```
/****************************************************************************
 Module
   TopHSMTemplate.c

 Revision
   2.0.1

 Description
   This is a template for the top level Hierarchical state machine

 Notes

 History
 When           Who     What/Why
 -------------- ---   --------
 02/20/17 14:30 jec     updated to remove sample of consuming an event. We
                        always want to return ES_NO_EVENT at the top level
                        unless there is a non-recoverable error at the
                        framework level
 02/03/16 15:27 jec     updated comments to reflect small changes made in '14 & '15
                        converted unsigned char to bool where appropriate
                        spelling changes on true (was True) to match standard
                        removed local var used for debugger visibility in 'C32
                        removed Microwave specific code and replaced with generic
 02/08/12 01:39 jec     converted from MW_MasterMachine.c
```

```
 02/06/12 22:02 jec      converted to Gen 2 Events and Services Framework
 02/13/10 11:54 jec      converted During functions to return Event_t
                         so that they match the template
 02/21/07 17:04 jec      converted to pass Event_t to Start...()
 02/20/07 21:37 jec      converted to use enumerated type for events
 02/21/05 15:03 jec      Began Coding
****************************************************************************/
/*---------------------------- Include Files ----------------------------*/
/* include header files for this state machine as well as any machines at the
   next lower level in the hierarchy that are sub-machines to this machine
*/
#include "ES_Configure.h"
#include "ES_Framework.h"

#include "dbprintf.h"

#include <xc.h>
#include <sys/attribs.h>

#include "TopHSM_Leader.h"
#include "GameplaySM.h"

/*---------------------------- Module Defines ----------------------------*/
#define US_P_TICK 4*(5E-8)*(1E6) // Assumes prescalar of 4
#define FREQ_IN_US 1100
```

```
#define THIRD_GAME_TIME 46000 //218000/3

#define GAMEPLAY_LED LATBbits.LATB3


/*---------------------------- Module Functions ----------------------------*/

static ES_Event_t DuringGameplayState( ES_Event_t Event);

static ES_Event_t DuringIdleState( ES_Event_t Event);


/*---------------------------- Module Variables ----------------------------*/

// everybody needs a state variable, though if the top level state machine

// is just a single state container for orthogonal regions, you could get

// away without it

static MasterState_t CurrentState;

// with the introduction of Gen2, we need a module level Priority var as well

static uint8_t MyPriority;


volatile uint16_t CapturedTime;

volatile static uint16_t RolloverCounter;

volatile static TimeTicks_t CurrentVal;

volatile static uint32_t PrevVal;

volatile static uint32_t DeltaTicks;


volatile static uint16_t PreviousBeacon;


/*------------------------------ Module Code ------------------------------*/

/****************************************************************************

 Function
```

```
    InitMasterSM

 Parameters

    uint8_t : the priorty of this service

 Returns

    boolean, False if error in initialization, True otherwise

 Description

    Saves away the priority,  and starts

    the top level state machine

 Notes

 Author

    J. Edward Carryer, 02/06/12, 22:06

*****************************************************************************/

bool InitMasterSM_Leader ( uint8_t Priority )

{

  ES_Event_t ThisEvent;

  MyPriority = Priority;  // save our priority

  clrScrn();

  ThisEvent.EventType = ES_ENTRY;

  // Start the Master State machine
```

```
  StartMasterSM_Leader( ThisEvent );

  return true;
}

/****************************************************************************
 Function
    PostMasterSM

 Parameters
    ES_Event_t ThisEvent , the event to post to the queue

 Returns
    boolean False if the post operation failed, True otherwise

 Description
    Posts an event to this state machine's queue
 Notes

 Author
    J. Edward Carryer, 10/23/11, 19:25
****************************************************************************/
bool PostMasterSM_Leader( ES_Event_t ThisEvent )
{
  return ES_PostToService( MyPriority, ThisEvent);
```

```
}

/****************************************************************************
 Function
    RunMasterSM

 Parameters
   ES_Event: the event to process

 Returns
   ES_Event: an event to return

 Description
   the run function for the top level state machine
 Notes
   uses nested switch/case to implement the machine.
 Author
   J. Edward Carryer, 02/06/12, 22:09
****************************************************************************/
ES_Event_t RunMasterSM_Leader( ES_Event_t CurrentEvent )
{
   bool MakeTransition = false;/* are we making a state transition? */
   MasterState_t NextState = CurrentState;
   ES_Event_t EntryEventKind = { ES_ENTRY, 0 };// default to normal entry to new state
   ES_Event_t ReturnEvent = { ES_NO_EVENT, 0 }; // assume no error
```

```
static uint8_t GameplayTimeoutCounter = 0;


        switch ( CurrentState )
{
        case GameplayState :    // If current state is state one
        // Execute During function for state one. ES_ENTRY & ES_EXIT are
        // processed here allow the lowere level state machines to re-map
        // or consume the event
       CurrentEvent = DuringGameplayState(CurrentEvent);
        //process any events
        if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
        {
        switch (CurrentEvent.EventType)
        {
                case ES_TIMEOUT : //If event is event one
                if(GAMEPLAY_TIMER == CurrentEvent.EventParam){
                DB_printf("REEnable Gameplay Timer (need to implement)\n");
                GameplayTimeoutCounter++;
                if(3 > GameplayTimeoutCounter){
                        ES_Timer_InitTimer(GAMEPLAY_TIMER,THIRD_GAME_TIME);
                }else{
                        NextState = IdleState;//Decide what the next state will be
                        MakeTransition = true; //mark that we are taking a transition
                }
                }
                // Execute action function for state one : event one
```

```
//                  NextState = IdleState;//Decide what the next state will be
                    // for internal transitions, skip changing MakeTransition
//                  MakeTransition = true; //mark that we are taking a transition
                    // if transitioning to a state with history change kind of entry
                    //EntryEventKind.EventType = ES_ENTRY_HISTORY;
                    break;
                    // repeat cases as required for relevant events
            }
            }
            break;
            case IdleState :            // If current state is state one
            // Execute During function for state one. ES_ENTRY & ES_EXIT are
            // processed here allow the lowere level state machines to re-map
            // or consume the event
           CurrentEvent = DuringIdleState(CurrentEvent);
            //process any events
            if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
            {
            switch (CurrentEvent.EventType)
            {
                    case ES_GAME_SWITCH : //If event is event one
                    // Execute action function for state one : event one
                    NextState = GameplayState;//Decide what the next state will be
                    // for internal transitions, skip changing MakeTransition
                    MakeTransition = true; //mark that we are taking a transition
                    // if transitioning to a state with history change kind of entry
```

```
                    //EntryEventKind.EventType = ES_ENTRY_HISTORY;
                    break;
                    // repeat cases as required for relevant events
            }
            }
            break;
            // repeat state pattern as required for other states
            }
            //   If we are making a state transition
            if (MakeTransition == true)
            {
            //   Execute exit function for current state
        CurrentEvent.EventType = ES_EXIT;
            RunMasterSM_Leader(CurrentEvent);

        CurrentState = NextState; //Modify state variable

            // Execute entry function for new state
            // this defaults to ES_ENTRY
            RunMasterSM_Leader(EntryEventKind);
            }
    // in the absence of an error the top level state machine should
    // always return ES_NO_EVENT, which we initialized at the top of func
    return(ReturnEvent);
}
/*****************************************************************************
```

```
 Function
    StartMasterSM

 Parameters
    ES_Event CurrentEvent

 Returns
    nothing

 Description
    Does any required initialization for this state machine
 Notes

 Author
    J. Edward Carryer, 02/06/12, 22:15
******************************************************************************/
void StartMasterSM_Leader ( ES_Event_t CurrentEvent )
{
  // if there is more than 1 state to the top level machine you will need
  // to initialize the state variable
  CurrentState = IdleState; //change to be IdleState!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  // now we need to let the Run function init the lower level state machines
  // use LocalEvent to keep the compiler from complaining about unused var
  RunMasterSM_Leader(CurrentEvent);
  return;
}
```

```
/*****************************************************************************
 Function
    QueryTopHSMTemplateSM

 Parameters
    None

 Returns
    MasterState_t  The current state of the Top Level Template state machine

 Description
    returns the current state of the Template state machine
 Notes

 Author
    J. Edward Carryer, 2/05/22, 10:30AM
*****************************************************************************/
MasterState_t  QueryTopHSM_Leader ( void )
{
  return(CurrentState);
}

/*****************************************************************************
 private functions
 *****************************************************************************/
```

```
static ES_Event_t DuringGameplayState( ES_Event_t Event)
{
   ES_Event_t ReturnEvent = Event; // assme no re-mapping or comsumption

        // process ES_ENTRY, ES_ENTRY_HISTORY & ES_EXIT events
        if ( (Event.EventType == ES_ENTRY) ||
        (Event.EventType == ES_ENTRY_HISTORY) )
        {
        // implement any entry actions required for this state machine

        /*******The following giant block of initialization will need to be moved to the
other************************
        During function when we go for the project*****************************************************************/
        // Set Up SPI
        // Create digital output pins for CS, SCK, and MOSI and map them
        // CS on RA0
      TRISAbits.TRISA0 = 0;
      ANSELAbits.ANSA0 = 0;
      RPA0R = 0b0011;
        // SCK on RB14
      TRISBbits.TRISB14 = 0;
      ANSELBbits.ANSB14 = 0;
        // MOSI on RA1
      TRISAbits.TRISA1 = 0;
      ANSELAbits.ANSA1 = 0;
```

```
        RPA1R = 0b0011;
//      // Create a digital input for MISO on RB5 and map it
//      TRISBbits.TRISB5 = 1;
//      SDI1R = 0b0001;
        // Disable SPI Module
        SPI1CONbits.ON = 0;
        // Clear the Receive Buffer
        SPI1BUF;
        // Enable the Enhanced Buffer
        SPI1CONbits.ENHBUF = 1;
        // Set the baudrate (since we're communicating with a PIC, choose the quickest)
        SPI1BRG = 0;
        // Clear the SPIROV bit
        SPI1STATbits.SPIROV = 0;
        // Write the desired settings to SPIxCON
        SPI1CONbits.MSTEN = 1; // Select Leader mode
        SPI1CONbits.MSSEN = 1; // Drive CS automatically
        SPI1CONbits.CKE = 0; // Read on 2nd edge
        SPI1CONbits.CKP = 1; // SCK idles high
        SPI1CONbits.FRMPOL = 0; // CS is active low
        SPI1CON2bits.AUDEN = 0;
        SPI1CONbits.MODE32 = 0; // Enable 8 bit transfers
        SPI1CONbits.MODE16 = 0;
        // Enable SPI
        SPI1CONbits.ON = 1;
```

```
CurrentVal.FullTime = 0;
PrevVal = 0;
RolloverCounter = 0;
DeltaTicks = 0;
PreviousBeacon = 0;

    // Configure Input Capture for capturing IR

    // Configure RA2 as a digital input and map to IC1
TRISAbits.TRISA2 = 1;
IC1R = 0b0000;
    // Configure Timer 2 for IC
    // Disable Timer 2
T2CONbits.ON = 0;
    // Select the internal clock as the source
T2CONbits.TCS = 0;
    // Choose a 1:4 prescaler (1:4 is just the standard for ME218)
T2CONbits.TCKPS = 0b010;
    // Set the initial value for the timer to be 0
TMR2 = 0;
    // Since it's an IC counter, set the PR to the max
PR2 = 0xFFFF;
    // Clear the Timer 2 interrupt flag
IFS0CLR = _IFS0_T2IF_MASK;
    // Set the priority of the Timer 2 interrupt to 6
IPC2bits.T2IP = 6;
```

```
    // Enable Interrupts from Timer 2
IEC0SET = _IEC0_T2IE_MASK;
    // Enable Timer 2
T2CONbits.ON = 1;


    // Configure IC
    // Disable IC1
IC1CONbits.ON = 0;
    // Specify a 16 bit timer
IC1CONbits.C32 = 0;
    // Select Timer 2
IC1CONbits.ICTMR = 1;
    // Configure to interrupt on every capture event
IC1CONbits.ICI = 0;
    // Configure to interrupt on every rising edge
IC1CONbits.ICM = 0b011;
    do{
   IC1BUF;
    }while(IC1CONbits.ICBNE != 0);
    // Clear the interrupt flag for IC1
IFS0CLR = _IFS0_IC1IF_MASK;
    // Set the priority of IC1 to 7
IPC1bits.IC1IP = 7;
    // Enable interrupts from IC1
    DB_printf("Enable Beacon ISR\n");
IEC0SET = _IEC0_IC1IE_MASK;
```

```
        // Enable IC1
    IC1CONbits.ON = 1;


        __builtin_enable_interrupts();

        /************************************* End of During Block to be
moved******************************************/


        // Start the Gameplay Timer BB!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        DB_printf("Enable Gameplay Timer (need to implement)\n");

        ES_Timer_InitTimer(GAMEPLAY_TIMER,THIRD_GAME_TIME);

        // Turn on Gameplay LEDs BB!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

        DB_printf("Enable Gameplay LED\n");

    TRISBbits.TRISB3 = 0;

    ANSELBbits.ANSB3 = 0;

    GAMEPLAY_LED = 1;


        // after that start any lower level machines that run in this state

        //StartLowerLevelSM( Event );

        StartGameplaySM( Event );


        // repeat the StartxxxSM() functions for concurrent state machines

        // on the lower level

        }

        else if ( Event.EventType == ES_EXIT )

        {

        // on exit, give the lower levels a chance to clean up first
```

```
    //RunLowerLevelSM(Event);
    RunGameplaySM(Event);
    DB_printf("Disable Gameplay LED\n");
GAMEPLAY_LED = 0;
    DB_printf("Send Motor PIC ES_SHUTDOWN (need to implement))\n");
    do{
   SPI1BUF;
    }while(SPI1STATbits.SPIRBF != 0);
SPI1BUF = STOP_DRIVE;

    // repeat for any concurrently running state machines
    // now do any local exit functionality
    // Disable Gameplay LEDs
    // Send Motor PIC command to shut down

    }else
    // do the 'during' function for this state
    {
    // run any lower level state machine
    // ReturnEvent = RunLowerLevelSM(Event);
ReturnEvent = RunGameplaySM(Event);

    // repeat for any concurrent lower level machines

    // do any activity that is repeated as long as we are in this state
    }
```

```
        // return either Event, if you don't want to allow the lower level machine
        // to remap the current event, or ReturnEvent if you do want to allow it.
        return(ReturnEvent);
}

static ES_Event_t DuringIdleState( ES_Event_t Event)
{
   ES_Event_t ReturnEvent = Event; // assme no re-mapping or comsumption

        // process ES_ENTRY, ES_ENTRY_HISTORY & ES_EXIT events
        if ( (Event.EventType == ES_ENTRY) ||
        (Event.EventType == ES_ENTRY_HISTORY) )
        {
        // implement any entry actions required for this state machine
        DB_printf("In Idle State\n");
      GAMEPLAY_LED = 0;
        // after that start any lower level machines that run in this state
        //StartLowerLevelSM( Event );
        // repeat the StartxxxSM() functions for concurrent state machines
        // on the lower level
        }
        else if ( Event.EventType == ES_EXIT )
        {
        // on exit, give the lower levels a chance to clean up first
        //RunLowerLevelSM(Event);
        // repeat for any concurrently running state machines
```

```
            // now do any local exit functionality
            DB_printf("Exit Idle State\n");

            }else
            // do the 'during' function for this state
            {
            // run any lower level state machine
            // ReturnEvent = RunLowerLevelSM(Event);

            // repeat for any concurrent lower level machines

            // do any activity that is repeated as long as we are in this state
            }
            // return either Event, if you don't want to allow the lower level machine
            // to remap the current event, or ReturnEvent if you do want to allow it.
            return(ReturnEvent);
}

void __ISR(_INPUT_CAPTURE_1_VECTOR,IPL7SOFT) IC1_ISR(void){
            static uint16_t CurrentBeacon;
            // read the IC1 buffer into a variable
    CapturedTime = IC1BUF;
            // Clear the interrupt flag for IC1
    IFS0CLR = _IFS0_IC1IF_MASK;
            // If a rollover has occurred and the Timer 2 interrupt flag is still set
            if((0x8000 > CapturedTime) && (1 == IFS0bits.T2IF)){
```

```
        // Increment the rollover counter
    RolloverCounter++;
        // Clear the Timer 2 Interrupt flag
    IFS0CLR = _IFS0_T2IF_MASK;


        }
        // Set the lower 16 bits of CurrentVal equal to the captured value
CurrentVal.ByBytes[0] = CapturedTime;
        // Set the upper 16 bits of CurrentVal equal to the rollover counter
CurrentVal.ByBytes[1] = RolloverCounter;
        // Compute the period (in ticks) between pulses
DeltaTicks = CurrentVal.FullTime - PrevVal;
        // Set the previous time equal to the current time
PrevVal = CurrentVal.FullTime;
        if((300 - 5 <= US_P_TICK*DeltaTicks) && (300 + 5 >= US_P_TICK*DeltaTicks)){
    CurrentBeacon = E;
        }else if((700 - 5 <= US_P_TICK*DeltaTicks) && (700 + 5 >= US_P_TICK*DeltaTicks)){
    CurrentBeacon = D;
        }else if((500 - 5 <= US_P_TICK*DeltaTicks) && (500 + 5 >= US_P_TICK*DeltaTicks)){
    CurrentBeacon = K;
        }else if((1100 - 25 <= US_P_TICK*DeltaTicks) && (1100 + 25 >= US_P_TICK*DeltaTicks)){
    CurrentBeacon = A;
        }else{
    CurrentBeacon = 0;
        }
        if((PreviousBeacon != CurrentBeacon) && (CurrentBeacon != 0)){
```

```
        ES_Event_t CMD_Event;
        CMD_Event.EventType = ES_BEACON;
        CMD_Event.EventParam = CurrentBeacon;
            PostMasterSM_Leader(CMD_Event);
            }
    PreviousBeacon = CurrentBeacon;
//    if((FREQ_IN_US - 20 <= US_P_TICK*DeltaTicks) && (FREQ_IN_US + 20 >=
US_P_TICK*DeltaTicks)){
//    if(((300 - 20 <= US_P_TICK*DeltaTicks) && (300 + 20 >= US_P_TICK*DeltaTicks))
//        || ((700 - 20 <= US_P_TICK*DeltaTicks) && (700 + 20 >= US_P_TICK*DeltaTicks))
//        || ((500 - 20 <= US_P_TICK*DeltaTicks) && (500 + 20 >= US_P_TICK*DeltaTicks))
//        || ((1100 - 20 <= US_P_TICK*DeltaTicks) && (1100 + 20 >= US_P_TICK*DeltaTicks))){
//        // Disable Interrupts from IC1
//        IEC0CLR = _IEC0_IC1IE_MASK;
//        // Post an event to the run function to send the SPI command
//        ES_Event_t CMD_Event;
//        CMD_Event.EventType = ES_SPI;
//        PostTopHSM_Leader(CMD_Event);
//    }
}


void __ISR(_TIMER_2_VECTOR,IPL6SOFT) Timer2_ISR(void){
            // Disable interrupts globally
            __builtin_disable_interrupts();
            // If the Timer 2 interrupt flag is set
            if(1 == IFS0bits.T2IF){
```

```
        // Increment the rollover counter
    RolloverCounter++;
        // Clear the Timer 2 interrupt flag
    IFS0CLR = _IFS0_T2IF_MASK;
        }
        // Enable interrupts globally
        __builtin_enable_interrupts();
}
```