EC-ENG 231 (Spring 2024)

Programming for Interrupts

Fatima Anwar
fanwar@umass.edu
UMass Amherst

Announcements

- **Mandatory Lab Sessions this week:**
  - Show your TA a working beaglebone setup
  - Show your TA blinding LED from Lab2

- Added more instructor support hours (Check on Moodle):
  - ALL Mondays: 12 - 1:30 pm at my office KEB209E
  - Thursday Feb 29th: 11 am - 1 pm at KEB209E
  - Wednesday Mar 6th: 11 am - 1 pm at Marston Lab
  - Friday Mar 15th: 2:30 - 4:30 pm at Marston Lab
  - Thursday Mar 28th: 11 am - 1 pm at Marston Lab

Announcements

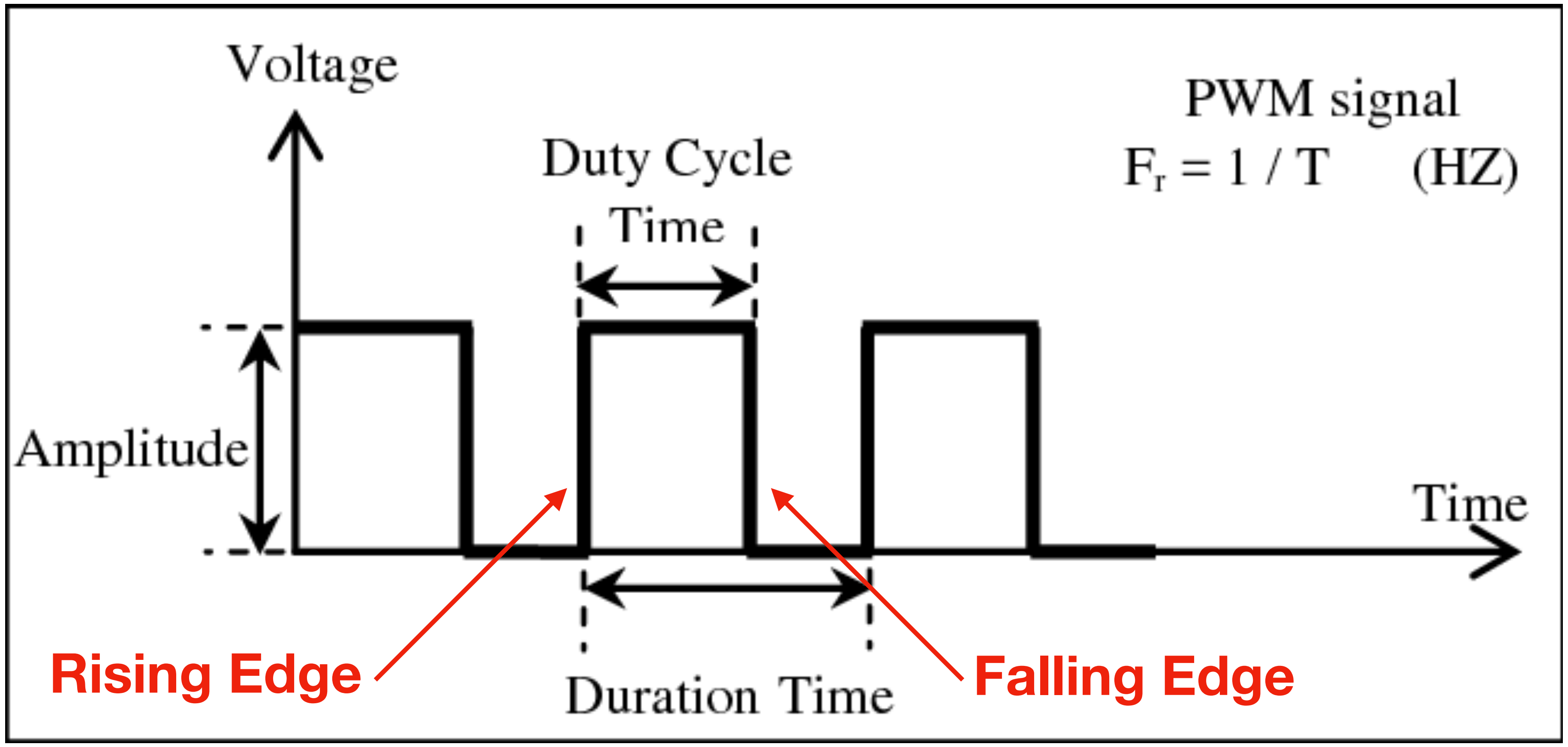
- New lab released on Thursdays, but due on Saturdays
  - Lab 3 is due on Saturday March 9th
  - Lab 4 due: March 16th
  - Lab 5 due: March 30th

- Midterm is on March 27th Wednesday, 2:30 to 3:45 pm (during class time)

Announcements

"Disability Services is in need of a **note taker** for this class. If you are interested, please email **notes@admin.umass.edu** with your name, student ID and the course info (i.e. BIOLOGY 100, Section 2, Professor Jones). Disability Services staff will contact you to confirm and provide you instructions. You may earn 45 hours of community service for your efforts, or 1 pass/fail practicum credit."

Pulse Width Modulation (PWM) Example

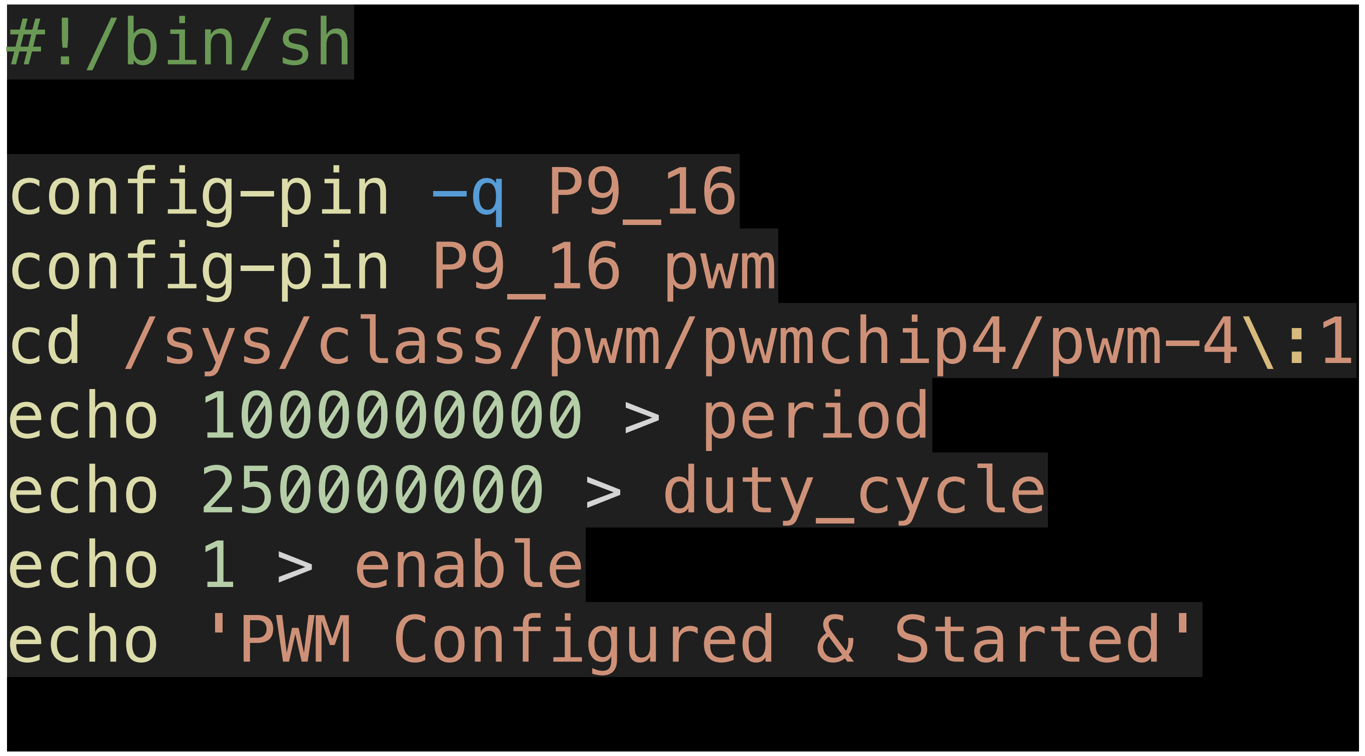


Frequency = 100Hz
Period = 1 / 100 = 0.01 second = 10 milliseconds = 10,000,000 nanoseconds

Duty cycle = 80%
ON period = 0.01 x 0.8 = 8 milliseconds = 8,000,000 nanoseconds
OFF period= 0.01 x 0.2 = 2 milliseconds = 2,000,000 nanoseconds

PWM using Shell script

- Write a shell script that configures a pin to generate a PWM with a period of 1 second and 25% duty cycle

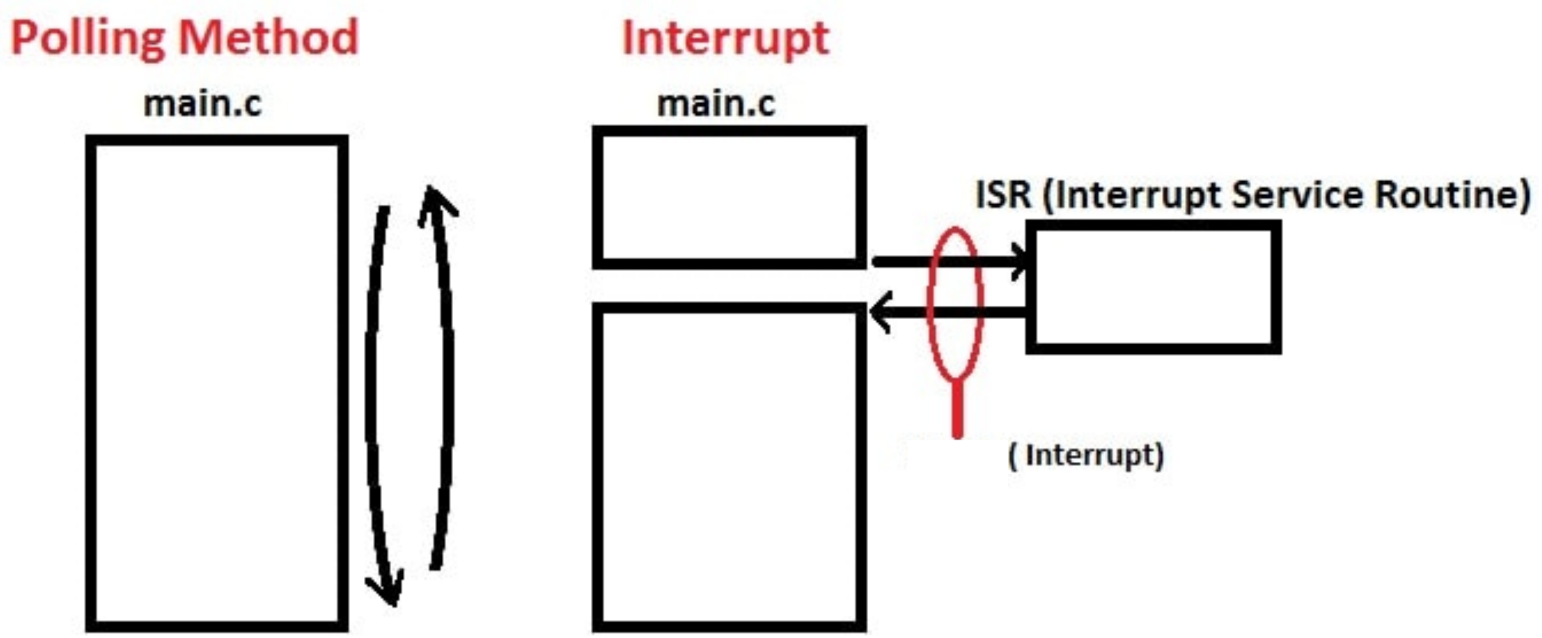


```sh
#!/bin/sh

config-pin -q P9_16
config-pin P9_16 pwm
cd /sys/class/pwm/pwmchip4/pwm-4\:1
echo 1000000000 > period
echo 250000000 > duty_cycle
echo 1 > enable
echo 'PWM Configured & Started'
```

Interrupts v/s Polling

- There are many sources of interrupts,
  - External devices
  - Peripherals and hardware on chip. e.g. timers
  - Software Exceptions
- Different Approaches
  - Polling: Software checks the status of event frequently
    - Cons: Wasteful of time, ugly code, inefficient
  - Interrupts: Asynchronous Event Driven



https://komhedos.com/polling-metodu-ile-interrupt-arasindaki-farklar/

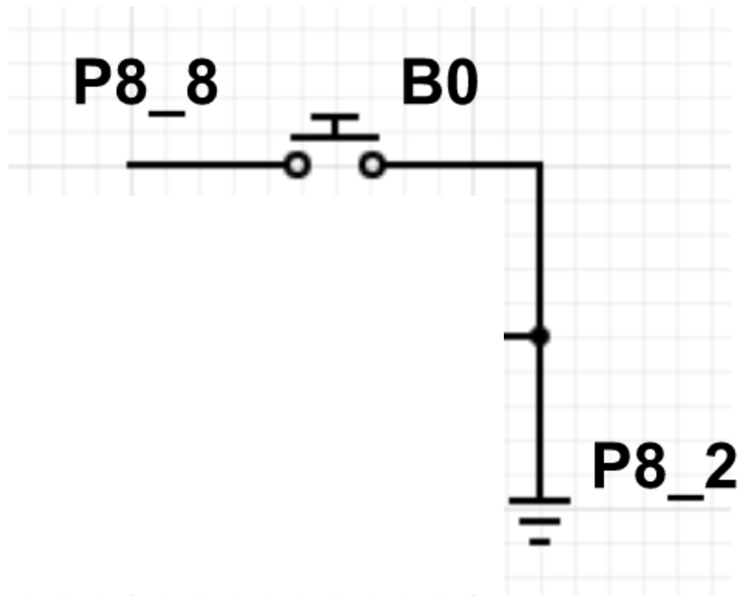
PWM Polling

- Attach the PWM pin to a GPIO using a jumper wire
- PWM will trigger the GPIO pin and desired statement is printed

```
int main(){
    // configure pin P8_8 as input with internal pull-up enabled
    int gpio_number = 67;
    configure_gpio_input(gpio_number);
    //  file path to read button status
    char valuePath[40];
    sprintf(valuePath, "/sys/class/gpio/gpio%d/value", gpio_number);
    // wait before first readings to avoid faulty readings
    sleep(1);

    long count = 0;
    int state;
    FILE *fp;
    // loop to monitor events
    while(1){
         fp = fopen(valuePath, "r");
         // default state is 1 since buttons are configured with
         // internal pull-up resistors. So, reading 0 means button
         // is pressed
         fscanf(fp, "%d", &state);

         fclose(fp);
         // event detected
         if( state == 0 ){
            count++;
            printf("Pin Interrupted %lu\n", count);
         }
    }
    return 0;
}
```



Interrupts Programming

Interrupts in Beaglebone

- GPIOs can be configured to receive interrupts as inputs
- Developers write ISR (interrupt service routines) mostly for 8-bit microcontrollers, and occasionally for real time operating systems
- In Linux kernel, interrupts are handled by pre-programmed ISRs at the device driver level
- In Beaglebone, GPIO interrupts are processed by GPIO driver
  - Developers can configure gpios to receive interrupts using standard Linux calls,
    1. Configure: **register** the events of interest
    2. Capture: **wait** for those events to happen

https://developer.ridgerun.com/wiki/index.php?title=How_to_use_GPIO_signals

Interrupts in Beaglebone

1. Program interrupts using these steps:
   1. Configure the GPIO pin as input (as described in GPIO lecture)
   2. Once configured as input, interrupts can be enabled on the pin by writing to the file:

      `/sys/class/gpio/gpio<number>/edge`

      - Interrupts are triggered by `rising`, `falling` or `both` edges by writing "rising", "falling" or "both" (without quotation marks) in this file
   3. Once interrupts are configured, the user can read the file:

      `/sys/class/gpio/gpio<number>/value`

      - Upon reading this file, the content is either "1" when the interrupt has occurred or "0" when there is no interrupt

Configure interrupt using command line

- Enable rising edge interrupt on P8_9 (gpio69) of beaglebone black,

```
echo rising > /sys/class/gpio/gpio69/edge
```
[Pin interrupted at rising edge]

```
cat /sys/class/gpio/gpio69/edge
```
[Prints the current content of edge file]

```
cat /sys/class/gpio/gpio69/value
```
[Prints the current content of value file]

Configure interrupt in C code

```c
//Sets up gpio pin as input
void configure_gpio_input(int gpio_number){
    char gpio_num[10];
    sprintf(gpio_num, "export%d", gpio_number);
    const char* GPIOExport="/sys/class/gpio/export";

    // exporting the GPIO to user space
    FILE* fp = fopen(GPIOExport, "w");
    fwrite(gpio_num, sizeof(char), sizeof(gpio_num), fp);
    fclose(fp);
    // setting gpio direction as input

    char GPIODirection[40];
    sprintf(GPIODirection, "/sys/class/gpio/gpio%d/direction", gpio_number);
    // setting GPIO as input
    fp = fopen(GPIODirection, "w");
    fwrite("in", sizeof(char), 2, fp);
    fclose(fp);
}

// sets up an interrupt on the given GPIO pin
void configure_interrupt(int gpio_number){
    // set gpio as input
    configure_gpio_input(gpio_number);

    // configure interrupts using edge file
    char InterruptEdge[40];
    sprintf(InterruptEdge, "/sys/class/gpio/gpio%d/edge", gpio_number);
    // configure interrupt on falling edge
    FILE* fp = fopen(InterruptEdge, "w");
    fwrite("falling", sizeof(char), 7, fp);
    // configure interrupt on both edges
    //fwrite("both", sizeof(char), 4, fp);
    fclose(fp);
}
```

Configure GPIO pin as input (e.g. gpio67)

Configure GPIO pin to receive interrupts on “falling” edge

Uncomment this if configuring GPIO pin to receive interrupts on “both” edges

Capture interrupt - Linux

- Beaglebone GPIO Driver has implemented Linux poll functions,
  - Linux Poll functions help passively wait on the file descriptors and not waste the CPU cycles
  - Potentially wait on multiple file descriptor at the same time and handles events based on their occurrence
  - Program that polls the file descriptor is put to sleep, and events associated with the wait_queue are used to wake up the process
  - Linux method used for interrupt handling with minimal CPU overhead,
    - `epoll` - I/O event notification facility

Capture interrupt - Linux epoll

- Linux **`epoll`** from the **`sys/epoll.h`** header file
- **`epoll_create`** system call creates a new epoll instance. The ***`size`*** argument is ignored, but must be greater than zero

```
int epoll_create(int size);
```

  - returns an integer file descriptor referring to the new epoll instance
  - returned file descriptor is used for all the subsequent calls to the epoll interface
  - when no longer required, the file descriptor returned by **`epoll_create`**() should be closed by using close function
  - when all file descriptors referring to an epoll instance have been closed, the kernel destroys the instance and releases the associated resources for reuse

Capture interrupt - Linux epoll

- Linux **`epoll`** from the **`sys/epoll.h`** header file
- **`epoll_ctl`** system call is used to add, modify, or remove entries in the interest list of the epoll instance referred to by the file descriptor ***`epfd`***

```
int epoll_ctl(int epfd, int op, int fd,
struct epoll_event *_Nullable event);
```

- When successful, **`epoll_ctl`**`()` returns zero
- It requests that the operation ***`op`*** be performed for the target file descriptor, ***`fd`***
  - **`EPOLL_CTL_ADD`**: Add an entry to the interest list of the epoll file descriptor, ***`epfd`***
- Register events of interest. Some of the available ***`event`*** types are
  - **`EPOLLIN`**: requests associated file to be available for read operations
  - **`EPOLLET`**: requests edge-triggered notification for the associated file descriptor

Capture interrupt - Linux epoll

- **`epoll_wait`** system call waits for events on the epoll instance referred to by the file descriptor *epfd*

```
int epoll_wait(int epfd, struct epoll_event *events,
int maxevents, int timeout);
```

  - on success, it returns the number of file descriptors ready for the requested I/O, or zero if no file descriptor became ready during the requested *`timeout`* milliseconds
  - referred to by the file descriptor *`epfd`*
  - buffer pointed to by *`events`* is used to return information from the ready list about file descriptors in the interest list that have some events available
  - up to *`maxevents`* are returned by **`epoll_wait`**(). The *maxevents* argument must be greater than zero
  - *`timeout`* argument specifies the number of milliseconds that **epoll_wait**() will block. Time is measured against the CLOCK_MONOTONIC clock

- A call to **epoll_wait**() will block until either:

https://man7.org/linux/man-pages/man2/epoll_wait.2.html

Capture interrupt - Linux epoll

- **`epoll_wait`** system call waits for events on the epoll instance referred to by the file descriptor *epfd*

```
int epoll_wait(int epfd, struct epoll_event *events,
int maxevents, int timeout);
```

- A call to **`epoll_wait`** will block until either:
  - a file descriptor delivers an event;
  - the call is interrupted by a signal handler; or
  - the timeout expires.

https://man7.org/linux/man-pages/man2/epoll_wait.2.html

Capture interrupt in C code

```c
int main(){
    int gpio_number = 67; // P8_8 gpio pin
    configure_interrupt(gpio_number);  // enable interrupt

    // the following code can be used to receive interrupts on the registered pin
    char InterruptPath[40];
    sprintf(InterruptPath, "/sys/class/gpio/gpio%d/value", gpio_number);

    int epfd;
    struct epoll_event ev;

    // (STEP 1) open the interrupt file
    // file pointer (C abstraction to manipulate files)
    FILE* fp = fopen(InterruptPath, "r");
    // file descriptor (Unix Linux file identifier used by system calls)
    int fd = fileno(fp);

    // (STEP 2) create epoll instance to monitor I/O events on interrupt file
    epfd = epoll_create(1);

    // (STEP 3) register events that will be monitored
    // detects whenever a new data is available for read (EPOLLIN)
    // signals the read events when the available read value has changed
    ev.events = EPOLLIN | EPOLLET;

    // (STEP 4) register interrupt file with epoll interface for monitoring
    ev.data.fd = fd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
```

monitor this "value" file to check for interrupt

Setup for Linux epoll to monitor the "value" file for changes

open the "value" file

A file descriptor is an integer value associated with an open file, which is used by the linux kernel to manipulate that file

obtain the file descriptor of "value" file

create epoll instance to monitor I/O events

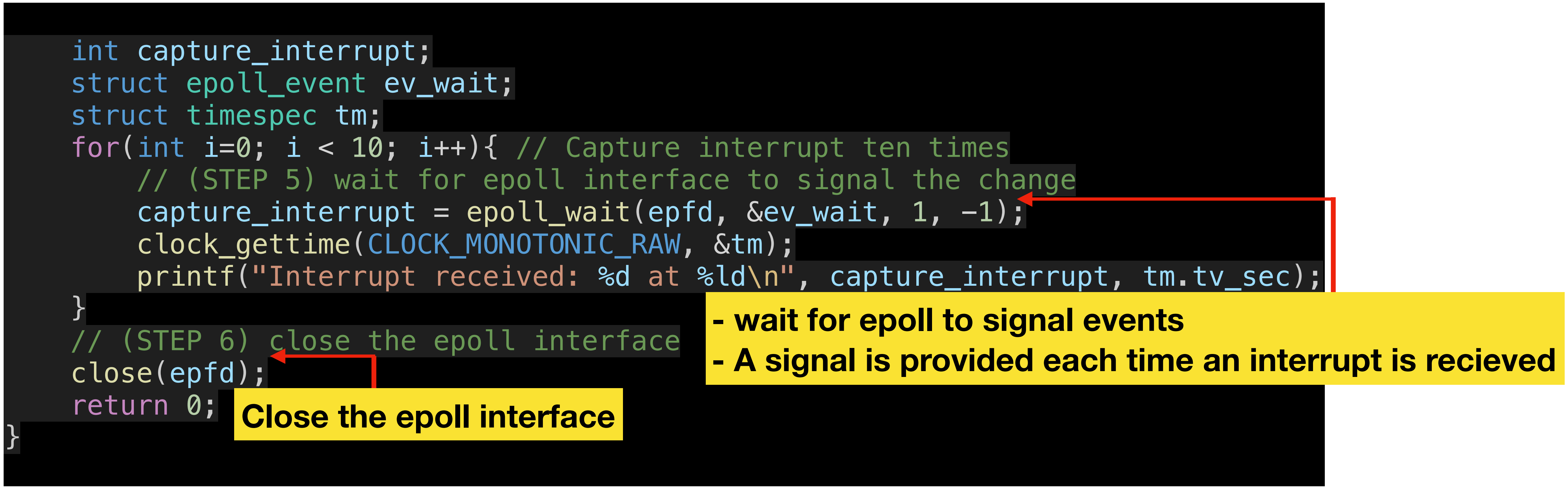
EPOLLIN (A new value in the file is available for read) and EPOLLET (edge triggered: when the value available for read has changed)

register specific I/O events:

register events and file descriptor with epoll instance

Code continues on next slide..

Capture interrupt in C code



```c
    int capture_interrupt;
    struct epoll_event ev_wait;
    struct timespec tm;
    for(int i=0; i < 10; i++){ // Capture interrupt ten times
        // (STEP 5) wait for epoll interface to signal the change
        capture_interrupt = epoll_wait(epfd, &ev_wait, 1, -1);
        clock_gettime(CLOCK_MONOTONIC_RAW, &tm);
        printf("Interrupt received: %d at %ld\n", capture_interrupt, tm.tv_sec);
    }
    // (STEP 6) close the epoll interface
    close(epfd);
    return 0;
}
```

- wait for epoll to signal events
- A signal is provided each time an interrupt is recieved

Close the epoll interface

Reading

- Practice the code on your devices
- Book: Read Chapter 6: Interfacing to Beagleboard Inputs/Outputs
  - Advanced GPIO Topics: Linux poll, Enhanced GPIO Class
- Read all the links provided in slides