Synchronicity: A wood gear clock with a unique drive mechanism


This software was developed using Texas instrument's Code Composer Studio Version: 5.1.0.09000. The MSP430 microcontroller was configured using Grace, TI's Graphical Peripheral Configuration Tool. The first section contains screenshots of the Grace configuration. The second section contains the source code listing.
Synchronicity: A wood gear clock with a unique drive mechanism

Note 1: Grace interrupt handlers are names of user-provided functions. Manual mode requires no arguments but requires a return value, e.g. unsigned short interruptHandler(void). All other modes require no arguments and return value, e.g., void interruptHandler(void).

Note 2: Manually configuring the DCO frequency can result in a +/-10% frequency deviation. The Pre-calibrated DCO has a tolerance of +/-3% frequency deviation. See datasheet for more information.

Note 3: Set a delay value in milliseconds based on the system rise time to ensure no violation of VCC vs MCLK. It is highly recommended when setting a non-default system clock frequency to ensure a proper system start-up.
Synchronicity: A wood gear clock with a unique drive mechanism
Synchronicity: A wood gear clock with a unique drive mechanism

---

**GPIO - Port 1 / Port 2 - Register Controls**

**PORT 1**
- **Output Register**
- **Direction Register**
- **Interrupt Flag Register**
- **Interrupt Edge Select Register**
- **Interrupt Enable Register**
- **Port Select Register**
- **Resistor Enable Register**
- **Interrupt Handler**
  - After Interrupt: **Do Not Change Operating Mode**

**PORT 2**
- **Output Register**
- **Direction Register**
- **Interrupt Flag Register**
- **Interrupt Edge Select Register**
- **Interrupt Enable Register**
- **Port Select Register**
- **Resistor Enable Register**
- **Interrupt Handler**
  - After Interrupt: **Do Not Change Operating Mode**

*Note: Grace interrupt handlers are names of user-provided functions. Manual mode requires no arguments but requires a return value, e.g., unsigned short interrupthandler(void). All other modes require no arguments and return value, e.g., void interrupthandler.*
Synchronicity: A wood gear clock with a unique drive mechanism

Comparator_A+ - Power User Mode

Inputs

PL1/CA1

Short inputs

PL2/CA2

Flip Inputs, Inverse Output

Output

Enable Filter

Timer_A CC1:B

PL7/CAOUT

Voltage Reference

- Channel

Reference Off

Enable Comparator Interrupt

Interrupt Edge Select

- Rising Edge

- Falling Edge

Interrupt Handler: ComparatorAISRHandler

After Interrupt: Do Not Change Operating Mode

Note: Grace interrupt handlers are names of user-provided functions. Manual mode requires no arguments but requires a return value, e.g., unsigned short interruptHandler(void). All other modes require no arguments and return value, e.g., void interruptHandler(void).
Synchronicity: A wood gear clock with a unique drive mechanism

**WDT+ - Basic User Mode**

- Overview
- Basic User
- Power User
- Registers

**WDT+ Mode Select**

- Stop Watchdog Timer
- Interval Timer Mode
- Watchdog Timer Mode

Enable Watchdog Timer Interrupt

**Interrupt Handler:** [ ]

**After Interrupt:** [ ]

**Note:** Grace interrupt handlers are names of user-provided functions. Manual mode requires no arguments but requires a return value, e.g., unsigned short interruptHandler(void). All other modes require no arguments and return value, e.g., void interruptHandler(void).
Synchronicity: A wood gear clock with a unique drive mechanism

Timer_A2 - 16-bit Timer - Power User Mode - CCR0

Clock Source

Divide

32.768 kHz

16-bit Timer/Counter

Clear

Counting Mode

Stop Mode

Up Mode

Continuous Mode

Up/Down Mode

Enable Timer Overflow Interrupt

Interrupt Handler: TimerAOverflowISRHandler

After Interrupt: Do Not Change Operating Mode

Timer Capture/Compare Block #0

Desired Timer Period: 1999.999 ms

Capture Register: 0

Clock Ticks

Time() Frequency: 0.5 Hz

Input Selection

Capture Mode

Mode

Output Pins

Note: Grace interrupt handlers are names of user-provided functions. Manual mode requires no arguments but requires a return value, e.g., unsigned short interruptHandler(void). All other modes require no arguments and return value, e.g., void interruptHandler(void).
Synchronicity: A wood gear clock with a unique drive mechanism

**Timer Capture/Compare Block #1**

- Desired PWM Duty Cycle: 0%
- Capture Register: 32768
- Input Selection:
  - PC13/OP2
  - P1.2/Timer_A2.CCRA
  - GND
  - VCC
- Capture Mode:
  - No Capture
  - Rising Edge
  - Falling Edge
  - Both Edges
- Mode:
  - Timer OFF
  - Output Compare
  - Input Capture
- Output Pins:
  - T1/T2/Timer_A2.T1A
  - P1.0/Timer_A2.T1B
  - P2.6/Timer_A2.T1A

**Output Mode:** PWM output mode 0 - OUT bit value

**Enable Capture/Compare Interrupt**

- Interrupt Handler: TimerACCRIEXHandler
- After Interrupt: Do Not Change Operating Mode

**Note:** Grace interrupt handlers are names of user-provided functions. Manual mode requires no arguments but requires a return value, e.g., unsigned short interruptHandler(void). All other modes require no arguments and return value, e.g., void interruptHandler(void).
/*
 * Electromagnetic Pendulum Driver
 * Version 1.1
 * This software pulses an electromagnetic coil to drive a clock pendulum.
 * The software precisely measures the period of each and every swing of the pendulum, and
 * uses a modified PID (proportional - integral - differential) control algorithm to adjust the swing angle
 * of a clock pendulum to speed up or slow down the pendulum and therefore the clock.
 *
 * Dick Bipes
 * dick@carveshop.com
 *
 * (c) Copyright 2014 by Dick Bipes  All rights reserved
 *
 */

AGIC DNL

#De RedLED (BIT0) // Red LED on the LaunchPad
#define GreenLED (BIT6) // Green LED on the LaunchPad
// For the clock, a red/green bi-directional LED is connected to these two port pins
#define CoilDriver (BIT3) // Base to driver transistor

Dick Bipes  carveshop.com
Synchronicity: A wood gear clock with a unique drive mechanism

#define DataLog (BIT5)  // Datalog output pin

#define CrystalFreq 32768  // timer counts per second, based on the external watch crystal
#define _600uS 20
#define _2mS 66
#define _5mS 164
#define _7mS 229
#define _10mS 328
#define _12mS 393
#define _15mS 492
#define _18mS 589
#define _20mS 655
#define _22mS 721
#define _23mS 753
#define _25mS 819
#define _30mS 983
#define _35mS 1147
#define _100mS 3277
#define NominalPulseDelay _10mS  // Delay time after voltage compare interrupt (magnet passes by coil) to coil on
#define NominalPulse _15mS  // Nominal coil pulse width
#define MinPulse _10mS  // Minimum pulse width to make sure the pendulum keeps moving
#define MaxPulse _35mS  // Maximum pulse width to limit current draw
#define QuiescePeriod _100mS  // Quiesce time after magnet pass with interrupt disabled

#define GraceTicks 50  // Number of timer ticks fast or slow that's OK (no warning LED)
#define LEDTime 600  // Experiments show that we can change the pendulum speed about 0.5% or about 150 ticks in 32768
#define ControlLimit 300  // Maximum time in seconds to allow the pulse width at a limit before lighting an LED
#define ControlError 5  // Minimum error required to flash a long-term pendulum speed LED, in seconds
#define Kp 20  // Proportional constant, empirically derived
#define Ki .1  // Integral constant, empirically derived

/*
 * ======== Variables ========
*/

int stage = 0;  // state machine stage
unsigned int timer_capture;  // current value of TAR (timer counter)
Synchronicity: A wood gear clock with a unique drive mechanism

```c
#define NominalPulse 15
#define NominalPulseDelay 16

#define TAR 4

unsigned int last_capture = 0; // previous value of TAR
unsigned int crystal_secs = 0; // elapsed seconds based upon the crystal oscillator (reference)
unsigned int pendulum_secs = 0; // elapsed seconds based upon the pendulum
int timer_overflow = 0; // timer overflow counter (generally >0, but may go to -1 in a race condition)
int error_ticks; // difference between crystal ticks and pendulum ticks (1 tick = 1/32768 second)

unsigned int pendulum_passes = 0; // pendulum pass counter, incremented when the pendulum passes the coil (twice per period)

int error = 0; // error, the difference between reference time base and the pendulum (clock)
float i_error = 0; // integral of the error
int pulse = NominalPulse; // coil pulse width
int pulse_delay = NominalPulseDelay; // delay time from detecting magnet to turning coil on
unsigned int control_count = 0; // number of consecutive seconds the pulse is set to its limit
unsigned int LEDs = 0; // port bit mask to turn on or off either LED
unsigned int enable_short_LEDs = 1; // flag to enable/disable the LEDs

/*
*  Timer A is regulated by a 32.768 kHz watch crystal. The timer is always running, and overflows exactly every two seconds.
*/

// Timer_A Overflow Interrupt Handler
void TimerAOverflowISRHandler(void)
{
    // Count seconds based on the watch crystal reference. This count is compared against the pendulum swings
    // to yield an error signal, which allows the pendulum period to be accurately controlled.
    crystal_secs += 2; // count the number of seconds that have passed based upon the crystal

    // Count of timer overflows. We expect the timer to overflow during normal operation.
    // If the pendulum is stopped for any length of time, the timer will overflow repeatedly.
    // A large overflow count in fact tells us that the pendulum has stopped, and we need to reset our software
    // and restart.
    if (timer_overflow < 1000) // increment the number of times the timer overflows, up to an arbitrary maximum
        timer_overflow++;
}

/*
* Sequencer for running the coil.
* The first state or stage is a delay, from the time induced current is detected in the coil, to when we turn
* on current to the coil.
*/
```
Synchronicity: A wood gear clock with a unique drive mechanism

* The second stage is coil on time.
* The third stage is a delay to let the coil quiesce and avoid triggering a second, undesired pulse.
*

// Timer_A Capture/Compare 0 Interrupt Handler
void TimerACCR0ISRHandler(void)
{
    switch(stage)
    {
    case 1: // turn on the coil
        P1OUT |= CoilDriver + LEDs; // turn on the coil driver (active high) and LEDs
        TACCR0 += pulse; // set the coil on time
        break;
    case 2: // turn off the coil
        P1OUT &= ~(CoilDriver + RedLED + GreenLED); // turn off the coil driver (active high) and LEDs
        TACCR0 += QuiescePeriod; // wait for the coil to settle down
        break;
    case 3: // enable the next cycle
        TACCTL0 &= ~CCIE; // disable timer interrupts
        CACTL1 &= ~CAIFG; // clear any spurious compare interrupt that may have occurred
        CACTL1 |= CAIE; // enable comparator interrupts for the next swing

        // The control loop calculation uses floating point math and can take some time to process.
        // The calculation is placed here, just after the coil is turned off, where we have sufficient time to do it
        // as nothing critical is happening.
        pulse = NominalPulse - (Kp*error + Ki*i_error); // standard PID algorithm, but with no differential term
        if (pulse > MaxPulse) // limit the max pulse to conserve battery
            { pulse = MaxPulse;
              control_count++; // this counter can tell us if we are out of control
            }
        else if (pulse < MinPulse) // limit the minimum pulse, to ensure that the pendulum does not stop
            { pulse = MinPulse;
              control_count++; }
        else
            control_count = 0;
        if (pulse < _15mS)
Synchronicity: A wood gear clock with a unique drive mechanism

```c
{  
pulse_delay = _10mS;  // for short pulses, wait until the magnet is away from the coil a bit
}
else if (pulse > _23mS)
{  
pulse_delay = _2mS;  // for long pulses, trigger the coil right away
}
else
{  
pulse_delay = _25mS - pulse;  // for intermediate pulses, delay an intermediate amount
}
break;
}
stage++;  // advance to the next state

/*
*  PWM output for logging purposes.
*/
// Timer_A Capture/Compare 1 Interrupt Handler
void TimerACCR1ISRHandler(void)
{
  // This routine sends a PWM output in proportion to the pulse width or error for data logging
  static int sw = 0;

  if (sw++ & 0x01)
  {  
P1OUT |= DataLog;  // set the port pin high
     TACCR1 += 50 + (MaxPulse - pulse);
     TACCR1 += 100 + (5*error);
  }
  else
  {  
P1OUT &= ~DataLog;  // set the port pin low
     TACCR1 += 100 - (5*error);
  }

  /*
  *  Pendulum magnet has induced a current in the coil and triggered the comparator
  */
```
Synchronicity: A wood gear clock with a unique drive mechanism

/*
   // Comparator_A+ Interrupt Handler
   void ComparatorAISRHandler(void)
   {
      timer_capture = TAR;       // capture the timer value
      CACTL1 &= ~CAIE;           // disable comparator interrupts to prevent a second trigger
      stage = 1;                 // set the sequencer to initial state
      TACCR0 = timer_capture + pulse_delay; // set the delay time from detecting the magnet to coil on
      TACCTL0 |= CCIE;           // enable timer interrupts

      if (pendulum_passes & 0x01) // every other pendulum pass (e.g. a full swing)
      {
         pendulum_secs ++;       // keep track of the elapsed time based on the pendulum
         if (pendulum_secs > LEDTime) // disable the LEDs after a period of time
            enable_short_LEDs = 0;

         // Check for more than one timer overflow, which means the pendulum had stopped. In that case, restart.
         if (timer_capture < last_capture) // OK if the timer has rolled over once
            timer_overflow--; // expected, so decrement the overflow count
         // Note that rarely the timer may overflow while in this interrupt service routine, and timer_overflow goes
         // negative
         if (timer_overflow > 0 ) // any other overflow means the pendulum was stopped
         {
            // Reset after the pendulum was stopped and restarted
            timer_overflow = 0;
            pendulum_passes = 0;
            crystal_secs = 0;
            pendulum_secs = 0;
            error = 0;
            i_error = 0;
            enable_short_LEDs = 1;
            //P1OUT ^= RedLED; // debug - toggle red LED
         }
      }

      if (pendulum_secs & 0x01) // every other full pendulum swing (same frequency as timer overflow)
      {
         // Warn of either short-term or long-term pendulum speed problems. Short term has priority.
         LEDs = 0; // assume neither LED should be lit
         // If within the short-term window after start-up, blink an LED if the pendulum is too fast or too slow
         if (enable_short_LEDs) // only turn on LEDs for the first few minutes
            LEDs = 0;
      }
   }
*/
Synchronicity: A wood gear clock with a unique drive mechanism

```c
{  error_ticks = CrystalFreq - (timer_capture - last_capture);  // compute the short-term error
  if (error_ticks > GraceTicks)  // if the pendulum is significantly faster than the reference,
    LEDs |= GreenLED;  // enable the green LED
  if (error_ticks < -GraceTicks)  // if the pendulum is significantly slower than the reference,
    LEDs |= RedLED;  // enable the red LED
}
// If the pulse width has been at its limit for an extended period of time,
// and were more than a few seconds fast or slow, we're probably out of control
else if (control_count > ControlLimit)  // if it hasn't been too long at the limit, don't turn the LED on
{
  if (error > ControlError)  // if we're more than a few seconds off, a positive error means
    LEDs |= RedLED;  // too few pendulum ticks -
  if (error < -ControlError)  // a negative error means too many pendulum ticks -
    LEDs |= GreenLED;  // too fast
};
// Capture values for the control loop
error = crystal_secs - pendulum_secs;  // error term
i_error += error;  // integral of the error
}
last_capture = timer_capture;  // remember the timer count
}
pendulum_passes++;  // count swings of the pendulum past the coil (we act on every other one only)

/*@  
 * Wait about a second and optionally light an LED or activate the coil
 */
void WaitABit(unsigned int LED)
{
  unsigned int i;
  for (i=0; i<50000; i++)  // wait a while
    P1OUT |= LED;  // turn the LED on while waiting
  P1OUT &= ~LED;  // turn the LED off
}
Synchronicity: A wood gear clock with a unique drive mechanism

/*
 * ======== main ========
 * Set up the microcontroller using Grace-generated configuration.
 * Blink the LED and pulse the coil.
 * Then go to sleep and wait for interrupts.
 */

int main(int argc, char *argv[])
{
    CSL_init(); // Activate Grace-generated configuration

    P1OUT = 0; // make sure the port pins, particularly the coil driver, is off
               // (not sure why Grace is not doing this)

    TACCTL0 &= ~CCIE; // disable timer interrupts while we play with the LEDs
    WaitABit(RedLED); // turn the red LED on for a while
    WaitABit(0);     // no LED
    WaitABit(GreenLED); // likewise with the green LED
    WaitABit(0);     // no LED
    WaitABit(CoilDriver); // activate the coil for a second so that the clock builder can determine if the
                          //  magnet polarity is correct

    CACTL1 |= CAIE;   // enable comparator interrupts for the pendulum swing

    // Enter Low Power Mode with global interrupt enabled
    __bis_SR_register(LPM0_bits + GIE);

    return (0);
}