Embedded Software Development

Assoc. Prof. Uluç Saranlı
Dept. of Computer Engineering

ORTA DOĞU TEKNİK ÜNİVERSİTESİ
MIDDLE EAST TECHNICAL UNIVERSITY

Embedded Systems

- What exactly is an “Embedded System”?
  - Any device that contains one or more **dedicated** computers, microprocessors, or microcontrollers
- Where can I find one?
  - Everywhere!








home appliances

printers&copiers

toy industry

robotics

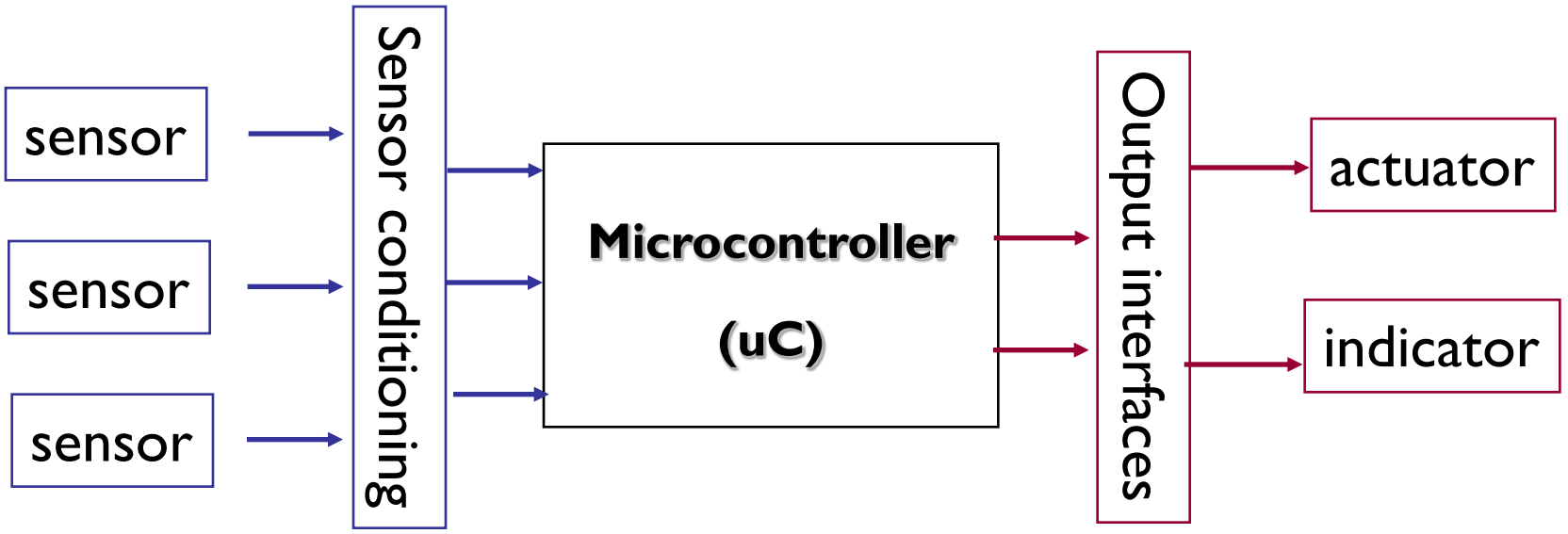
aircrafts

automotive

personal gadgetry

factory coordination

Embedded system: General structure



- Intense interaction with the external world
- Sensors convert physical quantities into electrical signals
- Microcontroller acquires data, processes it and issues outputs
- Actuators convert output signals into physical work

Embedded Systems: Some Goals

- Small code size, fast execution
  - Do not rely too much on external libraries
  - Be careful on how much memory/computation is used
- Predictability of execution (determinism)
  - Periodicity of sensor readings and actuator commands
  - Finite and predictable "response time" to important events
- Software modularity
  - Allows incremental development without breaking previous components
  - Allows teamwork

Relevant Microcontroller Features

- Ports: Used for digital and analog I/O
  - Group of pins on a microcontroller which can be accessed simultaneously (in parallel). Physically, simple registers.
- Interrupts
  - Allows immediate response to external signals/events
- Timers/Counters
  - Up/down counters connected to either the MCU clock or external signals
- A/D Converters
  - Allow converting an external voltage input to a digital value

Consult your MCU datasheet for ALL details

Some Recommendations

- Use Arduino/Google/StackOverflow/... responsibly, they are useful but dangerous resources. Be an ENGINEER, come up with your own solutions.
- DO NOT just copy/paste existing solutions without understanding. If you do that, you will most likely get stuck in your next step.
- If you are using Arduino, look inside its source code, understand what it does. Otherwise, you will be wasting LOTS of time chasing errors for which you have no idea where they might be coming from.
- RTFM (i.e. the datasheet for your platform).

Basic Embedded I/O Programming

- Simplest form for embedded code is a simple loop, called Round-Robin (or Cyclic Executive):

```
while (1) {
        task_1();
        ...
        task_n();
}
```

- Executes n tasks in sequence, within an infinite loop that never exits.
- The Arduino library has this structure. Check out /hardware/arduino/avr/cores/arduino/main.cpp

Basic Embedded I/O Programming

- Main entry point in the Arduino library:

```
int main(void) {
    init();      // To be written by the user
    setup();     // To be written by the user
    for (;;) {
        loop(); // To be written by the user
    }
    return 0;
}
```

Round-Robin Architecture

- Embedded software is often "event driven"
  - An "external" event triggers software to respond (e.g. a digital input went from 0 to 1)
  - An action is taken, which might trigger additional events
  - When no events are present, software simply waits, possibly putting the CPU into sleep mode

- In the Round-Robin architecture
  - Each task should "listen" to a specific set of events through polling (i.e. explicit check with an if statement)
  - When a task notices an event, it responds by doing whatever is necessary, which might result in other events

Round-Robin example

- Simple example: Use a button to toggle an LED
- Goal: Write a program that will monitor a button connected to Port B bit 0 and when it becomes true, toggle an LED connected to Port B pin 1

- Two tasks:
  - Button task: Monitor the button state
  - LED task: Toggle the LED when the button tasks says so

Warning: Our examples here use the PIC microcontroller. Principles are the same, details will differ for other MCU choices. Read the datasheet.

Main Program

```
unsigned char toggle_flag;     /* Tells led_task() to toggle LED */
unsigned char button_state;    /* Remembers previous state of the button */

void init_system() {
        toggle_flag = 0;         /* Initialize global variables. */
        button_state = 0;
        TRISB = 0xFD;            /* Pin 1 output, rest is input (for PIC)*/
}
void main() {
        init_system();
        while(1) {               /* Here is the Round-Robin structure */
                button_task();
                led_task();
        }
}
```

Button Task

```
void button_task() {
        switch (button_state) {
        case 0:  /* Previous button state was 0 */
                if  (PORTB & 0x01 != 0) {
                        button_state = 1;
                        toggle_flag = 1;
                }
                break;
        case 1: /* Previous button state was 1 */
                if  (PORTB & 0x01 == 0) button_state = 0;
                break;
        }
}
```

- Implements a "State Machine". More on this later.

LED Task

```
void led_task() {
        if (toggle_flag != 0) {
                PORTB = PORTB ^ 0x02;
                toggle_flag = 0;
        }
}
```

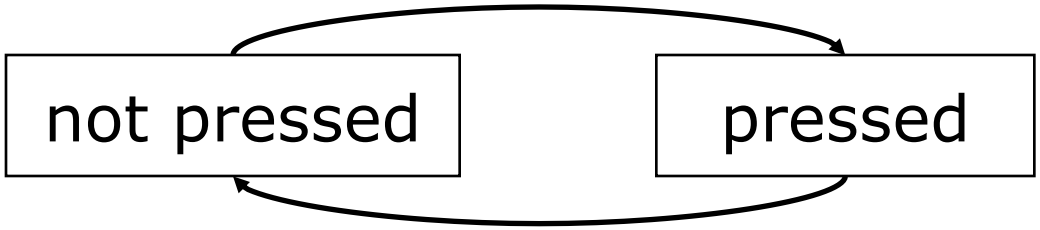
- Monitors the toggle flag, performs the requested toggle if detected.
- Event sources in this design:
  - Button connected to Port B (externally triggered)
  - toggle_flag variable (inter-task communication)

Timeline of Events

- Nothing happens until the button is pressed. If statements in both tasks are false
- When the button is pressed, the button task switches to state 1, and alerts the LED task through the toggle flag
- The LED task toggles the output and resets the flag.
- Once again, nothing happens until the next event, which is the button release!
- When the button is released, the button task switches to state 0, and starts waiting for another press.

Observations: State Machines

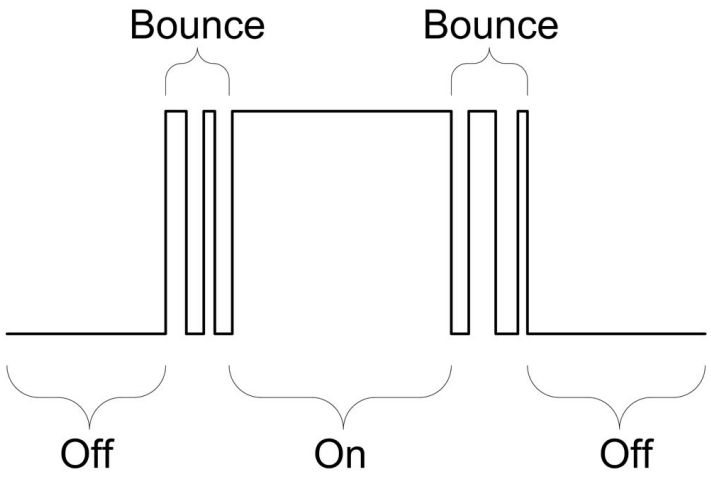
- The button task adopts what is called a "state machine" structure. It remembers past events through a carefully selected set of states, implemented in a switch case statement



- This represents an internal state, that attempts to track what the physical state of the button looks like
- State machines are essential to embedded software design. Use them!

Observations: Button Behavior

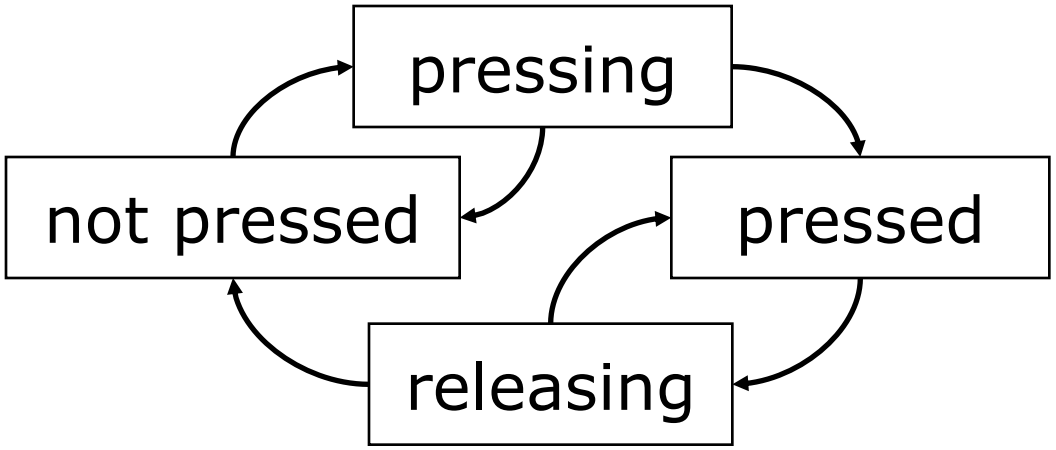
- Normally, buttons do not behave this nicely. The "bounce", generating multiple pulses



- Additional logic and states are needed in the button task for "debouncing", which involves waiting for the bounce to end.

Observations: Button Behavior

- For example, a debouncing button task might have a more complex state machine:



- "Pressing" and "releasing" states should wait for a short period for bouncing to end.
- WARNING: Waiting here does not mean another while loop!

Avoiding Busy Waits

- Ideally, the only busy wait (i.e. waiting for something to happen in a while loop) should be the while(1) in main().
- All other waiting should take the form of counters, or monitoring of timers (more on timers later)

```
void button_task() {
        ...
        case PRESSING:
                if (++wait_counter > DELAY_COUNT)
                        button_state = PRESSED;
                break;
        case PRESSED:
                ...
}
```

Avoiding Busy Waits

- Eliminating all busy waits except main() greatly helps debugging
  - When something is stuck, you know it's because no events are received
  - You can still use your debugger to single-step through all tasks, inspecting their internal states
- Even when a single task is "stuck" in some undesired state, remaining tasks will keep working
- This is similar to the difference between a preemptible and a non-preemptible operating system.

Round-Robin: Flaws

- Unpredictable execution
  - In the worst case, a particular task needs to wait for all others to finish before it can respond to an event
  - Adding a new task changes all timing related considerations: Task response is "coupled" to other tasks
- Solution: Use interrupts to quickly respond to important external events
  - Allows prioritizing certain events above others by allowing immediate response by the Interrupt Service Routine (ISR)
  - Adding new tasks does not effect the deployment of ISR execution

Recall: Interrupts

- An **interrupt** is a (temporary) break in the flow of execution of a program
  - the CPU is said to be “interrupted”

- When an interrupt occurs, the CPU deals with the interruption, then carries on where it was left off
  - Program segment dealing with the interrupt is called the "Interrupt Service Routine", ISR.
  - ISR programmer should put the CPU back to its beginning state before returning.

Basic Concepts of Interrupts

- Interrupts are (generally) used to overlap computation & input/output tasks
  - Rather than explicitly waiting for I/O to finish, the CPU can attend other tasks
  - Examples would be console I/O, printer output, and disk accesses, all of which are slow processes

- Normally handled by the OS. Rarely coded by programmers under Unix, Linux or Windows.
  - In embedded and real-time systems, however, they are part of the normal programming work.

Basic Concepts of Interrupts

- Why interrupts over polling? Because polling
  - Ties up the CPU in one activity
  - Uses cycles that could be used more effectively
  - Code can't be any faster than the tightest polling loop

- Bottom line: an interrupt is an asynchronous subroutine call (triggered by a hardware event) that saves both the return address and the system status
  - The main difference from a subroutine is that the main routine is unaware that it has been interrupted.

Extending the Round-Robin Loop

- Round-Robin with Interrupts
  - Keeps the same infinite loop structure
  - Uses interrupts to respond to "asynchronous" events quickly, informing tasks in the main loop

```
void interrupt event1_isr() { ... } // Depends on the MCU
void interrupt event2_isr() { ... } // Depends on the MCU

main() {
    initialize();
    while (1) {
          task_1();
          ...
          task_n();
    }
}
```

Round-Robin with Interrupts

- Before, each task was explicitly polling for events
  - If there are too many tasks, a short event might be detected late, or even missed
  - Multiple, successive occurrences of the same event might go undetected

- Now, the ISR immediately responds to the event
  - Performs all time-critical tasks to do
  - Alerts the main routine through a flag or buffers for less time-critical tasks

Simple Example with Interrupts

- Same as before: Use a button to toggle an LED
- Previous design:
  - button_task monitors the button on Port B pin 0
  - led_task controls the LED on Port B pin 1

- New design:
  - RB0 pin is an external interrupt source (for the PIC MCU)! We can detect the rising edge with an interrupt!
  - So, replace the button_task with an ISR
  - Keep the LED task as before

Simple Example with Interrupts

```
unsigned char toggle_flag;

void init_interrupts() {
        /* Enable INT0 source */
        INTCON = 0x10;

        /* Once all configuration is done,
           enable all interrupts */
        INTCONbits.GIE = 1;
}
void init_ports() {
        TRISB = 0xFD;
}
void main() {
        toggle_flag = 0;
        init_ports();
        init_interrupts();
        while(1) {
                led_task();
        }
}
```

```
void led_task() {

        if (toggle_flag != 0) {
                PORTB = PORTB ^ 0x02;
                toggle_flag = 0;
        }

}

void interrupt high_priority high_isr() {

        /* Check flags to detect which source
           triggered the interrupt
        if (INTCONbits.INT0IF) {
                /* Clear interrupt flag. */
                INTCONbits.INT0IF = 0;

                /* Inform LED task */
                toggle_flag = 1;
        }

}
```

More Complex Example

- Monitor a digital input on Port B pin 0, when it is pressed:
  - read the 8 bit value from PORTC
  - send it serially, one bit at a time from PORTD pin 0
  - ensuring that there is at least 50ms between each bit.
- Simple possible solution:

```
main() {
      unsigned char value, count;
      initialize();
      while (1) {
        if (PORTB & 0x01 != 0) {
          value = PORTC;
          for (count = 0; count < 8; count++) {
                if (value & (0x01 << count))          PORTD |= 0x01;
                else          PORTD &= 0xfe;
                busy_wait(50);
      } } }
}
```

Problems with the Simple Solution

- Continuously reading Port B pin 0 is problematic, possibly triggering multiple reads of PORTC on one button press
  - That's easy, use state machines just like before
- What if Port B pin 0 is pressed again while we were sending data on PORTD?
  - That's why we need to avoid busy waits. They prevent other parts of the code from running
  - Also, if RB0 is pressed multiple times before we have a chance to send all data, we need to buffer data!
- In any case, we better detect that RB0 is pressed quickly and buffer the reading of PORTC

Studying The Problem

- Event sources:
  - RB0 is pressed (external INT0 interrupt can be used)
  - Sending of the current bit completed (i.e. 50ms elapsed)
  - Sending of the current byte completed (all 8 bits done)
- An alternative design
  - INT0 interrupt service routine reads PORTB and places the data in a buffer (FIFO, ring buffer)
  - send_task() monitors the buffer, starts sending when a new byte arrives. We should use state machines for this.
  - When a bit is written to PORTD, send_task() holds off for 50ms until the next bit.
- This is a typical use of round-robin with interrupts

Implementation

- Initialization and main

```
void init_interrupts() {
    /* Do whatever it takes to initialize interrupts for your MCU */
}
void init_ports() {
    TRISB = 0xFF;           /* PORTB is all inputs */
    TRISC = 0xFF;           /* PORTC is all inputs */
    TRISD = 0xFE;           /* PORTD pin0 is an output */
}
main() {
    init_ports();
    init_interrupts();
    INTCONbits.GIE = 1;     /* Enable all interrupts

    while (1) {
            send_task();
    }
}
```

Ring Buffers

- Useful for buffering incoming/outgoing data

```
#define BUFSIZE 16                      /* Static buffer size. Maximum amount of data */
unsigned char buffer[BUFSIZE];          /* No malloc's in embedded code! */
unsigned char head = 0, tail = 0;       /* head for pushing, tail for popping */

bool buf_isempty() { return (head == tail); }

void buf_push( unsigned char v ) {      /* Place new data in buffer */
      buffer[head++] = v;
      if (head == BUFSIZE) head = 0;
      if (head == tail) { /* Overflow!!! */ error(); }
}

unsigned char buf_pop() {               /* Retrieve data from buffer */
      unsigned char v;
      if (buf_isempty()) { /* Underflow!! */ error(); return 0; }  else {
            v = buffer[tail++];
            if (tail == BUFSIZE) tail = 0;
            return v;
      }
}
```

External INT0 ISR

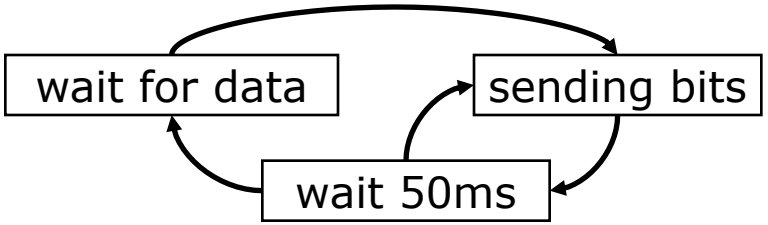
- Main function is to read PORTB and buffer data

```
void interrupt high_priority high_isr() {
    unsigned char v;

    if (INTCONbits.INT0IF) {            /* Check flags to make identify source */

        INTCONbits.INT0IF = 0;          /* Clear interrupt flag */

        v = PORTB;                      /* Read data */
        buf_push( v );                  /* Push into buffer */
    }
}                                       /* Done! */
```

- The ISR is short! It performs only the most urgent task, which is reading and storing data

Tasks for Sending Data

- Once again, use state machines




```
unsigned char send_state = 0;
void send_task() {
    unsigned char sending, send_count;
    switch (send_state) {
    case 0:                                  /* Wait for new data */
        disable_int0();                      /* IMPORTANT: Prevents corruption of buffer! */
        if (!buf_isempty()) {
           sending = buf_pop();
           send_state = 1; send_count = 0; }
        enable_int0();                       /* Re-enable interrupts to resume operation */
        break;
    case 1:                                  /* Send next bit */
        if ( (sending & (0x01 << send_count) != 0)      PORTD = PORTD | 0x01;
        else                                           PORTD = PORTD & ~0x01;
        send_count++; send_state = 2; break;
    case 2:                                  /* Skip until 50ms elapses */
        if (time_since_laststate() < 50) break;
        if (send_count == 8) { send_state = 0; break; }
        send_state = 1;  break;
    }
}
```

Observations

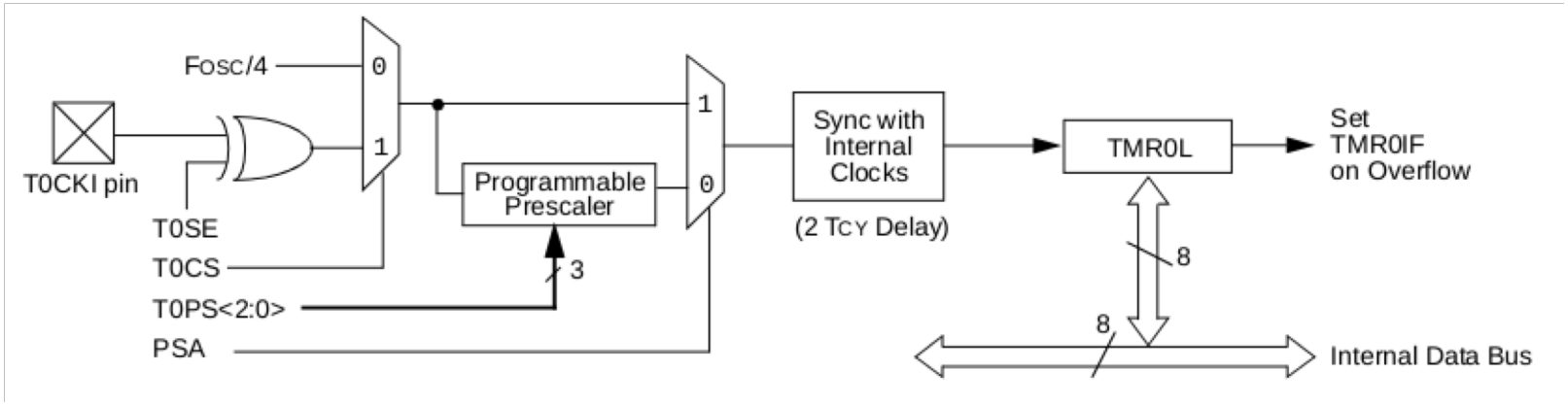
- No busy waits at all!
- Button press on RB0 immediately results in reading of data on PORTC
- 50ms waiting is accomplished by send_task() staying in state 2 until 50ms elapses since last state transition
  - we can use timers for this, stay tuned...
- Ring buffers are extremely useful for communicating between ISRs and tasks
- Data Sharing: Disabling/enabling interrupts in send_task() to prevent buffer corruption

Counters/Timers

- All Microcontrollers include multiple timers and counters that can be used in a variety of ways
- Physically, timers are registers continually increasing (or decreasing) and then starting over following an overflow
  - 0, 1, 2, 3, 4 ..... 255, 0, 1, 2, 3, 4 ..... etc.
- Their clock inputs can be chosen from a variety of different options (depends on the MCU)
- !!! Timers and counters operate independently from the microcontroller's program execution. No CPU effort is required for their normal operation.

Application: Event Counters

- Timer/counter registers can usually be used as an event counter.
  - If the clock of this register is replaced by a signal coming from an external event (this can be configured by using methods specific to each MCU)
  - We can obtain the number if times the event has occurred.
  - For example, on the PIC microcontroller Timer 0, the clock circuit looks like:



Event Counter Example

- A microcontroller monitoring the capacity of a parking lot
  - Whenever a car enters the lot, the counter is incremented
  - When a car leaves, the counter is decremented
  - When the count reaches the maximum number of cars allowed for that lot, it can display a sign “Parking Lot Full”

- Also, digital clocks, traffic light controllers, timers in microwave owens.

Timers

- A timer is a counter that is always enabled, with a time-periodic input fed to the clock input
- It counts the number of cycles on its clock input
- Time is calculated by
  - subtracting beginning count from the ending count and
  - multiplying the difference by the clock period.

Observations

- Incrementation is done in the background by the microcontroller hardware, not in software
- It is up to programmer to think up a way how (s)he will take advantage of these tools for their needs.
- Example: Increasing some variable on each timer overflow.
  - If we know how much time a timer needs to make one complete round, then multiplying the value of a variable by that time will yield the total amount of elapsed time.

Timers/Event Counters

- Uses of Programmable Timers and Event Counters
  - Count external events
  - Generate event caused interrupts

  - Generating real-time interrupts
  - Outputting precisely timed signals
  - Programmable baud rate generation
  - Measuring time between events
  - Measuring frequency of signals
  - Generating waveforms (e.g. sound)
  - Establishing a time base (multi tasking systems, sampling systems, etc.)

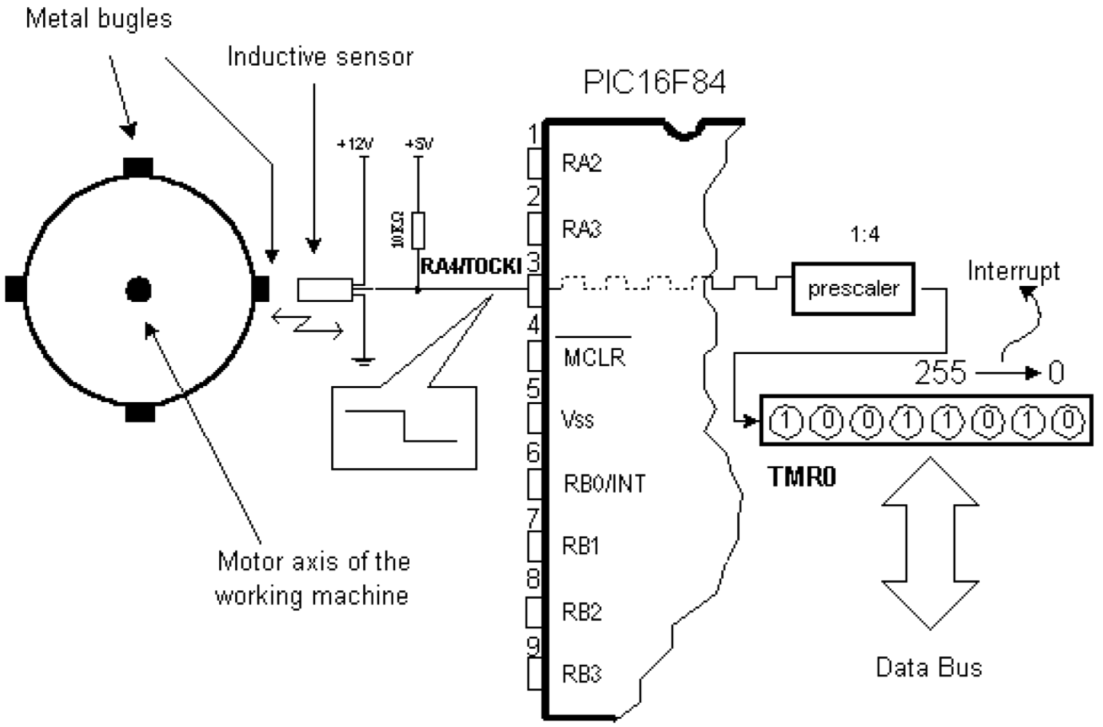
Example 1: Tracking Motor Angle

- A typical application solved using an external input clock and a timer is counting full turns of a motor shaft.
- Let's attach four metal screws on the axis of a motor.
- Then place an inductive sensor at a distance of 5mm from the head of a screw

Example 1: Setup

- The inductive sensor will generate a falling signal edge every time the head of the screw is in front of the sensor head.
- Each signal will represent one fourth of a full turn. We would like the sum of all full turns to be found in the timer register
- The main program can then easily read this data from the timer register through the data bus.

Example 1: Connections



Determining a number of full axis turns of the motor

Example 1: Outline of the Software

- Required steps (probably valid for most microcontrollers):
  - Configure the clock input to the timer to come from the external pin
  - Configure the "prescaler" for the timer to choose how many input pulses will cause an increment
  - Configure the timer to generate an interrupt when the timer "overflows"
  - Use a round-robin with interrupts architecture to monitor the timer value, which will keep track of how many rotations the disk has gone through

Example 2: Periodic interrupt

- Universally used in all operating systems to establish a consistent and reliable time basis
  - Interrupt is generated periodically every, say, 1ms
  - Updates a system clock (time-of-day)
  - Invokes the OS scheduler to switch processes
  - Performs any other tasks requiring periodicity

- Embedded software, particularly real-time systems, almost always needs such a regular, reliable interrupt as well

Example 2: Setup

- Goal: Generate an interrupt once every 10ms
- Oscillator frequency is 20MHz
- Which timer is appropriate?
  - 20/4 = 5MHz system clock
  - 10ms = 10000us = 50000 clock cycles
- Suppose we use Timer0 in 8-bit mode
  - 1:128 prescaler yields 390.625 counts per 10ms, or 6.553ms between two overflows
  - 1:256 prescaler yields 195.3125 counts per 10ms, or 13.1072ms between two overflows. Enough?
  - Please consult the MCU datasheet for details. These are for the PIC microcontroller

Example 2: Setup

- Idea:
  - What if we start the timer value not from zero, but from an initial value of our choice?
  - The counter will then reach 255 and overflow earlier, achieving a shorter period

- So, if we want to overflow after counting up by 195, we need to preload the timer with 256-195=61
  - This needs to be repeated after every overflow, preferably as the very first instruction in the Timer0 ISR.
  - This will still not yield exactly 10ms between two interrupts. Why?

Example 2: Putting it together

- Initialization and main

```
void init_interrupts() {
      RCONbits.IPEN = 0;            /* Disable priorities */
      INTCON = 0x20;                /* Enable timer 0 interrupt( */
}
void init_ports() {
      TRISC = 0xFE;                 /* PORTC pin 0 is an output */
}
void init_timer() {
      T0CON = 0xA0;
      TMR0L = 0xB0; TMR0H = 0x3C;   /* Is this ok? Any potential problems?*/
}
main() {
      init_ports();
      init_interrupts();
      init_timer();
      INTCONbits.GIE = 1;           /* Enable all interrupts

      while (1) {
      }
}
```

Example 3: Variable delays

- Task:

  Once every 10ms, output a 100us pulse, wait for 1ms, output another 100us pulse, wait for 2ms and output another 100us pulse

- Design:
  - Use the Timer 0 to generate a periodic interrupt once every 10ms
  - Use Timer 1 to implement a configurable delay. A main loop task will setup Timer 1 and wait for a flag to be set by the Timer 1 ISR.

Example 2: Putting it together

```
unsigned int cycle_count = 0;

void interrupt high_priority high_isr() {

    if (INTCONbits.TMR0IF) {            /* Check flags to make identify source */

        INTCONbits.TMR0IF = 0;                  /* Clear interrupt flag */

        TMR0L = 0xB0; TMR0H = 0x3C;             /* Reload timer with initial value*/

        cycle_count++;

        LATC = LATC ^ 0x01;                     /* Toggle PORTC bit 0 */
    }
}                                               /* Done! */
```

Example 3: Variable delays

- Observations:
  - Timer 0 and Timer 1 will run independently, without affecting each other's operation
  - Your ISR will need to handle both interrupts

```
void interrupt high_priority high_isr() {
    if (INTCONbits.TMR0IF)              Timer0_ISR();
    if (PIR1bits.TMR1IF)  Timer1_ISR();
}
```

  - Utility function for configuring Timer1 and starting it.

```
void configure_delay( int ms ); /* Sets up Timer1 for ms microseconds */
```

  - You can stop Timer 1 when you no longer need the delay

```
T1CONbits.T1RUN = 0;
```

- Left as an exercise for your project :)