hrt.c

Purpose: We check test\_hrt.c to see how the program control the timer interrupt.

Code skimming and document reading: The program makes a decrementor (backward counter down to 0) that counts starting after 1s and count every 1s.

Familiarize with the interface: To get familiar with the code interface, I also tried with different interval, starting time and starting number.

Code comprehension: Then, I dissect the program to understand the code:

* Init HR timer: The timerspec is used to store the timer specification, including start time and interval. Then timer\_settime and timer\_settime is called to create and set the timer.

int i = 0;

timer\_t t\_id;

struct itimerspec tim\_spec = {.it\_interval= {.tv\_sec=1,.tv\_nsec=10000},

.it\_value = {.tv\_sec=1,.tv\_nsec=10000}};

if (timer\_create(CLOCK\_MONOTONIC, NULL, &t\_id))

perror("timer\_create");

if (timer\_settime(t\_id, 0, &tim\_spec, NULL))

perror("timer\_settime");

* Create signal handler: The most impartant job is updating the prev parameter, such that it store the current time. Depends on application we have different tasks, such as print out the time different and count down in this example.

struct timespec now;

clock\_gettime(CLOCK\_MONOTONIC, &now);

printf("[%d]Diff time:%lf\n", count, timerdiff(&now, &prev));

prev = now;

count --;

* Setup the handler: We use a *sigaction* to store the signal handler the signal set (stored in a *sigset\_t*) (including SIGALRM, which will do the timer interrupt).

Then, the sigaction function will bind the handler (action) to the signal.

struct sigaction act;

sigset\_t set;

sigemptyset( &set );

sigaddset( &set, SIGALRM );

act.sa\_flags = 0;

act.sa\_mask = set;

act.sa\_handler = &handler;

sigaction( SIGALRM, &act, NULL );

1. Test the algorithm of Metronome\_hrt.c on PC:
2. Necessary functionality:

* Mode manipulation by keyboard: Using non-blocking mode as in lab 3.
* LED manipulation by mmap: Refine the code Metronome\_tui\_thread.c in lab 3.
* Main function: Setting necessary parameters, setting the interrupt and setting
* Interrupt handler function: Handle the interrupt (timer interrupt) by appropriate function and update interrupt parameters. This is the link between main program and LED manipulation.

Note that Ctrl+C interrupt is already handled in lab 2-3.

1. Design choice:

I design 4 (5 with PC simulation) code files:

* Metronome\_hrt.c for main and interrupt handler
* Gpio\_led\_fu.c / gpio\_led\_fu\_sim for LED manipulation (real and simulation)
* Key\_input\_fu for keyboard mode manipulation
* userLEDmmap.h for interfacing between gpio and main

1. Build the algorithm:

Three main-parts

1. Initialization: We need to initialize these parameters

Init GPIO LED

Init HR timer

Make signal handler for HR timer

Init key processing - Set termios - Print title and menu – metronome processing

Set default values to parameters (TimeSig 3 (3/4) , Tempo 90, Stop) and print default values

2. Loop:

- Wait for key press and update the mode depending on input key. Arm or disarm the timer when appropriate.

- Print single line message: Input & Status (Without linefeed)

3. Cleanup:

- Print quit message

- Reset termios

1. Code design:

Global variables: We need to share multiple information between gpio thread and main thread. Therefore, I bring many shared variables outside to be global variables. By this, we could share information between threads. These are defined in the userLEDmmap.h, hence available in both gpio and main file.

Clean-up: I deleted every thread-related code because we will use interrupt.

GPIO refinement: Since now we use interrupt, we dont need the while loop for gpio anymore. However, we still want to only map and mmap once at the beginning and the end. Therefore, I partition the gpio code into 3 parts: gpio\_map, gpio\_play (main) and gpio\_unmap. For the simulation, gpio\_map and gpio\_unmap does nothing. The loop is eliminated since interrupt will do the timing. The loop’s inside body code is utilized in the play (main) function, but we delete all the sleep function.

Signal Handler: The handler is based on the studied test\_hrt.c in part 1. Instead of printing the time difference. Now we will call gpio\_play.

Setting the HR timer is done similarly with test\_hrt\_timer.c. and key processing is the same with previous lab’s code. The timer is stop (disarmed) if we stop and armed if we start.

Disarm and arm is done by timer\_settime. Disarm is when we pass 0 interval as new timespec. Else, it will update and start the timer.

1. Difficulty & Debug:

* Now we cannot set the active and inactive at once (we do nto use sleep). Instead, we need to turn on and off the LED in two different interrupts.

Therefore, now I turn on if the index is even, and turn off otherwise (odd).

* Interrupted system call error: Since the read function might be called while in interrupt handler, it could generate this error. We add codes in getch() to catch this error and handle (ignore) it instead of printing the error (EINTR).
* We want the metronome to restart if we change mode while running (Piazza TA’s answer). Therefore, instead of just update the timer when we press ‘m’, we will update it for ‘c’, ‘b’, ‘z’ as well.
* We also need to make sure that we start at the first beat of the new state after the state (time signature/tempo/on-off) is changed. I do that by using a (wrap-around) counter and reset it to 0 (first beat) when state is changed.

1. Experiment:

Except GPIO, most of the code can be test with PC. Hence, we use the simulation to debug on PC.

i. Debug on PC

This is the early debug for our code, since we would like to make sure that everything in the input handling is correct before moving on the Beaglebone control

In gpio\_led\_fu\_sim.c, I just make gpio\_map and gpio\_unmap do nothing, and gpio\_play will only print the message (no gpio manipulation)

The program simply add the output of character corresponding to the LED ( ‘7’ means 3 LEDs, ‘3’ means 2 LEDs, and ‘1’ means 1 LED. After this step, we confirm that the algorithm could pass the desired correct LEDs status to the Beaglebones and the Beaglebones now can realize the LED status.

ii. Test on Beaglebones

Finally, we write the code to manipulate GPIO and realize the LED on/off status.

Before and after experiment, I reuse the shell file from last lab to turn off all LEDs and restore LEDs.

In gpio\_led\_fu\_sim.c, I just implement the gpio mapping, unmapping and gpio manipulation (as we did in lab 2-3) and make it gpio\_led\_fu.c

1. Experimental results:

Note: To produce all the executable files, simply use make command is enough.

1. Test timer\_hrt.c

(1) Detect if your timer system supports high resolution

a. Kernel startup message

Command: dmesg | grep resolution

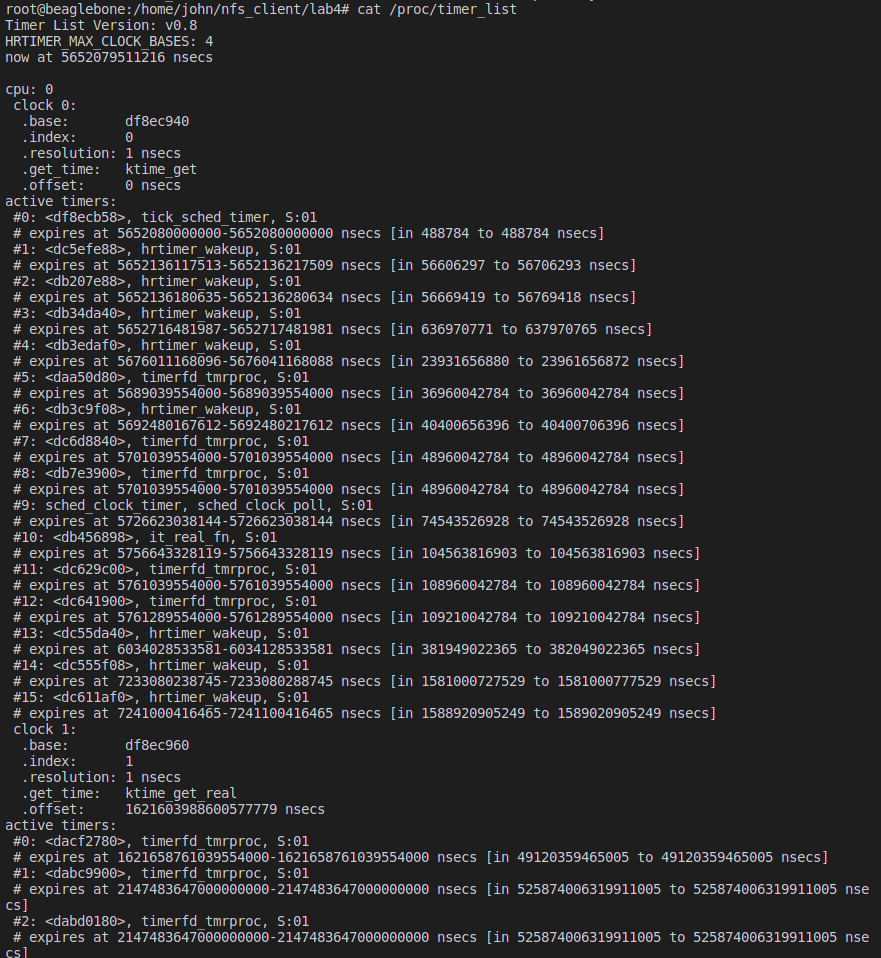
Result: 41ns

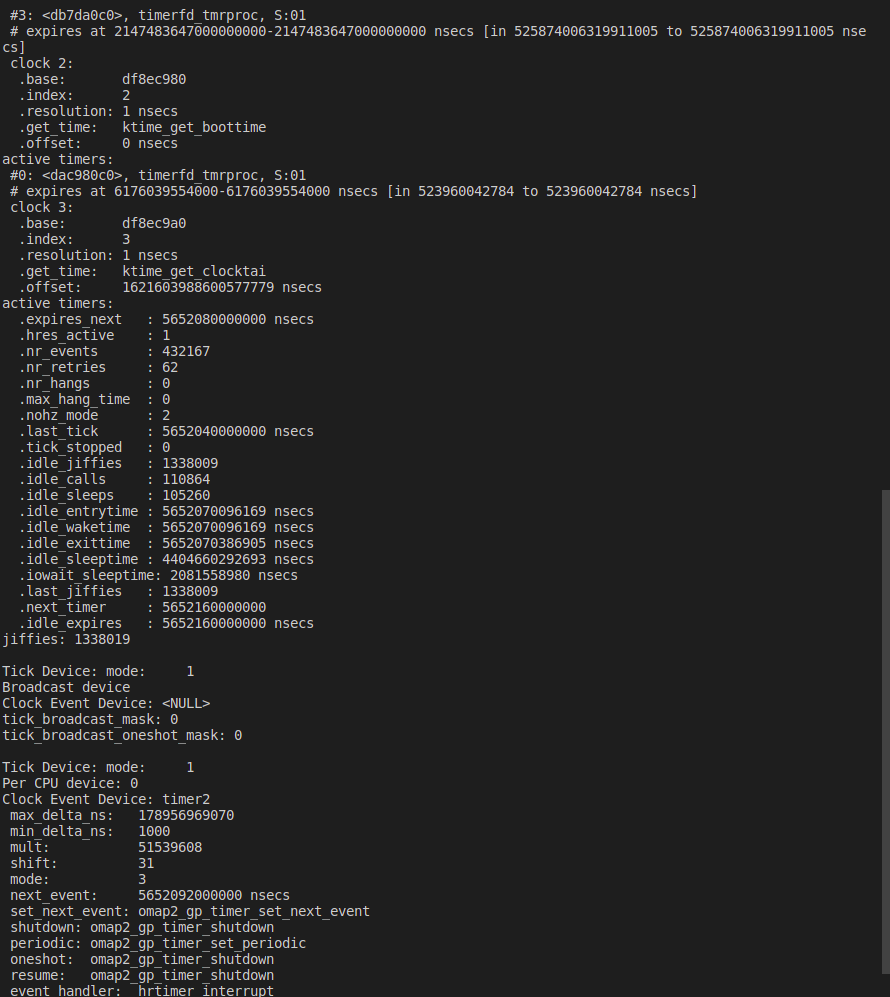
https://lh3.googleusercontent.com/R4A_atX1Wn_hXwrwZEI3Sw-u4j7BxZ_PHuDQSn7P5lqIL6VRb9s3UcEFbu-d_6_3FmETjf6QEmejwmYpVxE1xcBM_7oasBeGblvq4u4C5oUyvDCtv77HcB6nsLhDN5lHV6fn1gce

b. Examine proc file system

command: cat /proc/timer\_list

Result: 1ns





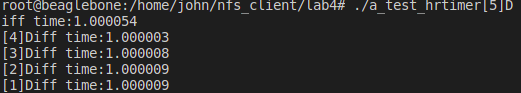
Conclusion: Our system includes support for high resolution timers, since there are timers with accuracy better than 1 jiffy

(2) Test test\_hrtimer.c

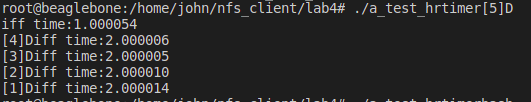
Command: make time

Or: arm-linux-gnueabihf-gcc -o a\_test\_hrtimer test\_hrtimer.c -lrt

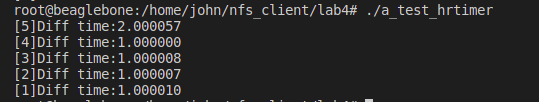
Interval 1s, Start time 1s, count from 5:



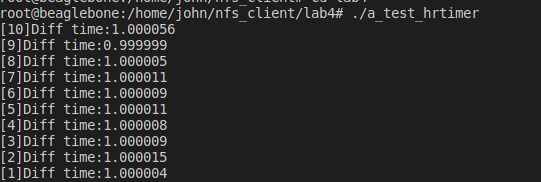
Change interval 2s:



Change start time to 2s:



Count from 10:

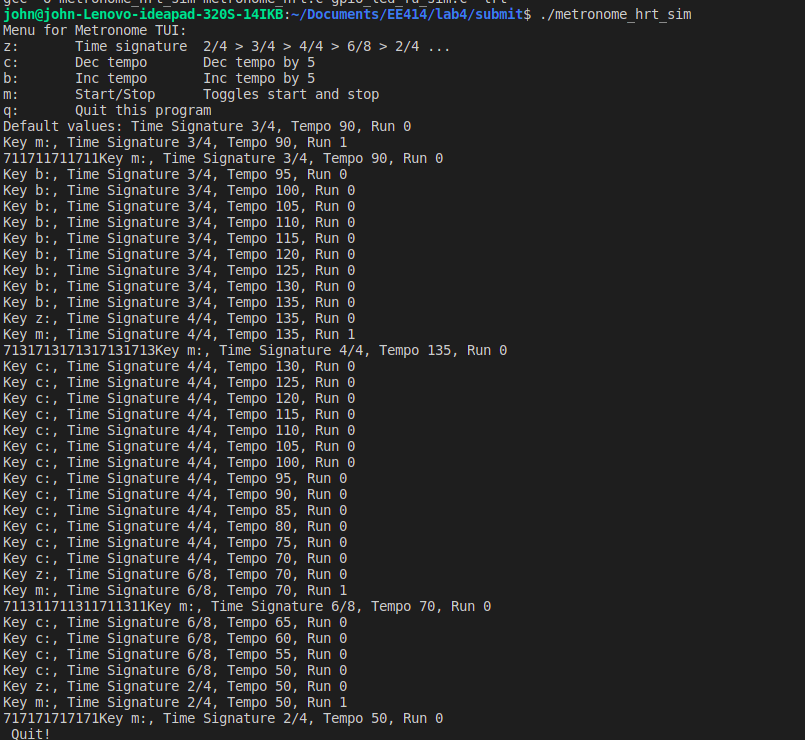
s

2. Test metronome\_hrt.c

3.1. Simulation on PC

Command: make sim

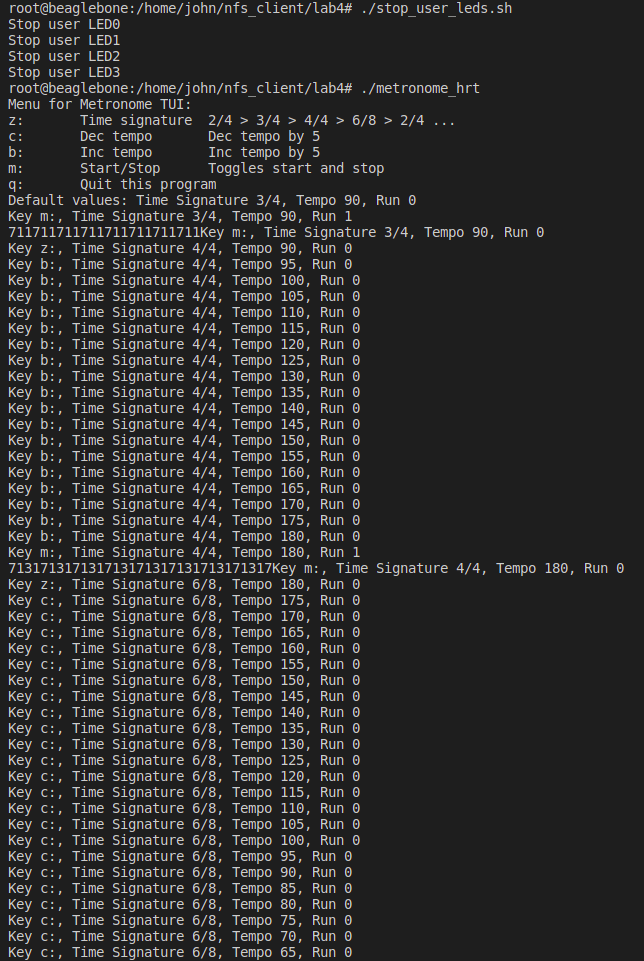
Or: gcc -o metronome\_hrt\_sim metronome\_hrt.c gpio\_led\_fu\_sim.c -lrt

ess

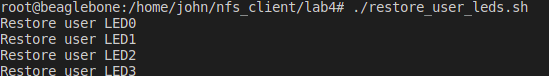
3.2. Actual test on Beaglebones

Command: make met

Or: arm-linux-gnueabihf-gcc -o metronome\_hrt metronome\_hrt.c gpio\_led\_fu.c -lrt







1. Discussion:
2. Search timers on AM3359 and summarize features.

There are 4 types of timers with the following features:

DMTimer: A general-purpose timer module (upward counter). Interrupt on overflow or when matches a programmable value. A programmable clock divider (prescaler) is provided

• Counter timer with compare and capture modes

• Auto-reload mode

• Start-stop mode

• Programmable divider clock source

• 16-32 bit addressing

• “On the fly” read/write registers

• Interrupts generated on overflow, compare and capture

• Interrupt enable

• Wake-up enable

• Write posted mode

• Dedicated input trigger for capture mode and dedicated output trigger/PWM signal

• Dedicated output signal for general purpose use PORGPOCFG

• OCP interface compatible

DMTimer 1ms:

• Counter timer with compare and capture modes

• Auto-reload mode

• Start-stop mode

• Generate 1 ms tick with 32768-Hz functional clock

• Programmable divider clock source

• 16–32 bit addressing

• On-the-fly read/write registers

• Interrupts generated on overflow, compare and capture

• Interrupt enable

• Wake-up enable

• Write posted mode

• Dedicated input trigger for capture mode and dedicated output trigger/PWM signal

• Dedicated output signal for general purpose use PORGPOCFG

• OCP interface compatible

Real-Time Clock Subsystem RTC\_SS:

The basic purpose for the RTC is to keep time of day.

The final purpose of RTC is to wake the rest of chip up from a power down state.

• 100-year calendar (xx00 to xx99)

• Counts seconds, minutes, hours, day of the week, date, month, and year with leap year compensation

• Binary-coded-decimal (BCD) representation of time, calendar, and alarm

• 12-hour clock mode (with AM and PM) or 24-hour clock mode

• Alarm interrupt

• Periodic interrupt

• Single interrupt to the CPU

• Supports external 32.768-kHz crystal or external clock source of the same frequency

Watchdog:

• L4 slave interface support:

– 32-bit data bus width

– 32-/16-bit access supported

– 8-bit access not supported

– 11-bit address bus width

– Burst mode not supported

– Write nonposted transaction mode only

• Free-running 32-bit upward counter

• Programmable divider clock source (2n where n = 0-7)

• On-the-fly read/write register (while counting)

• Subset programming model of the GP timer

• The watchdog timers are reset either on power-on or after a warm reset before they start counting.

• Reset or interrupt actions when a timer overflow condition occurs

• The watchdog timer generates a reset or an interrupt in its hardware integration.

1. Search how the interrupt on AM3359- interrupt request, Acknowledge, Priority, NMI - can be handled in kernel space

**Interrupt request**: We have pins dedicated for interrupt (Int). If Int is 1, processor suspends current program and jumps to an Interrupt Service Routine (ISR).

The source of the interrupt (Processor or Peripheral) can trigger the interrupt by asserting Int to request servicing by the microprocessor. Interrupt forces next instruction to be a subroutine call to a predetermined location (where the ISR locates). Return address is saved to resume executing the main program after finishing the ISR.

Upon getting interrupt signal of M\_IRQ\_n (n=[95:0]) from peripheral, INTC notifies the core of IRQ or FIQ according to M\_IRQ\_n and asserts MPU\_INTC\_IRQ (MPU\_INTC\_FIQ) signal for IRQ (FIQ). Upon detecting MPU\_INTC\_IRQ or MPU\_INTC\_FIQ signal, the microprocessor will execute the interrupt handling process described above.

**Acknowledge**:

Clearing a trigger flag is called acknowledgement, which occurs only by specific software action. Each trigger flag has a specific action software must perform to clear that flag. We will pay special attention to these enable/disable software actions. The SysTick periodic interrupt will be the only example of an automatic acknowledgement. For SysTick, the periodic timer requests an interrupt, but the trigger flag will be automatically cleared when the ISR runs. For all the other trigger flags, the ISR must explicitly execute code that clears the flag.

The Interrupt Acknowledge Register is a read-only register. The Interrupt Acknowledge Register is used to determine the source of the interrupt request received by the CPU. If no interrupt is Pending then the Interrupt ID returned is 1023, spurious interrupt.

**Priority**: Priority is decided by the interrupt controller to determine what interrupt gets CPU first.

We can assign priority for each interrupt line by:

– Assign priority level (0 being the highest)

– Assign interrupt request type (FIQ or IRQ): Priority FIQ > IRQ

Software prioritization: Interrupt priority set by programming the interrupt controller

– By setting registers in the interrupt controller

– The approach in ARM cores

Hardware prioritization types

– Fixed priority

• each peripheral has unique rank

• highest rank chosen first with simultaneous requests

• preferred when clear difference in rank between peripherals.

– Rotating priority (round-robin)

• priority changed based on history of servicing

• better distribution of servicing especially among peripherals with similar

priority demands

• More equitable distribution of service

**NMI**: Non-maskable interrupt (NMI):

– A separate interrupt pin that can’t be masked. It has the highest-priority, never masked. (usually programmer can set bit that causes processor to ignore interrupt but we don’t allow it here).

– Typically reserved for drastic situations, like power failure. It needs immediate backup of data to non-volatile memory.

V- References:

[1] Getting Started with Beaglebone and Beaglebone Black,

http://www.beagleboard.org/Getting%20Started

[2] Beaglebone Rev. A5. System Reference Manual,

http://circuitco.com/support/index.php?title=Beaglebone#Rev\_A5. NOTE.

[3] “Embedded Linux Primer”, C. Hallinan, Prentice Hall.

[4] Lab and lecture material, EE414 Teaching Staffs, KLMS

[5] EBC Exercise series, elinux.org

[6] GPIO Programming series, ics.com

[7] AM335x reference manual

[8] <https://users.ece.utexas.edu/~valvano/Volume1/E-Book/C12_Interrupts.htm>

[9] <https://developer.arm.com/documentation/ddi0360/f/mpcore-distributed-interrupt-controller/cpu-interrupt-interface-registers/interrupt-acknowledge-register--0x0c>