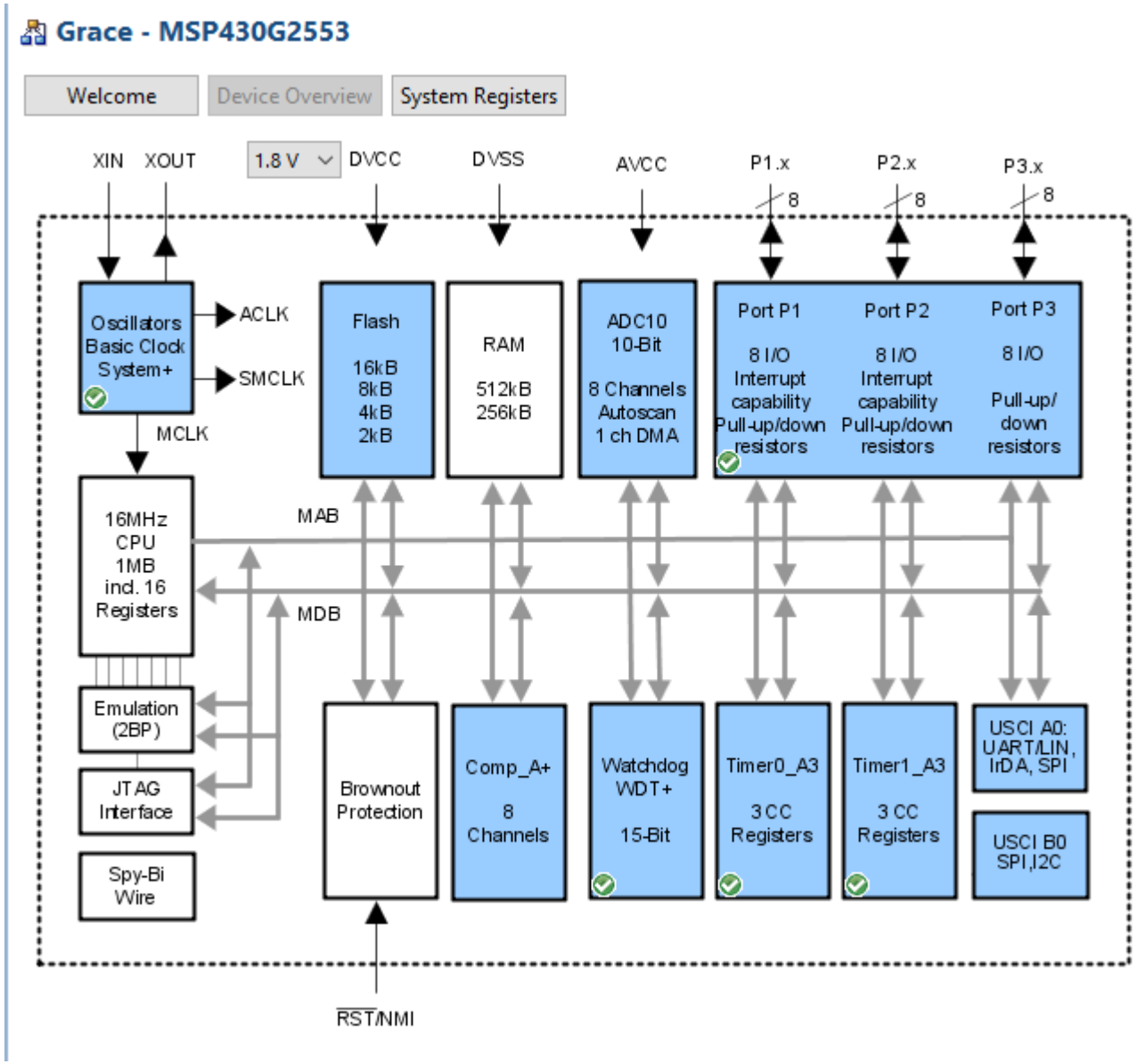

# Tempus: a Metric Clock

Dick Bipes

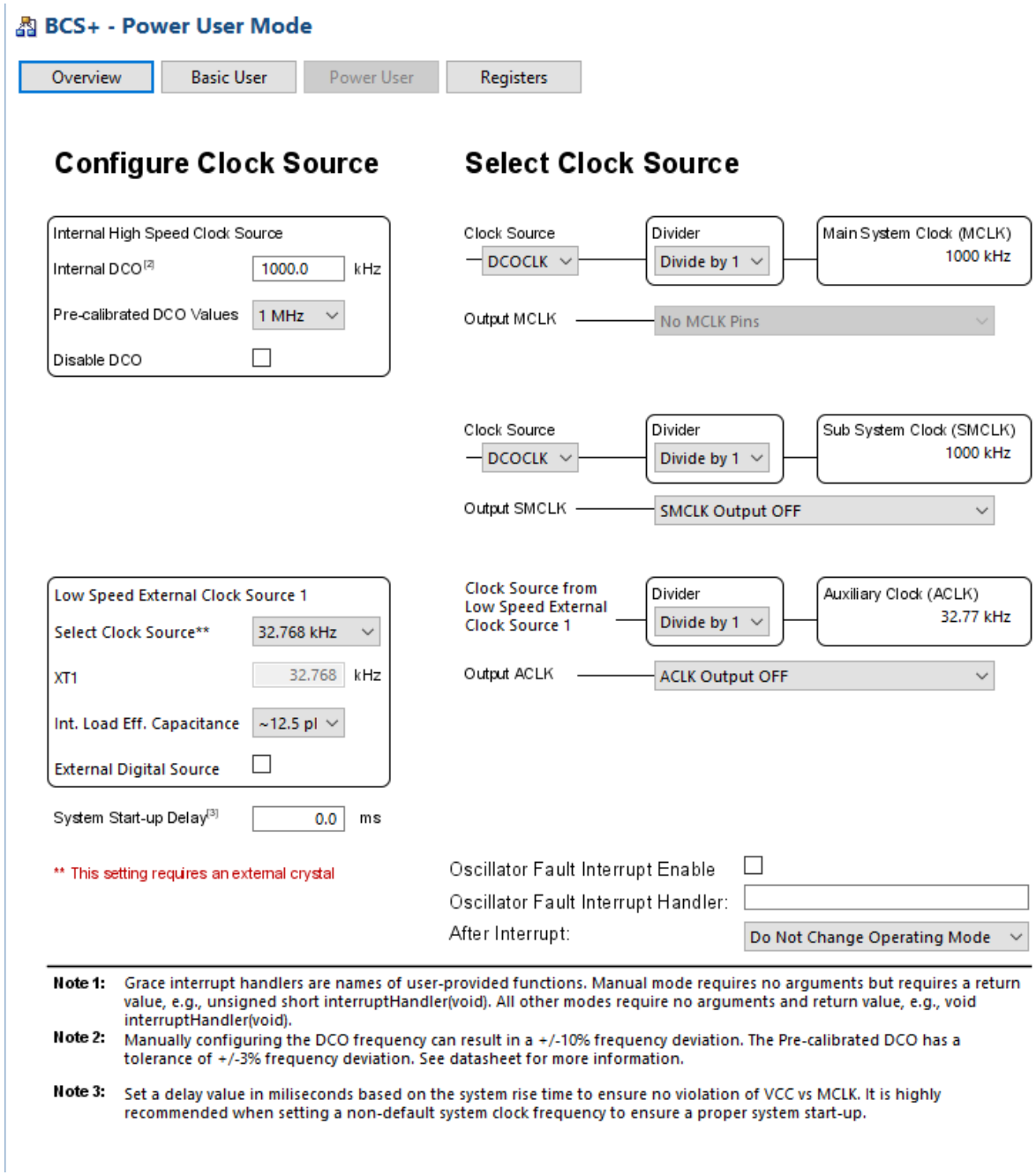
**dick@carveshop.com**

## Grace

Grace is TI's graphical code environment for the MSP430.  It can be used to graphically configure peripherals in the microcontroller.  I use Grace  for my projects.  I have included screen shots of the Grace screens that I sued to configure the microcontroller for this project.




Blocks marked with a green check mark were configured via Grace and used in this project.



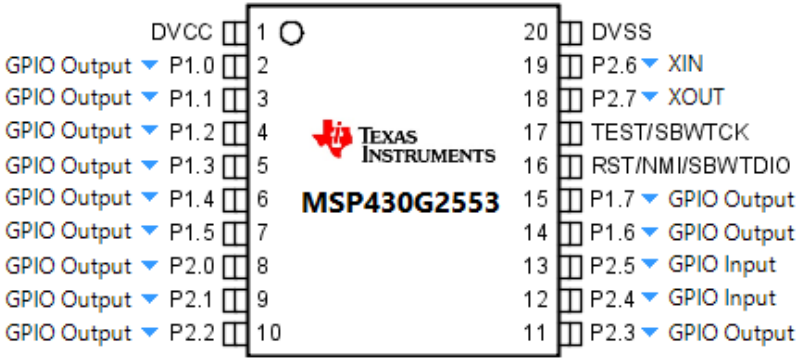

**Note 1:** Grace interrupt handlers are names of user-provided functions. Manual mode requires no arguments but requires a return value, e.g., unsigned short interruptHandler(void). All other modes require no arguments and return value, e.g., void interruptHandler(void).

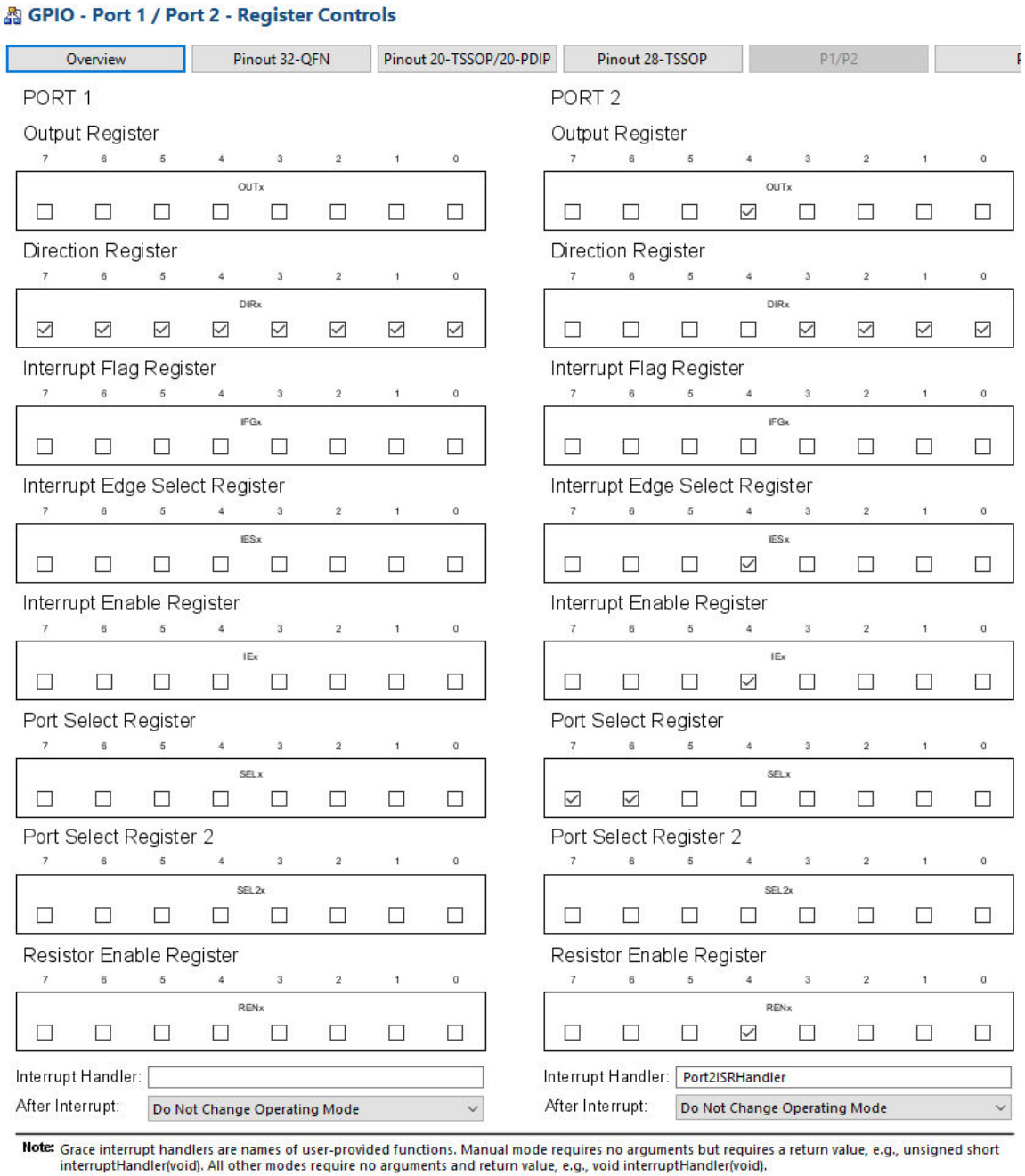
**Note 2:** Manually configuring the DCO frequency can result in a +/-10% frequency deviation. The Pre-calibrated DCO has a tolerance of +/-3% frequency deviation. See datasheet for more information.

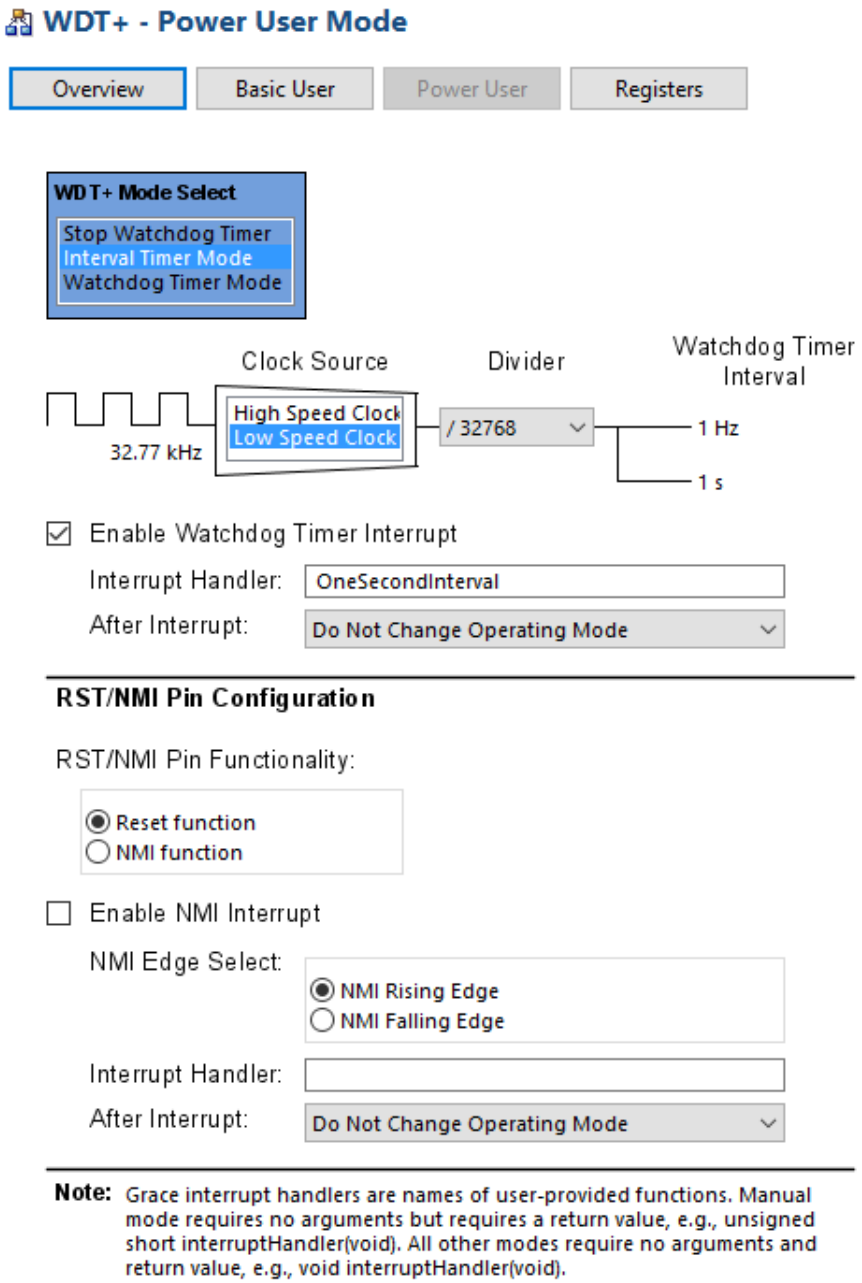
**Note 3:** Set a delay value in miliseconds based on the system rise time to ensure no violation of VCC vs MCLK. It is highly recommended when setting a non-default system clock frequency to ensure a proper system start-up.

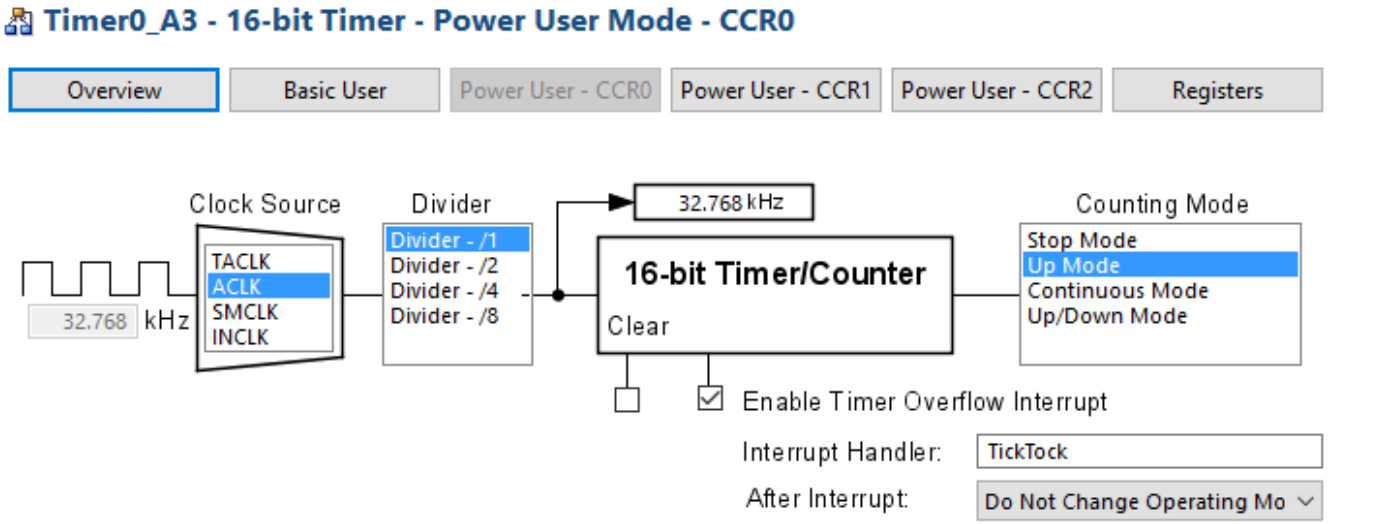
## GPIO - Pinout 20-TSSOP/20-PDIP

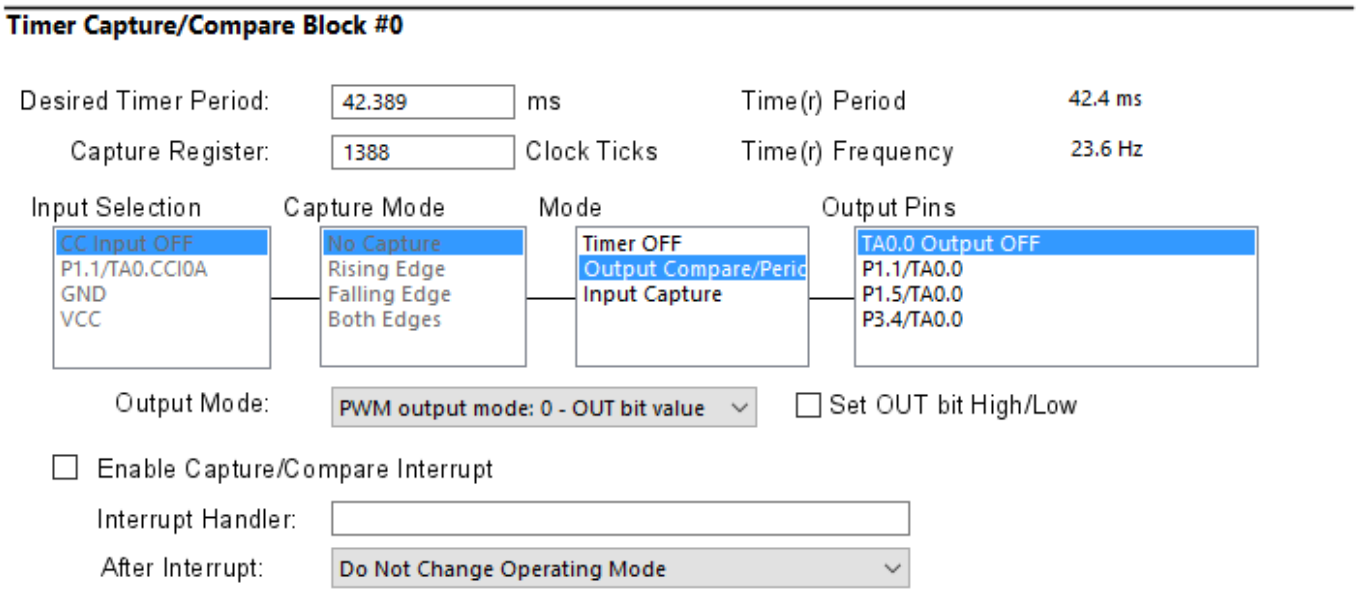
Overview | Pinout 32-QFN | Pinout 20-TSSOP/20-PDIP | Pinout 28-TSSOP | P1/P2 | P3



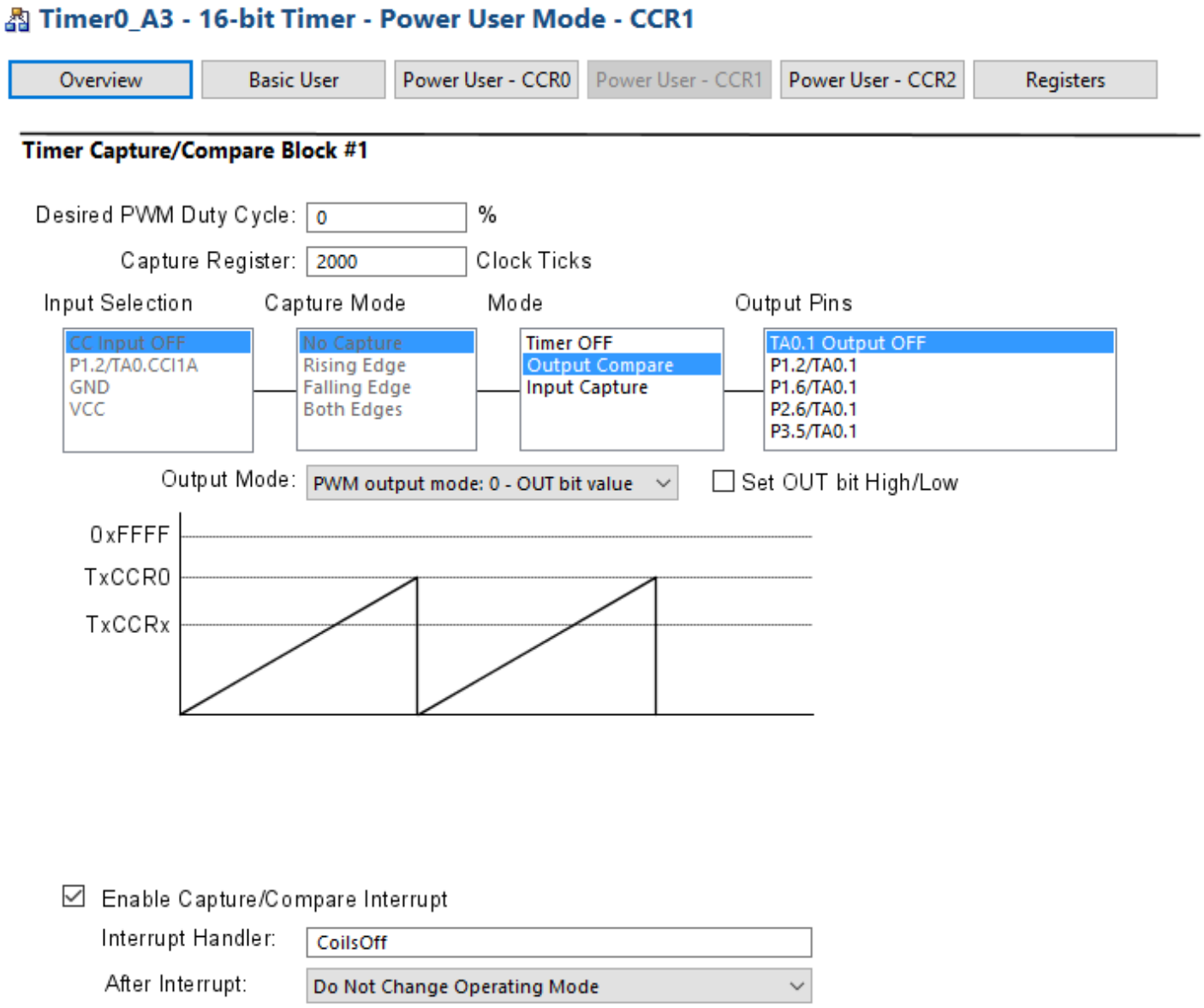

GPIO - Port 1 / Port 2 - Register Controls
Overview
Pinout 32-QFN
Pinout 20-TSSOP/20-PDIP
Pinout 28-TSSOP
P1/P2
PORT 1
Output Register
OUTx
Direction Register
DIRx
Interrupt Flag Register
IFGx
Interrupt Edge Select Register
IESx
Interrupt Enable Register
IEx
Port Select Register
SELx
Port Select Register 2
SEL2x
Resistor Enable Register
RENx
Interrupt Handler:
After Interrupt:
Do Not Change Operating Mode
PORT 2
Output Register
OUTx
Direction Register
DIRx
Interrupt Flag Register
IFGx
Interrupt Edge Select Register
IESx
Interrupt Enable Register
IEx
Port Select Register
SELx
Port Select Register 2
SEL2x
Resistor Enable Register
RENx
Interrupt Handler:
Port2ISRHandler
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).

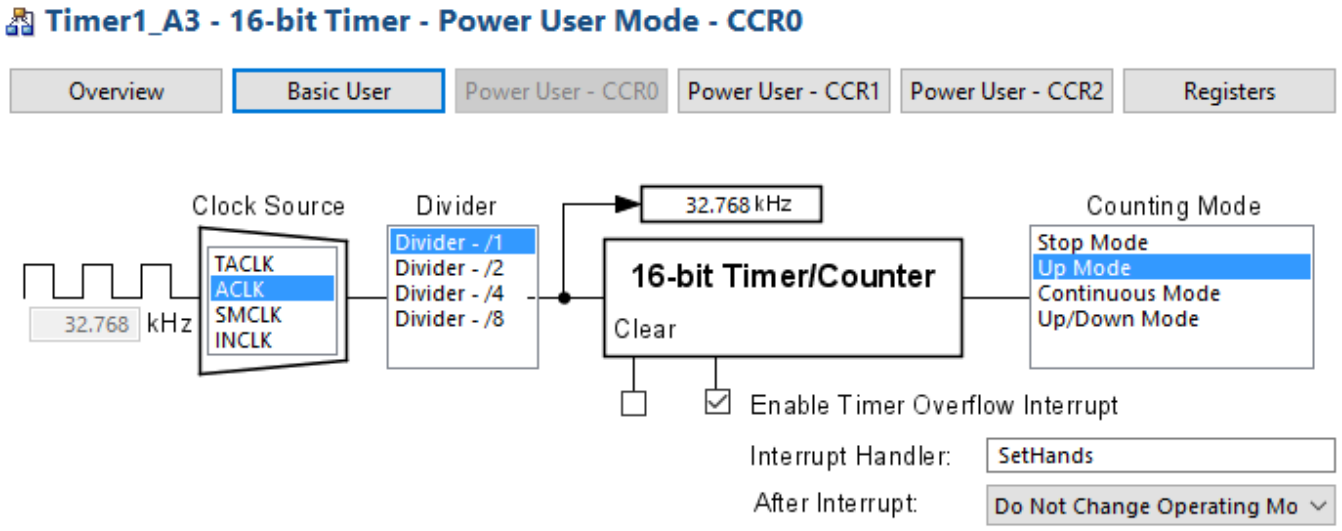

WDT+ - Power User Mode
Overview
Basic User
Power User
Registers
WDT+ Mode Select
Stop Watchdog Timer
Interval Timer Mode
Watchdog Timer Mode
Clock Source
Divider
Watchdog Timer Interval
High Speed Clock
Low Speed Clock
32.77 kHz
/ 32768
1 Hz
1 s
Enable Watchdog Timer Interrupt
Interrupt Handler:
OneSecondInterval
After Interrupt:
Do Not Change Operating Mode
RST/NMI Pin Configuration
RST/NMI Pin Functionality:
Reset function
NMI function
Enable NMI Interrupt
NMI Edge Select:
NMI Rising Edge
NMI Falling Edge
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(void).

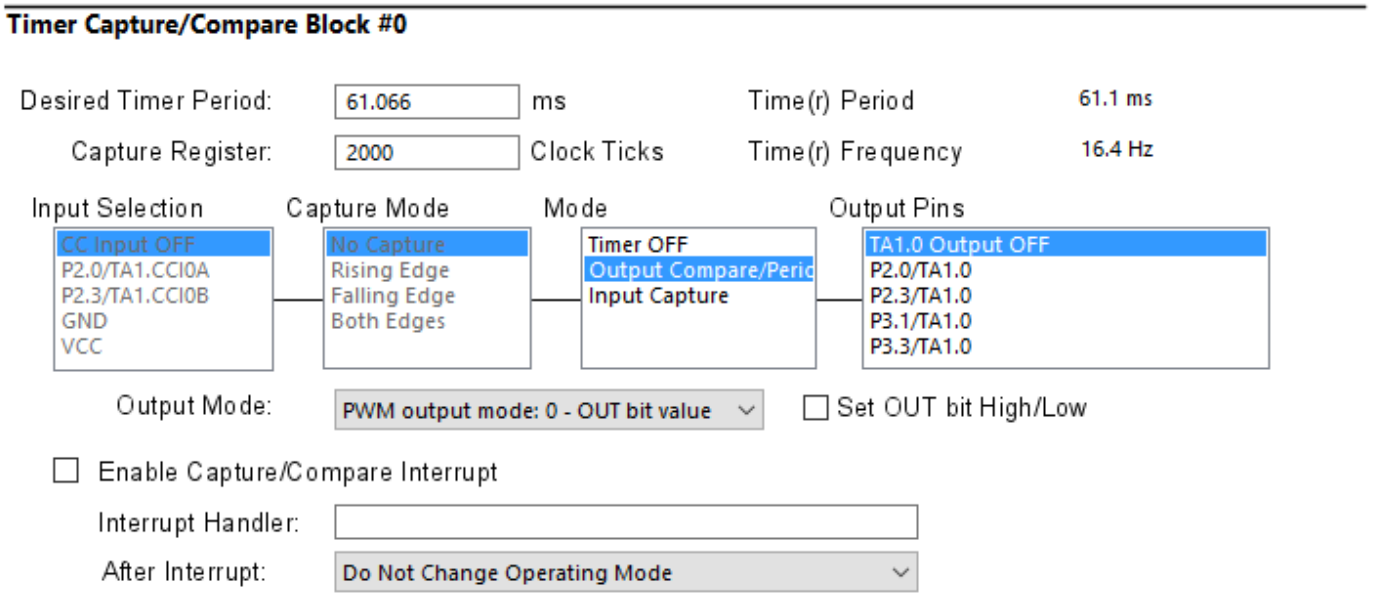


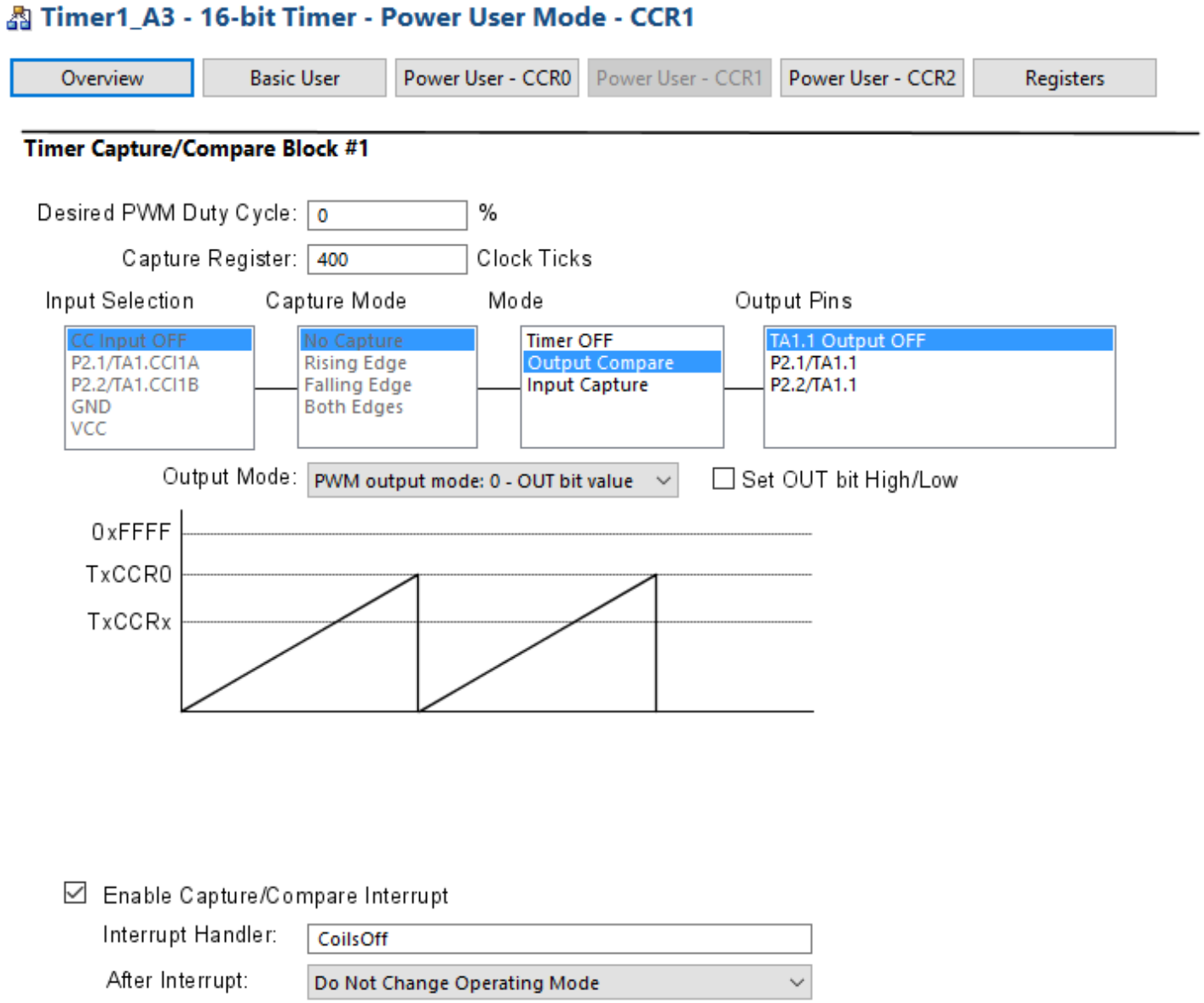

**Note:** Grace interrupt handlers are names of user-provided functions. Manual mode requires no arguments but requires a return value, e.g., unsigned short interruptHandler(void). All other modes require no arguments and return value, e.g., void interruptHandler(void).


Timer0_A3 - 16-bit Timer - Power User Mode - CCR1
Overview
Basic User
Power User - CCR0
Power User - CCR1
Power User - CCR2
Registers
Timer Capture/Compare Block #1
Desired PWM Duty Cycle: 0 %
Capture Register: 2000 Clock Ticks
Input Selection
Capture Mode
Mode
Output Pins
CC Input OFF
P1.2/TA0.CCI1A
GND
VCC
No Capture
Rising Edge
Falling Edge
Both Edges
Timer OFF
Output Compare
Input Capture
TA0.1 Output OFF
P1.2/TA0.1
P1.6/TA0.1
P2.6/TA0.1
P3.5/TA0.1
Output Mode: PWM output mode: 0 - OUT bit value
Set OUT bit High/Low
0xFFFF
TxCCR0
TxCCRx
Enable Capture/Compare Interrupt
Interrupt Handler: CoilsOff
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).




**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).


Timer1_A3 - 16-bit Timer - Power User Mode - CCR1
Overview
Basic User
Power User - CCR0
Power User - CCR1
Power User - CCR2
Registers
Timer Capture/Compare Block #1
Desired PWM Duty Cycle: 0 %
Capture Register: 400 Clock Ticks
Input Selection
CC Input OFF
P2.1/TA1.CCI1A
P2.2/TA1.CCI1B
GND
VCC
Capture Mode
No Capture
Rising Edge
Falling Edge
Both Edges
Mode
Timer OFF
Output Compare
Input Capture
Output Pins
TA1.1 Output OFF
P2.1/TA1.1
P2.2/TA1.1
Output Mode: PWM output mode: 0 - OUT bit value
Set OUT bit High/Low
0xFFFF
TxCCR0
TxCCRx
Enable Capture/Compare Interrupt
Interrupt Handler: CoilsOff
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).

# Source Code

```
/*
 *  Metric Clock
 *
 *  This software operates stepper motors to implement a metric clock.
 *  Metric time has 10 hours per day, 100 minutes per hour, and
 *  100 seconds per minute/\.
 *
 *  Dick Bipes
 *  dick@carveshop.com
 *
 *
 *  (c) Copyright 2015 by Dick Bipes  All rights reserved
 *
  */


/*
 * ======== Standard MSP430 includes ========
 */
#include <msp430.h>

/*
 * ======== Grace related includes ========
 */
#include <ti/mcu/msp430/csl/CSL.h>

/*
 * GRACE port setup notes
 * P2REN = 1 to enable the pullup/pulldown resistor
 * P2OUT = 1 to select pullup
 */


enum SetStateType               // States of the clock - either setting the hands, or running
{
        SetHours,               // Set the hours (by placing the hand directly over the numeral -
                                // i.e. if the time is 8:50, place the hand directly pointing to 8
        AlignMinutes,           // Align the minutes hand to the index position
        AlignSeconds,           // Align the seconds hand to the index position (10 on our metric clock)
        SetMinutes,             // Set the minutes hand.  The hour hand will be advanced accordingly.
                                // i.e. if the minutes is set to 50, the hour hand will advance halfway from 8 to 9
        Run                     // Start running the clock
};
enum SetStateType set_state = SetHours;  // After power up, the first state is to align the hours hand

// Pushbutton definitions
#define PushButtonPortIN P2IN           // Pushbutton port
#define PushButton BIT4                 // Pushbutton bit
#define PushButtonPortIFG P2IFG         // Pushbutton interrupt flag
```

```
#define PushButtonPortIES P2IES          // Pushbutton interrupt edge select

#define PushButtonStateChange 3          // number of seconds pushbutton must be released to advance to the next hand-setting state

// We are driving 28BYJ-48 5v stepper motors for our clock.  There are three motors, one each for
// hours, minutes, and seconds hands. The motors are being operated in full step mode.  Consequently,
// two coils are energized simultaneously. Due to the orientation of the motors, the seconds motor
// runs clockwise; the minutes and hours motors run counterclockwise.

#define SecondsMotorPort P2OUT                  // port to which this motor is connected
#define SecondsMotorMask 0x0f                   // mask used to change only this motor's 4 bits on the port
#define SecondsMotorPhase1 BIT0 + BIT1          // each phase energizes two coils in the motor
#define SecondsMotorPhase2 BIT1 + BIT2
#define SecondsMotorPhase3 BIT2 + BIT3
#define SecondsMotorPhase4 BIT3 + BIT0
unsigned int seconds_motor_step[4] = {SecondsMotorPhase1, SecondsMotorPhase2, SecondsMotorPhase3, SecondsMotorPhase4};
unsigned int seconds_motor_step_ccw[4] = {SecondsMotorPhase4, SecondsMotorPhase3, SecondsMotorPhase2, SecondsMotorPhase1};

unsigned int seconds_motor_phase = 0;           // this variable indexes the array containing the motor phases

#define MinutesMotorPort P1OUT
#define MinutesMotorMask 0xf0
#define MinutesMotorPhase1 BIT4 + BIT5
#define MinutesMotorPhase2 BIT5 + BIT6
#define MinutesMotorPhase3 BIT6 + BIT7
#define MinutesMotorPhase4 BIT7 + BIT4
unsigned int minutes_motor_step[4] = {MinutesMotorPhase4, MinutesMotorPhase3, MinutesMotorPhase2, MinutesMotorPhase1};
unsigned int minutes_motor_step_ccw[4] = {MinutesMotorPhase1, MinutesMotorPhase2, MinutesMotorPhase3, MinutesMotorPhase4};

unsigned int minutes_motor_phase = 0;


#define HoursMotorPort P1OUT
#define HoursMotorMask 0x0f
#define HoursMotorPhase1 BIT0 + BIT1
#define HoursMotorPhase2 BIT1 + BIT2
#define HoursMotorPhase3 BIT2 + BIT3
#define HoursMotorPhase4 BIT3 + BIT0
unsigned int hours_motor_step[4] = {HoursMotorPhase4, HoursMotorPhase3, HoursMotorPhase2, HoursMotorPhase1};
unsigned int hours_motor_step_ccw[4] = {HoursMotorPhase1, HoursMotorPhase2, HoursMotorPhase3, HoursMotorPhase4};

unsigned int hours_motor_phase = 0;

#define SlowSetSpeed     2000           // Rate at which to step the motors in slow mode when aligning hands and setting time
                                        // 2000/32,768 or about 60 mS per step
#define FastSetSpeed     450            // Rate at which to step the motors in fastest mode when aligning hands and setting time
                                        // 300/32,768 or about 10 mS per step


#define PhaseMask 0x03   // The step index counts from 0 to 3 then back to zero.

#define MinutesPerRev  100       // number of minutes in one complete revolution of the minute hand
```

```
#define HoursPerRev    10       // number of hours in one complete revolution of the hour hand

// The number of timer counts in the period is TACCR0+1, so ideal ticks per step is set to 1389.2605459057 - 1
#define IdealTicksPerStep 1388.2605459057               // Exact number of timer ticks per motor step to operate seconds hand
unsigned int actual_ticks_per_step = 1388;              // Actual integer number of timer ticks per step being executed
float error = 0;                                        // Difference between actual timer ticks needed versus integer number executed
unsigned int timer_ticks = 0;                    //

unsigned int xx = 0;
unsigned long int steps = 0;
unsigned int minute_steps = 0;
unsigned int PushbuttonReleasedSeconds = 0;             // Interval timer whose value is the number of seconds since the pushbutton was released


/*
 * ======== Interrupt handlers ========
 */

/*
 * Turn the motor coils off.
 *
 * Invoked by both a Timer A0 and Timer A1 compare interrupt to turn the coils off after about 12 mS
 * (set via Grace).
 * Also called by other interrupt service routines to ensure coils are off.
 *
 */

        void CoilsOff (void)
        {
                SecondsMotorPort &= ~SecondsMotorMask;          // turn off seconds motor coils
                MinutesMotorPort &= ~MinutesMotorMask;          // turn off minutes motor coils
                HoursMotorPort &= ~HoursMotorMask;              // turn off hours motor coils
        }



/*
 *  Timer A0 is used to time pulses to the stepper motors. Timer A0 is clocked by a 32.768 kHz watch crystal.
 *  The timer is set to "Up" mode and counts up to the value in TA0CCR0 to set the interval of our basic clock's "tick".
 *  When the timer reaches the "tick" count, it generates an interrupt.  This interrupt is used to advance the
 *  stepper motors to their next incremental step by turning on the proper coils.  The timer is reset to zero
 *  to start timing the next interval.
 *
 *  To conserve power and reduce heat in the motors, the stepper motor coils are only engaged for a short time,
 *  sufficient to advance the motor's shaft. Once it is moved, current can be turned off and the motor will hold its position.
 *  Capture/compare register TA0CCR1 is set to "compare" at a value equal to the "on" time for the coils.  The timer generates
 *  an interrupt when it reaches this count, and the coils are turned off.
 *
 */
```

```
/*
 * Advance motors to the next step while the clock is running.
 * Invoked by Timer A0 overflow, which is running at the basic clock 'tick'.
 *
 */

        void TickTock (void)
        {
                error += (IdealTicksPerStep - actual_ticks_per_step);    // compute the difference between actual and ideal ticks

                if (error >= 1.0)                                // has the error reached a positive integer value?
                {
                        ++actual_ticks_per_step;                 // yes, make up the error
                        error = error - 1;                       // take credit for doing so
                }

                if (error <= -1.0)                               // has the error reached a negative integer value?
                {
                        --actual_ticks_per_step;                 // yes, make up the error
                        error = error + 1;                       // take credit for doing so
                }

                TA0CCR0 = actual_ticks_per_step;                 // set the next timer overflow period

                CoilsOff();                     // ensure motor coils are off

                SecondsMotorPort |= (SecondsMotorMask & seconds_motor_step[PhaseMask & seconds_motor_phase++]);  // activate next pair of coils
                if (++timer_ticks >= MinutesPerRev)      //  count metric seconds - time to move minutes hand?
                        {
                                // activate next pair of coils to step motor
                                MinutesMotorPort |= (MinutesMotorMask & minutes_motor_step[PhaseMask & minutes_motor_phase++]);

                                if (++minute_steps >= HoursPerRev)       // count metric minutes - time to move hours hand?
                                {
                                        // activate next pair of coils to step motor
                                        HoursMotorPort |= (HoursMotorMask & hours_motor_step[PhaseMask & hours_motor_phase++]);
                                        minute_steps = 0;        // reset minutes
                                }
                                timer_ticks = 0;         // reset seconds

                        }
        }

/*
 * One second interval timer, used to determine how long the pushbutton was released
 * Invoked via the watchdog timer.
 *
 */

        void OneSecondInterval (void)
        {
                ++PushbuttonReleasedSeconds;     // count the number of seconds since the pushbutton was released
```

```
        }

/*
 * Advance motors to the next step while setting the clock.
 * Invoked via Timer A1 overflow interrupt.
 */

        void SetHands (void)
        {
                CoilsOff();                     // ensure motor coils are off

                switch (set_state)  // operate one of the motors depending upon the state
                {
                case AlignSeconds:
                        SecondsMotorPort |= (SecondsMotorMask & seconds_motor_step[PhaseMask & seconds_motor_phase++]);
                        break;
                case AlignMinutes:
                        MinutesMotorPort |= (MinutesMotorMask & minutes_motor_step[PhaseMask & minutes_motor_phase++]);
                        break;
                case SetHours:
                        HoursMotorPort |= (HoursMotorMask & hours_motor_step[PhaseMask & hours_motor_phase++]);
                        minute_steps = 0;
                        break;
                case SetMinutes:
                        MinutesMotorPort |= (MinutesMotorMask & minutes_motor_step[PhaseMask & minutes_motor_phase++]);
                        if (++minute_steps >= HoursPerRev)              // count metric seconds
                        {
                                HoursMotorPort |= (HoursMotorMask & hours_motor_step[PhaseMask & hours_motor_phase++]);
                                minute_steps = 0;
                        }

                        break;
                };

                if (TA1CCR0 > FastSetSpeed)                     // Is the timer running at less than max speed?
                        TA1CCR0 = TA1CCR0 - TA1CCR0/40;         // yes, speed it up a bit

                PushbuttonReleasedSeconds = 0;                  // reset the "pushbutton released" interval

        }

        void Port2ISRHandler (void)
        {
                PushButtonPortIFG &= ~PushButton;               // Clear the pushbutton interrupt flag


                if (PushButtonPortIN & PushButton)              // was the pushbutton pressed or released?
                {
                    // pushbutton was released - normal operation
                        TA1CTL &= ~(MC1 + MC0);                 // Clear MCx bits to stop timer A1
                        CoilsOff();                     // ensure motor coils are off (since we may not have reached timer A1 compare count yet)
                    WDTCTL = WDTPW + WDTTMSEL + WDTSSEL; // Start the watchdog timer in interval mode from ACLOCK.
```

```
                                                // The timer will interrupt every one second.
            }
            else
            {
               // pushbutton was pressed - start moving one of the hands fairly fast to set the clock

                    if (PushbuttonReleasedSeconds >= PushButtonStateChange)  // if the pushbutton had been released for several seconds...
                            ++set_state;                                     //   go to the next state

                    if (set_state != Run)                           // have we executed all states?
                    {
                            TA1CCR0 = SlowSetSpeed;                 // No.  Start slowly...
                            TA1CTL |= MC_1;                         // Start timer A1 in up mode to move the hands fairly rapidly
                    }
                    else
                       TA0CTL |= MC_1;                              // Yes, start timer A0 in up mode to start the clock running
            }

            PushButtonPortIES ^= PushButton;        // Toggle the interrupt edge to generate an interrupt on the opposite edge
                                                    // that generated this one - in other words, generate an interrupt both
                                                    // when the pushbutton is pressed and when it is released
        }


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

    // >>>>> Fill-in user code here <<<<<

    // The timers are set up via Grace and default to being set to run - turn them back off
    WDTCTL = WDTPW + WDTHOLD;    // Stop the watchdog timer
    TA0CTL &= ~(MC1 + MC0);      // Clear MCx bits to stop timer
    TA1CTL &= ~(MC1 + MC0);      // Clear MCx bits to stop timer

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


    return (0);
}
```