

MicroORE Library

A heartbeat LED running in the background

The heartbeat LED ("LED1") toggles on and off every second (1000 ms).

Timer1 generates an interrupt every millisecond and decrements a counter ("heartbeat").

When "heartbeat" reaches zero, "LED1" is toggled and "heartbeat" is reset to 1000.

Here is how Timer1 generates an interrupt every 1 ms:

The internal oscillator selected by the config bits runs at 7.370 MHz. I'm assuming PLL4 has been defined.

PLL4 multiplies this by 4, so $F_{osc} = 4 \times 7.370 \text{ MHz}$. The instruction cycle, $F_{cy} = F_{osc}/4 = 7.370 \text{ MHz}$.

initTimer1() sets up Timer1 to be clocked by the instruction cycle, then prescaled by 1:256.

Therefore, Timer1 increments every $1 / (7.370 \text{ MHz} / 256) = 34.7 \text{ us}$.

Therefore, $1 \text{ ms} / 34.7 \text{ us} = 29$ Timer1 ticks are required to count off 1.007327 ms. (Close enough!)

The interrupt handler for Timer1 does the following:

```
    heartbeat--;           // Decrement the counter
    if (heartbeat == 0) {  // Only when "heartbeat" reaches zero do we . . .
        LED1 ^= 1;        // Toggle LED1, and . . .
        heartbeat = 1000; // Reset the counter.
    }
```

A simple Delay routine

The interrupt handler for Timer1 also increments a variable called "systemTime".

"systemTime" counts milliseconds. delay_ms() uses it like this:

```
void delay_ms(unsigned int duration)
// *****
//     Waits for "duration" number of milliseconds.
//     Timer1 and its interrupt must be enabled.
// *****
{
    unsigned long endTime=0;           // Declare a local variable.
    endTime = systemTime + duration;   // Set "endTime" to the current time ("systemTime") plus the
    amount of time you want to delay.
    while (systemTime <= endTime);    // Spin in a "while" loop until the "systemTime" equals
    "endTime", then return from this procedure.
} // End of delay_ms()
```

Setup the ADC subsystem to scan AN0..AN3 and AN6 continuously in the background

initADC() scans AN0..AN3 and AN6, puts the readings into ADCBUF0..ADCBUF4, then generates an interrupt.

Note that the readings go into 5 contiguous buffers, NOT ADCBUF0..ADCBUF3 and ADCBUF6!

initADC() sets up the ADC subsystem as follows:

ADCON1 = 0b0000000011100100 turns the ADC subsystem off, continuous operation if the PIC goes into idle mode, the readings will be formatted as unsigned integers, automatic conversion following the sampling period.

ADCON2 = 0b0000010000010000 sampling on channel 0 only, scan inputs specified by ADCSSL, interrupt after converting the 5th sample.

ADCON3 = 0b0001111100111111 sample duration = 31 x tAD, conversion clock = 32 x Tcy.

ADCHS = 0b00000000100000001 CH0 negative input will be Vref- and CH0 positive input will be AN1.

ADCSSL = 0b0000000001001111 AN0..AN3 and AN6 will be scanned.

ADCON1bits.ADON = 0b1 turns the ADC subsystem on. (Bit 15 in ADCON1 is called ADON)

After initADC() is called and ADC interrupts are enabled (IEC0bits.ADIE = 1), the voltage on AN0..AN3 and AN6 can be read whenever you like by simply reading ADCBUF0..ADCBUF4. The buffers are continuously updated in the background.

Driving Servos

servoControl(servo, position) can move the specified servo to a specified position. Position is specified as -100% .. 0% .. +100%.

-100% corresponds to the full counterclockwise position; 0% to the middle position and +100% to the full clockwise position.

Servos can be connected to J13 and J14.

The servos are driven by the output compare modules, OC1 and OC2. The OC modules use Timer3 as a time base.

See section "Dual Compare Mode: Continuous Output Pulses" in the following document for a detailed description:

<http://ww1.microchip.com/downloads/en/DeviceDoc/70061D.pdf>

When PLL4 is defined, Timer3 is setup to increment every 8.68 us.

SERVO_RATE is defined as 2304 ticks of Timer3 because 20 ms / 8.68 us = 2304. (20 ms is the required repetition rate for a servo.)

SERVO_FULL_CCW (1.0 ms pulse), SERVO_MIDDLE (1.5 ms pulse) and SERVO_FULL_CW (2.0 ms pulse) are defined similarly.

Note: You may want to fine tune these values for the PLLx you have chosen. As they are defined in the library, the servo does not move over its full range, not quite :)

PWM Motor Control

`void motorControl(motor, direction, dutyCycle)` sets the direction (FWD, BWD) and PWM duty cycle (0% .. 100%) for the motor specified. For the motors to operate at all, `MOTOR_ENABLE` (RC13) must be TRUE (ie. logic high). Note, since RC13 drives LED3 and `MOTOR_ENABLE`, you must not manipulate LED3 while driving the motors. MicroORE hardware is configured to drive the H-bridges using the locked antiphase mode of operation (as opposed to signed magnitude). `motorControl()` hides the details. You simply specify the direction, FWD or BWD, and how fast you want to go (0% to 100%), and which motor (A, B or BOTH).

See section "Pulse-Width Modulation Mode" in the following document for a detailed description:
<http://ww1.microchip.com/downloads/en/DeviceDoc/70061D.pdf>

Note: `initOC()` requires a parameter (`SERVO_MODE` or `PWM_MODE`) to tell it to setup for servo operation or PWM operation.

Also, you may want to change the PWM frequency. It is currently 3600 Hz.

PING))) Sonar Rangefinder Control

PING))) is a "stupid" ultrasonic rangefinder from Parallax, Inc. It requires an external trigger to generate a sonic pulse and an external microcontroller to measure the sonic pulse flight time. `pingControl()` generates a 10 us trigger by toggling RD0. Input capture 1 interrupts when it detects the rising edge from the PING))) which indicates the start of the sonic pulse. The interrupt handler records the time as indicated by Timer2, then sets IC1 to interrupt when a falling edge is detected. When the PING))) detects the returning sonic pulse, it pulls its signal pin low. The interrupt handler records the time as indicated by Timer2. The time difference is used to calculate the distance. `pingControl()` can return the distance in centimetres (CM) or the number of Timer2 ticks (RAW). If `pingControl()` returns zero, then it timed out waiting for either the rising edge or the falling edge.

Quadrature Encoder Interface

Channel A of the encoder must be connected to J9 because J9 goes to INT0 of the microcontroller.

Channel B of the encoder must be connected to J11 which goes to RC14 of the microcontroller.

`initQEI()` sets up INT0 to interrupt on a positive edge, ie. low-to-high transition.

The INT0 interrupt handler increments "encoderTicks". This is used as an indication of the speed of the wheel. The handler then samples channel B. If channel B is low, then "direction" is set to 0 which means forward, otherwise it is set high which means backward.

The Timer1 interrupt handler has also been modified. "motorLoop" is decremented. When it is zero, the number of encoder ticks is captured in "motorSpeed", "motorSpeed" is reset and "encoderTicks" is zeroed ready to count again when channel A interrupts INT0.

In `main()`, `motorLoop` is initialized to 100, so "motorSpeed" is captured every 100 ms. The quadrature encoder test in `main()` sets a motor speed, lets it run for a second, then prints "motorSpeed" and "direction".