Overview

Today: Periodic tasks

- Real-time periodic task and the problems of jitter and drifts
- Implementation of real time periodic task by using ISR and hw timers
- Servo motors
- Output compare module for Pulse Wide Modulation (PWM)
- touchscreen

The content of this lecture is partially covered by

- The content of this lecture is partially covered by the lab manual (pp. 29-30,12-14, and 27-29). Additional material is on dsPic data sheet (section 13, Output Compare).

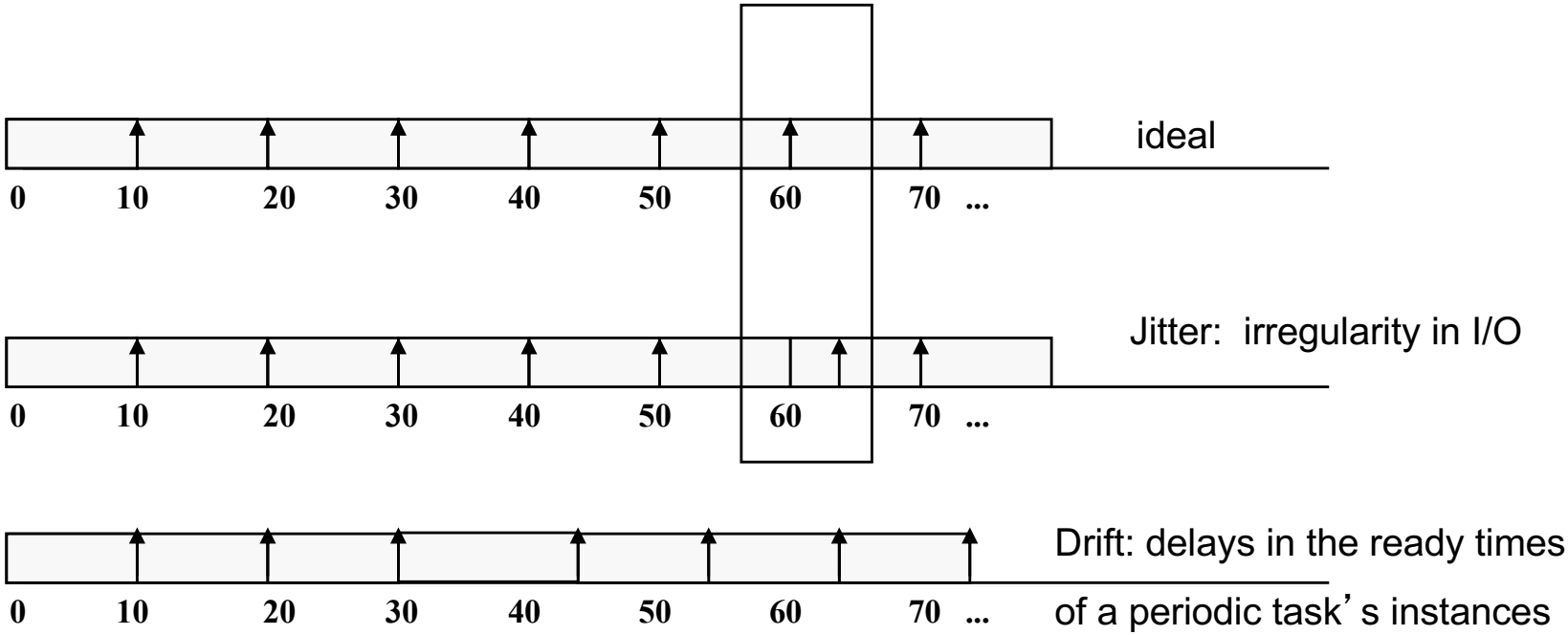
Periodic Tasks

- Periodic tasks are commonly used in embedded real time systems, e.g., a 10Hz task updates the display and a 20 Hz task samples the temperature.

- Rate monotonic scheduling (RMS) algorithm is commonly used in practice. RMS assigns high priorities to tasks with higher rates, e.g., 20 Hz task should be given higher priority than 10 Hz tasks.
  - If every instance of a periodic task has the same priority, it is called a fixed priority scheduling method
  - commercial RTOS support only fixed priority scheduling
  - RMS is an optimal fixed priority scheduling method
  - The timing behavior of a real time system scheduled by RMS, can be fully analyzed and predicted by Rate Monotonic Analysis (RMA). We will study this subject later in the course
  - we will study the design and implementation of periodic tasks on dsPic board

Periodic Tasks: Jitters & drifts

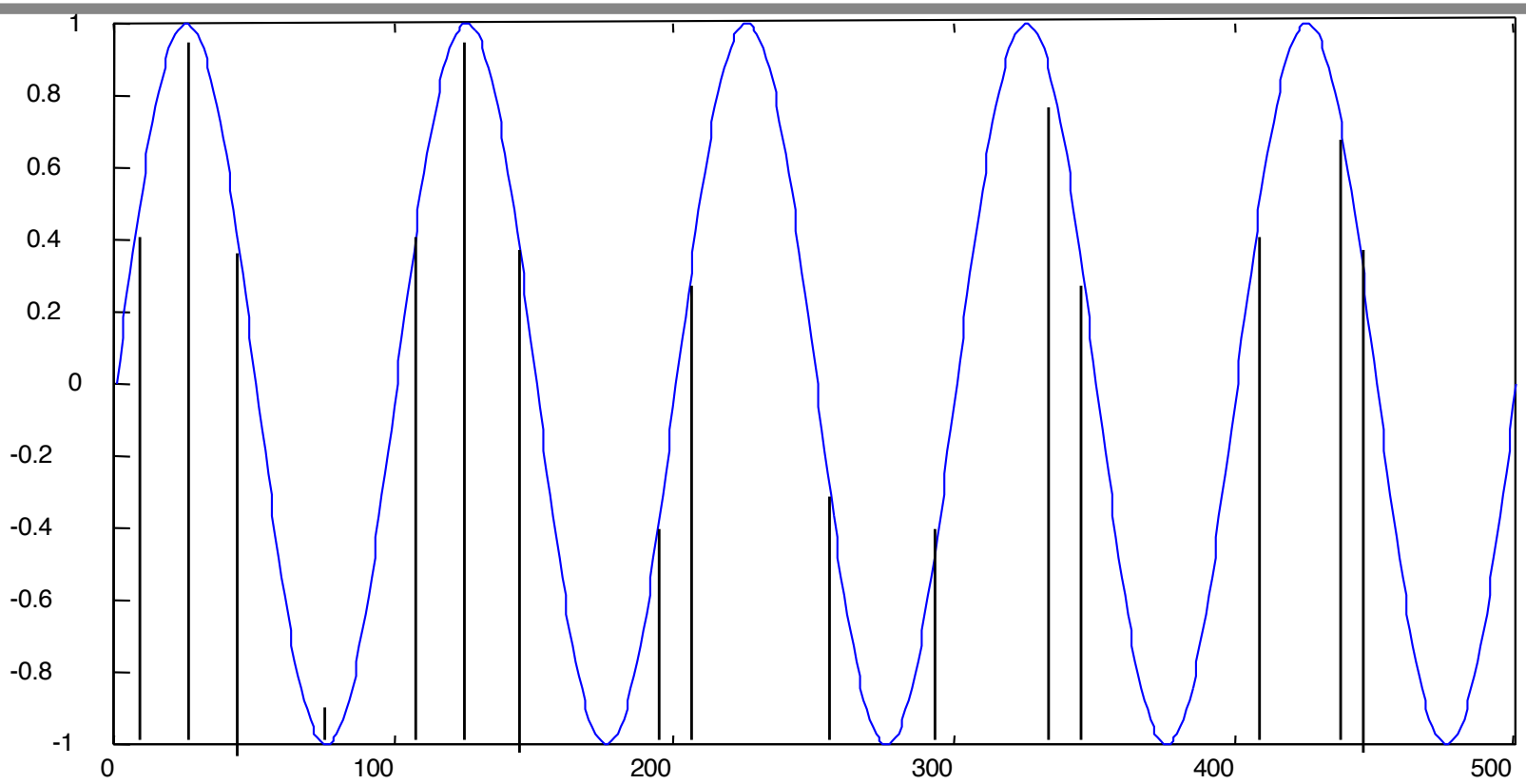
- A periodic task should repeat regularly according to a given period.
- For example, consider a periodic task with period 10 starting at t = 0.



↑ I/O operations

| periodic task's activation

Why Jitter is Bad?



- Jitters and drifts affect performance of sensing and control:
  - Classic control theory requires the digital controller to run periodically at constant rate. I/O has to be performed periodically too. Jitters and drifts, in the worst case, can cause the instability of controlled plant.
  - Sampling of signals is sensitive to jitters and drifts too: they introduce distortion on the sampled signals.

Evaluation: Where in the Code are Jitters and Drifts

- Suppose that we want to implement a periodic task in a Linux system and we want to use functions like **time** and **sleep**:

time_t **time(time_t *what_time_it_is)**

unsigned int **sleep(unsigned int nseconds)**

- According to UNIX and POSIX.1, both functions have resolution of seconds! However, just imagine that sleep and time actually could provide the resolution of milliseconds (for instance, see nanosleep!)

Evaluation: Where in the Code are Jitters and Drifts?

- Suppose that we want to implement a periodic task in a Linux system and we want to use functions like time and sleep. Let's assume that the tick resolution is 1ms.

- Suppose that we want to control a device using a 20 msec task starting at instant START_TIME:

```
1. current_time = time(NULL)
2. If  (START_TIME - current_time < 1 msec) {//report too late and exit}
3. sleep(START_TIME - current_time)
 loop
      4.  current_time = time(NULL)
      5.  wake_up_time = current_time + 20 msec
      6.  //read sensor data from the device, it takes << 20 msec
      7.  //do work, It takes  << 20 msec
      8.  current_time = time(NULL)
      9.  //send control data to the device
      10.sleep(wake_up_time - current_time)
 end_loop
```

- Quiz: can you identify operations that are drift and jitter prone?
- Hint: think of paging, multi-tasking and preemption.

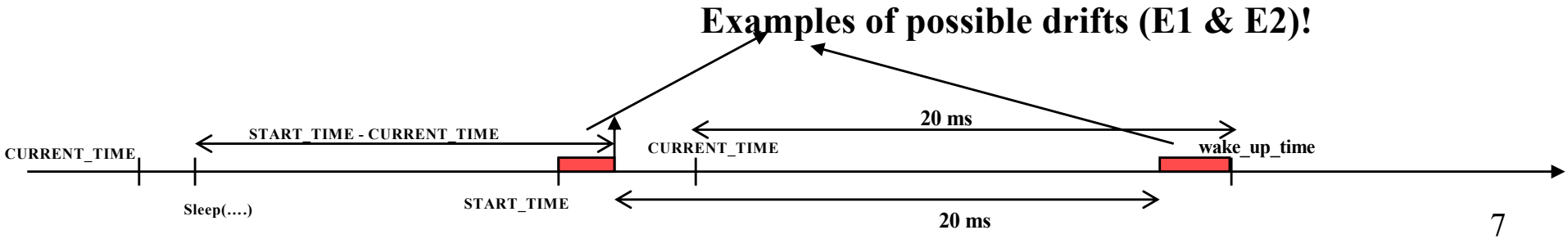
Potential Timing Problems

1. current_time = time(NULL)
2. If (START_TIME - current_time < 1 msec) {//report too late and exit}
3. sleep(START_TIME - current_time) //E1: if process is preempted between lines 1 and 3, there will be a **drift**!

loop

4. current_time = time(NULL)
5. wake_up_time = current_time + 20 msec //E2: **drift** if line 4 is delayed by a preemption
6. Read sensor data //E3: if preempted, **input jitter**
7. //do work
8. current_time = time(NULL)
9. //send control data to the device //E4: **output jitter** (caused by preemption or variable execution time)
10. sleep(wake_up_time - current_time) //E5: if process is preempted between lines 8 and 10, there will be a **drift**!

end_loop



Potential Timing Problems

- Summarizing all the potential timing problems:
- The process could be swapped out of memory (drift and jitter): pin it down in main memory! In LynxOS, processes with priority greater than 16 (default) will not be swapped out.
- Function sleep has drifting problems (see previous slide!)
- Every line which manipulates time or does I/O, has problems. **To solve them, we need to use signals and timers**

Timing problems: a solution for dsPic board when I/O is short

- **Jitters and drifts affect performance of sensing and control**:
  - Classic control theory requires the digital controller to run periodically at constant rate. I/O has to be performed periodically too. Jitters and drifts, in the worst case, can cause the instability of controlled plant.
  - Sampling of signals is sensitive to jitters and drifts too: they introduce distortion on the sampled signals.

- **To solve the drift problem**, use a periodic hardware based timer to kick off each instance of a periodic task, it will be ready at the correct time instants.

- **To minimize the jitter problem when I/O is short**, do the I/O in the timer interrupt handler. As long as each task finishes before its end of period, I/O can be done at nearly the regular instants of timer interrupts: it gives the highest regularity.

Timing problems: a solution for dsPic board when I/O is short

- **Initialization:** lock mutex wake_task_1_up
  ➔this mutex synchronizes the action of handler and application task.

- **Timer_interrupt handler()** //*by convention* executes before application tasks
  { //Do I/O: sends data to hardware from output buffer and gets sensor data into input buffer;
  //unlock wake_task_1_up}

- **Task_1**
  loop
  //lock wake_task_1_up (to block the loop, otherwise, it loops non-stop)
  //read the data from input buffer, do computation & put result in output buffer;
  end_loop

- Note: the I/O are still interfering with each others. However, when the duration of I/O operations are short, this method works just fine.
- **Quiz**: Is it correct if we do both work and I/O in the interrupt handler?

Timing problems: a solution for dsPic board when I/O is short

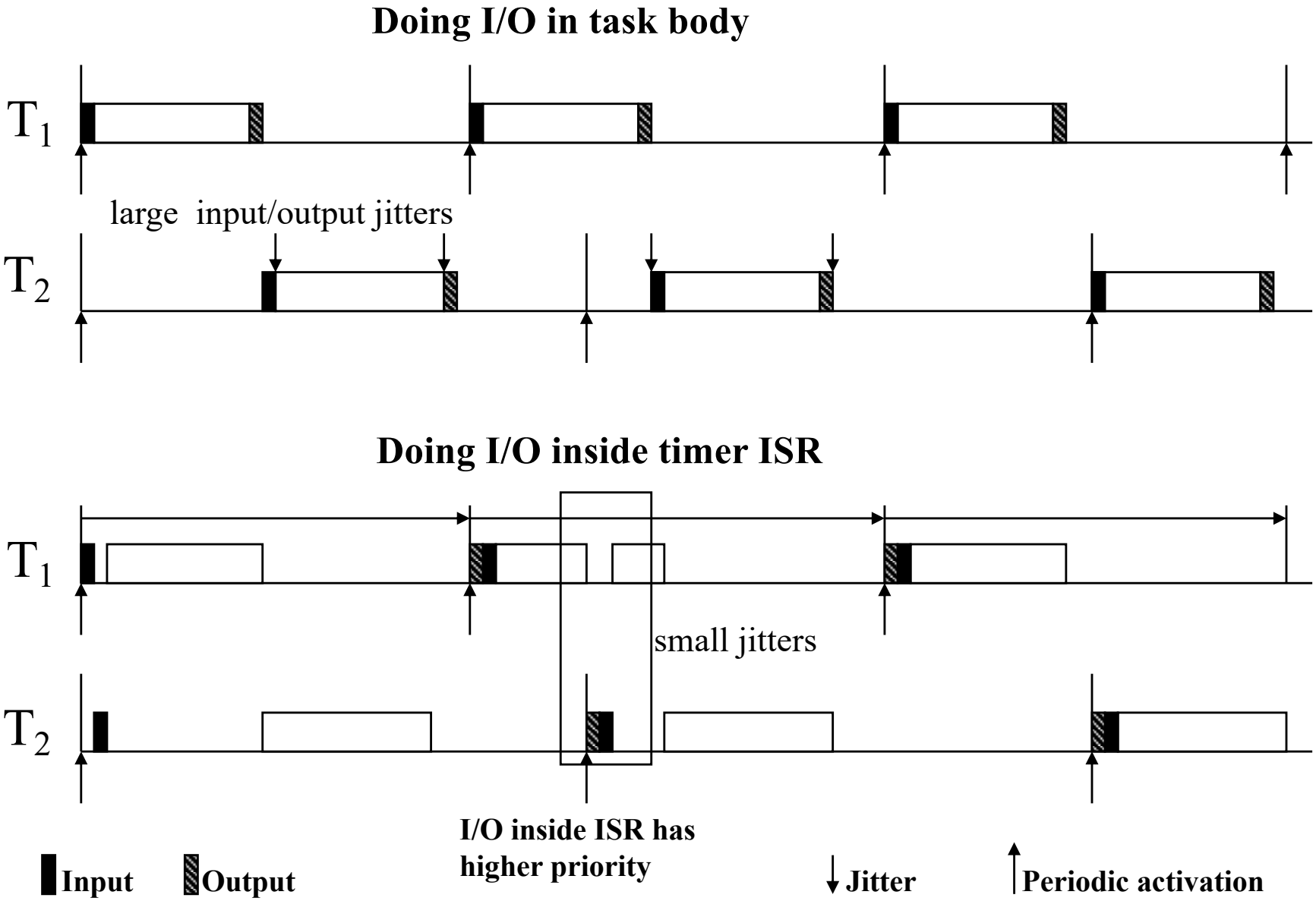
- **Initialization:** lock mutex wake_task_1_up
  ➔this mutex synchronizes the action of handler and application task.

- **Timer_interrupt handler()** //*by convention* executes before application tasks
  { //Do I/O: sends data to hardware from output buffer and gets sensor data into input buffer;
  //unlock wake_task_1_up}

- **Task_1**
  loop
  //lock wake_task_1_up (to block the loop, otherwise, it loops non-stop)
  //read the data from input buffer, do computation & put result in output buffer;
  end_loop

- Note: the I/O are still interfering with each others. However, when the duration of I/O operations are short, this method works just fine.
- **Quiz**: Is it correct if we do both work and I/O in the interrupt handler?
- **Answer**: interrupt handlers would become too long and they will start to interfere among each others reintroducing jitters back.

Solution Approach When I/O is short



Solutions Approach: When RT I/O is too long

- Consider now the case that I/O operations are long because high volume of data needs to be exchanged.

- In this case, hardware solution is called for. In high end A/D – D/A cards, there is a hardware timer for each input and output channel. For example,

  - ch 0 is input.
    - We can program its timer to sample the input line every 20 ms, starting at time 0.
    - Put the data in ch 0's input buffer (jitter free read).
    - Generate an interrupt to trigger Task_1 execution.

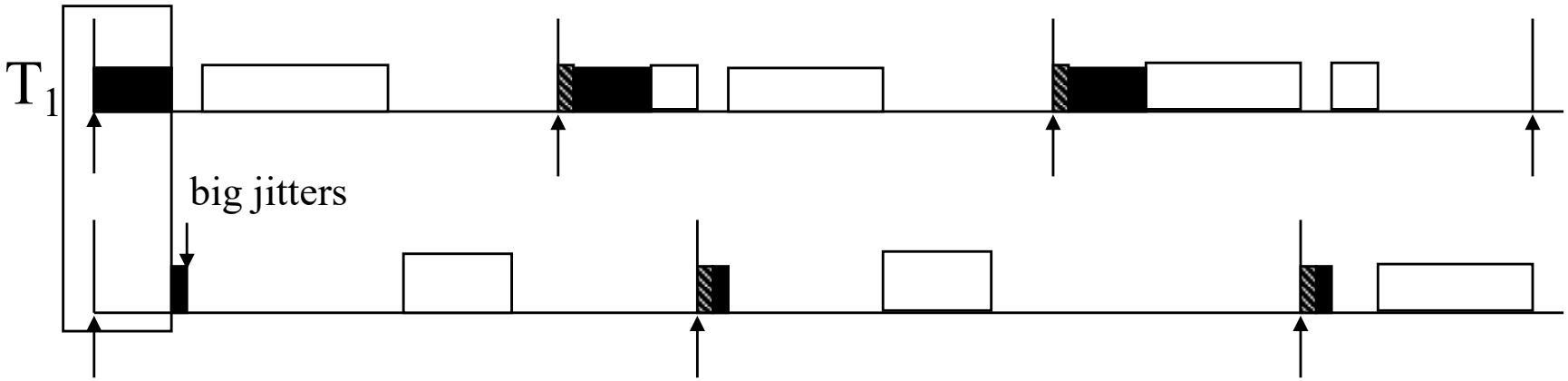
  - A/D ISR: kicks off the Task_1 body

  - Task_1 body: read the data from buffer, do work and put result to ch1 buffer

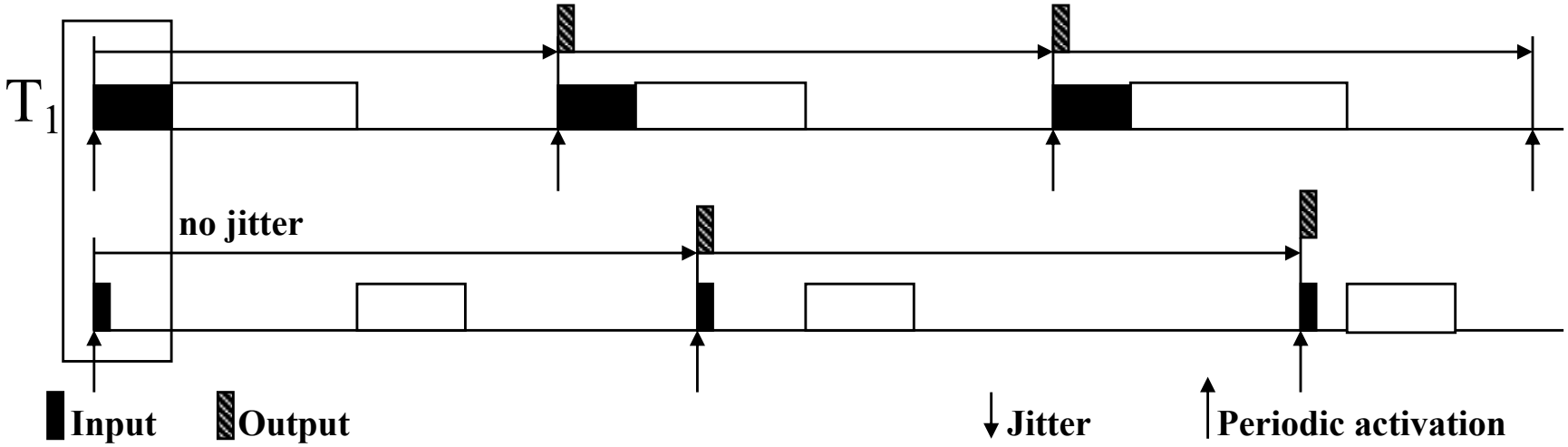
  - ch 1 is output. We program its timer to output data from its buffer every 20 ms starting at 20 ms (jitter_free output). Do not start at time zero, because no output data is ready at that time.

Solutions Approach: When RT I/O is too long

Doing I/O inside timer ISR for Long I/O

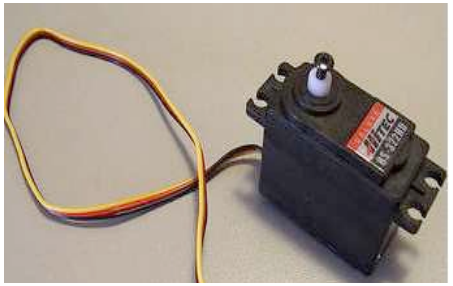


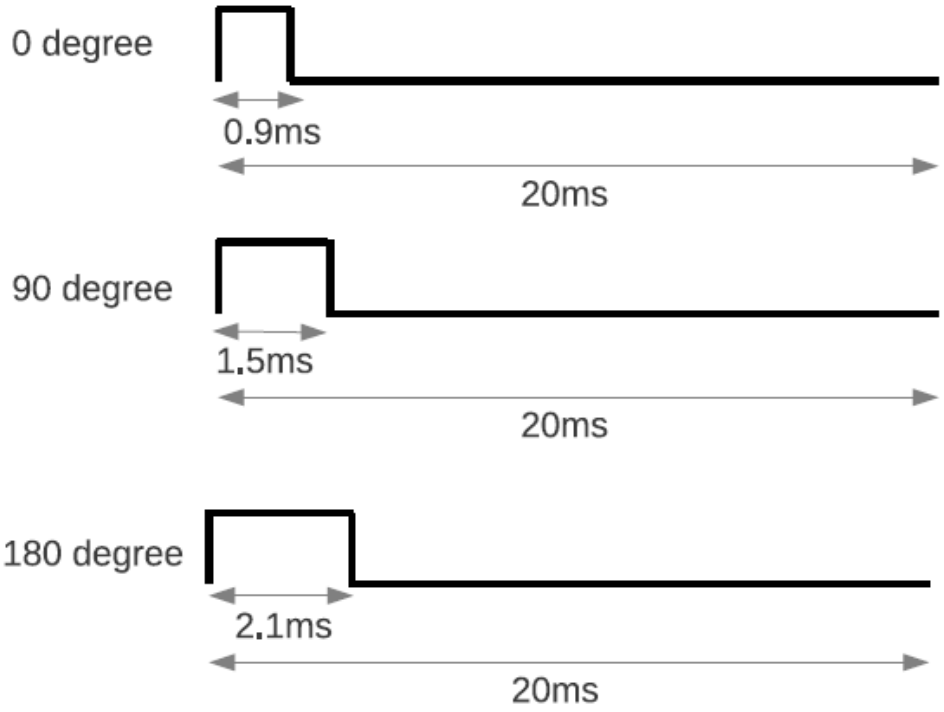
Doing I/O with external hardware timers



Servo motors

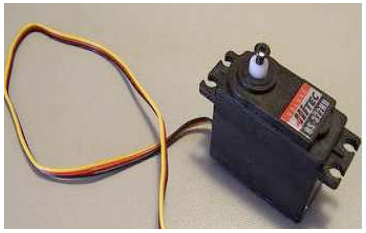
- The Amazing Ball System has two servos: one for X dimension and one for Y dimension.
- Servos are commanded through “Pulse Width Modulation”, or PWM signals sent through the command wire.
- The width of a pulse defines the position. For instance, in our lab, sending a 0.9ms pulse to the servo, tells the servo that the desired position is 0 degrees. In order for the servo to hold this position, the command must be sent at about 50Hz, or every 20ms.





Servo motors

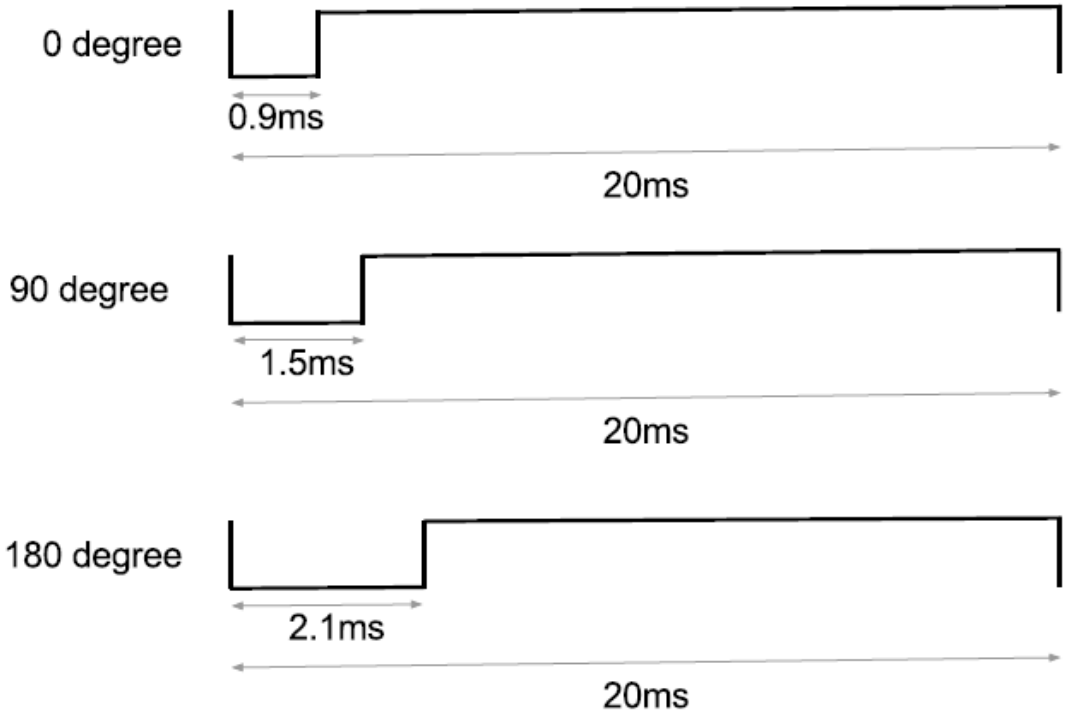
- A servo is a motor that includes a position feedback circuit
- Servos are commonly used in radio controlled models and they can be instructed to move to a specified "position“. Servos are constrained from full rotation; in fact, usually they have a limited rotation of 180 degrees or less.
- A servo has three wires: Vdd, Control, and GRN. They typically run on 4.8v and the control line drives the angle position.
- Servos are controlled by a periodic "pulse" with variable width. The pulse has a minimum width, a maximum width, and a period. By convention, a pulse of approximately 1.5 ms is the "neutral” point for the servo. Neutral is defined as the intermediate position (half of permitted rotation or 90 degree)
- Servos are "active" devices, meaning that when they are set on a position under the control of a 50Hz pulse they will actively hold the position. For example, if a servo is controlled toward the neutral position and a mechanical force is pushing against the servo’s arm, it will actively apply a torque to hold the set position. The maximum amount of torque the servo can exert is part of its specifications.



Servo motors

- In the Amazing Ball System, the command wires of X and Y servo are connected to Output Compare pin OC8 and OC7, respectively, through servo controllers.

- Notice that the servo controllers invert the signal from OC8 and OC7, so when the Output Compare pin is 1 the servo command signal is 0 and vice-versa. As a consequence, to correctly drive the servo motors, the Output Compare module needs to generate an 'inverted' signal as shown in figure below.



Note:

The inverted signal will not be strictly periodic (~ 47-53Hz). In fact, assuming a timer period of 20msec, the variable duty cycle (from 0.9 to 2.1 msec) will cause the PWM cycle to fluctuate between 20 - .9 +2.1=21.2 msec and 20-2.1+.9=18.8 msec

Servo motors & Output compare

- If command pulse is longer than 2.1ms or shorter than 0.9ms, the servo would attempt to overdrive (and possibly damage) itself.

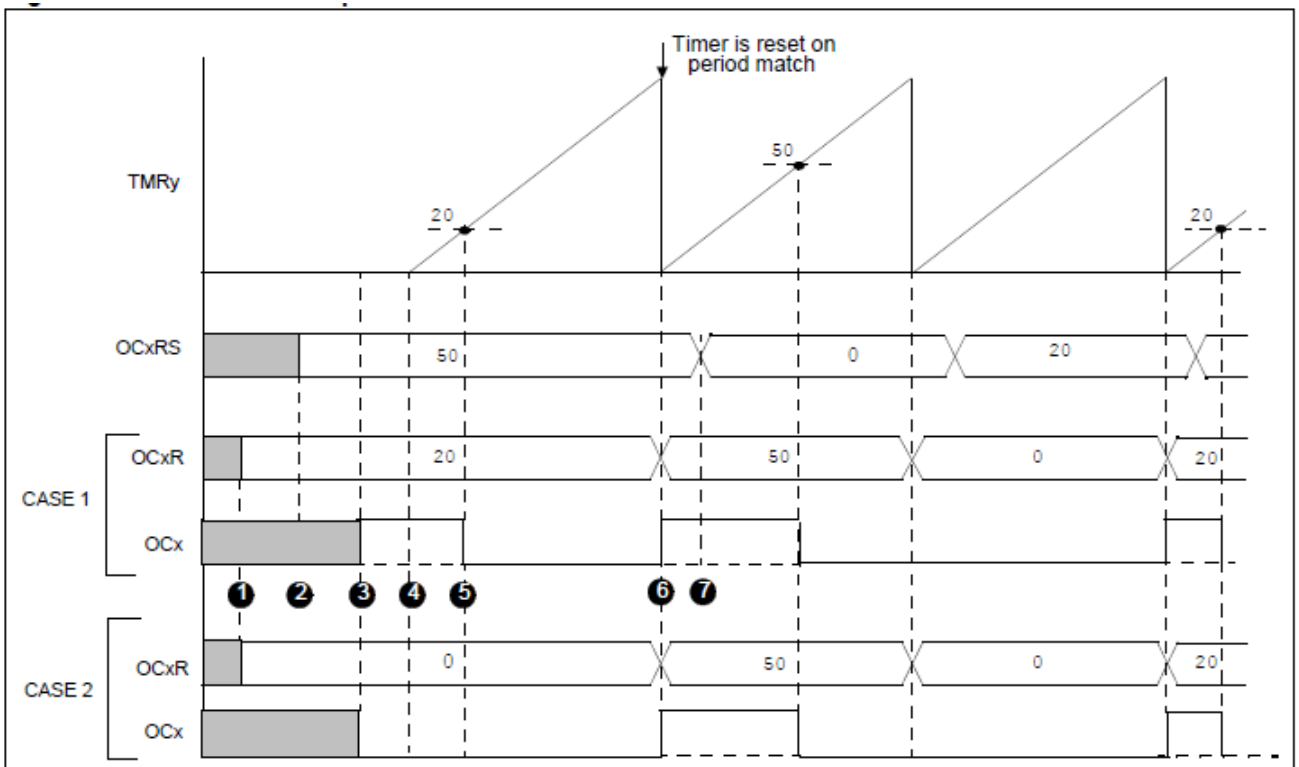
- To control the servos, users will have to setup OC8 and OC7 to work in PWM mode.

- The output compare module controls output pins (OCx) by comparing the value of a timer with the value of one or two compare registers depending on the operating mode. The state of the output pins change when the timer value matches the compare register value.

- Each Output Compare module consists of the following readable/writable registers:
  - OCxR: 16-bit Output Compare register (current PWM duty cycle)
  - OCxRS: 16-bit Secondary Output Compare register (next PWM duty cycle)
  - OCxCON: 16-bit control register associated with the Output Compare

Output compare in PWM mode

- In PWM mode, the OCx pin is:
  - Driven high if the OCxR register value is non-zero (see Case 1)
  - Driven low if the OCxR register value is zero (see Case 2)
- When a selected timer is enabled, it starts incrementing until it reaches the value in the period register. The Compare Register (OCxR) value is constantly compared with the timer value. When a match occurs, the OCx pin is driven low.
- On a timer rollover (period match), the OCxRS value is loaded into the OCxR register and the OCx pin is:
  - Driven high if the OCxR register value is non-zero
  - Driven low if the OCxR register value is zero



1. The duty cycle for first PWM cycle is written to OCxR register before enabling the PWM mode.
2. The duty cycle for second PWM cycle is written to OCxRS register.
3. PWM mode is enabled, OCx pin is driven low if OCxR is zero. OCx pin is driven high if OCxR is non-zero.
4. Timer is enabled and starts incrementing.
5. On a compare match, the OCx pin is driven low.
6. On timer rollover, OCxRS value is loaded into OCxR register. OCx pin is driven low if OCxR is zero. OCx pin is driven high if OCxR is non-zero.
7. The duty cycle for third PWM cycle is written to the OCxRS register.

Output Compare setup

- The following steps must be taken to configure the Output Compare module to control output pins OCx:
  - set the related timer with desired values
  - set OCx to be an output
  - set the Output Compare module to a desired operating mode

Output compare: coding example

- The following code sets up OC8 to work in PWM mode and be controlled by Timer 2. When operating, the OC8 pin will be set to high for 5ms every 40ms:

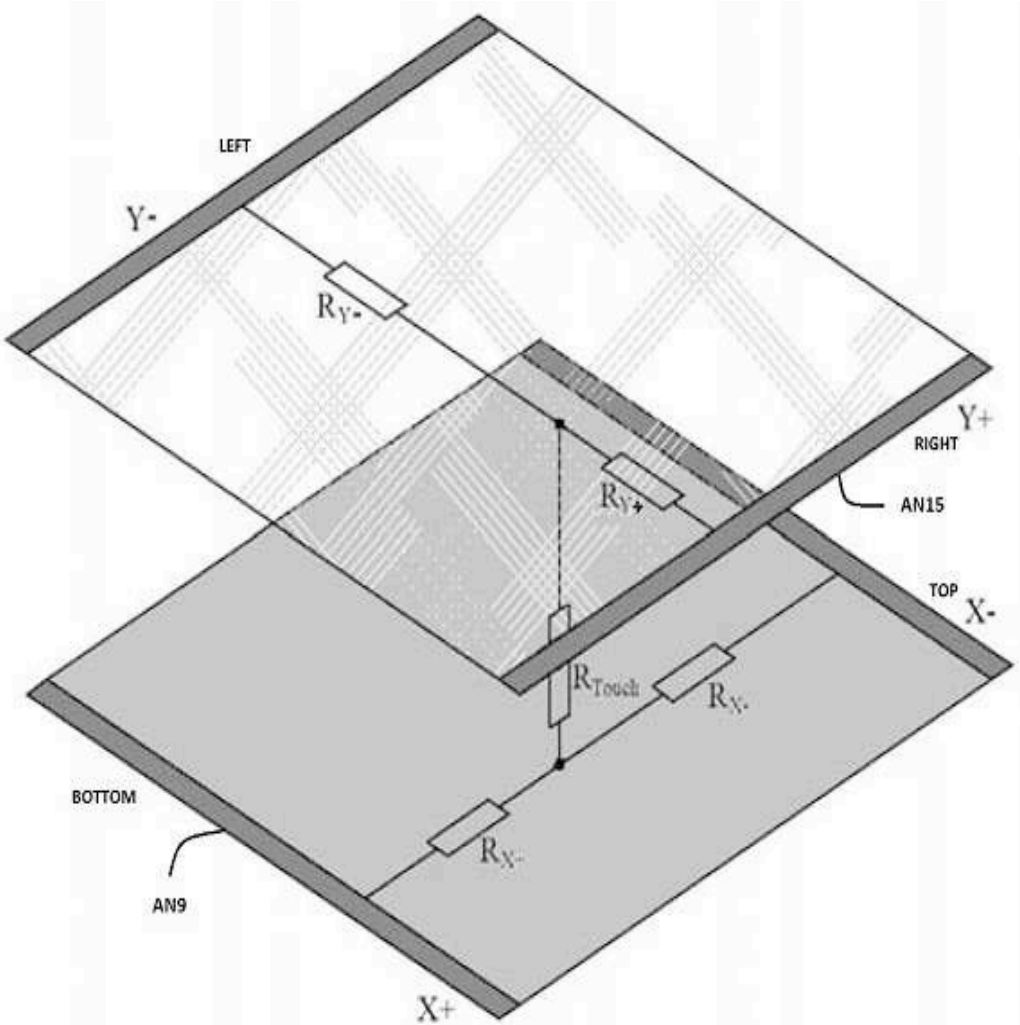
```
//setup Timer 2
CLEARBIT(T2CONbits.TON);          // Disable Timer
CLEARBIT(T2CONbits.TCS);          // Select internal instruction cycle clock
CLEARBIT(T2CONbits.TGATE);        // Disable Gated Timer mode
TMR2 = 0x00;                      // Clear timer register
T1CONbits.TCKPS = 0b10;           // Select 1:64 Prescaler
CLEARBIT(IFS0bits.T2IF);          // Clear Timer2 interrupt status flag
CLEARBIT(IEC0bits.T2IE);          // Disable Timer2 interrupt enable control bit
PR2 = 8000;                       // Set timer period 40ms:
                                  // 8000= .04 * 12.8*10^6 * 1/64

//setup OC8
CLEARBIT(TRISDbits.TRISD7);       // Set OC8 as output
OC8R = 1000;                      // Set the initial duty cycle to 5ms
OC8RS = 1000;                     // Load OCRS: next pwm duty cycle
OC8CON = 0x0006;                  // Set OC8: PWM, no fault check, Timer2
SETBIT(T2CONbits.TON);            // Turn Timer 2 on
```

Touchscreen

- The Amazing Ball System has a four-wire touchscreen. The touchscreen has two homogenous resistive layers.

- **The point of contact divides each layer in a series of resistor network with two resistors. By measuring the voltage at this point on X (or Y) axis, the user gets information about the position of the contact point along the Y (or X) voltage gradient.**



Touchscreen

- To get a complete set of coordinates, the voltage gradient must be applied once in vertical and then in horizontal direction: first a supply voltage must be applied to one layer and a measurement of the voltage across the other layer is performed, next the supply is instead connected to the other layer and the opposite layer voltage is measured.
- To measure the X-coordinate, the user must apply Vdd to X+, GRN to X-, set Y+ and Y- as high impedance (Hi-Z), and measure the signal from analog input pin AN15.
- To measure the Y-coordinate, the user must apply Vdd to Y+, GRN to Y-, set X+ and X- as high impedance (Hi-Z), and measure the signal from analog input pin AN9.
- There is a circuit designed to control the touchscreen measurement. This circuit is controlled through three I/O pins E1, E2, E3.
- Note that it takes about 10ms for the touchscreen output signal to be stable when the touchscreen is switched from one operation mode to another.

	Bottom (X+)	**Top (X-)**	**Right (Y+)**	**Left (Y-)**
Standby	Hi-Z	Hi-Z	Hi-Z	GND
Read X-coordinate	Vdd	GRN	Hi-Z (AN15)	Hi-Z
Read Y-coordinate	Hi-Z (AN9)	Hi-Z	Vdd	GND

Coding example for touchscreen operation

- **Quiz**: What lines of code will allow to read touchscreen X-coordinate by using ADC analog input pin AN15?

	Bottom (X+)	**Top (X-)**	**Right (Y+)**	**Left (Y-)**
E1=1	Hi-Z	Hi-Z	No effect	No effect
E1=0	Vdd	GRN	No effect	No effect
E2=1	No effect	No effect	Hi-Z	No effect
E2=0	No effect	No effect	Vdd	No effect
E3=1	No effect	No effect	No effect	Hi-Z
E3=0	No effect	No effect	No effect	GRN

Coding example for touchscreen operation

- **Quiz**: What lines of code will allow to read touchscreen X-coordinate by using ADC analog input pin AN15?

  - //set up the I/O pins E1, E2, E3 to be output pins
  - CLEARBIT(TRISEbits.TRISE1); // I/O pin set to output
  - CLEARBIT(TRISEbits.TRISE2); // I/O pin set to output
  - CLEARBIT(TRISEbits.TRISE3); // I/O pin set to output

  - //set up the I/O pins E1, E2, E3 so that the touchscreen X-coordinate pin is read
  - CLEARBIT(PORTEbits.RE1);
  - SETBIT(PORTEbits.RE2);
  - SETBIT(PORTEbits.RE3);

	Bottom (X+)	**Top (X-)**	**Right (Y+)**	**Left (Y-)**
E1=1	Hi-Z	Hi-Z	No effect	No effect
E1=0	Vdd	GRN	No effect	No effect
E2=1	No effect	No effect	Hi-Z	No effect
E2=0	No effect	No effect	Vdd	No effect
E3=1	No effect	No effect	No effect	Hi-Z
E3=0	No effect	No effect	No effect	GRN