```
/*******************************************************************************
 Module
   StrategyFSM.c

 Revision
   1.0.1

 Description
   This is a template file for implementing flat state machines under the
   Gen2 Events and Services Framework.

 Notes

 History
 When           Who     What/Why
 -------------- ---     --------
*******************************************************************************/
/*---------------------------- 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 <stdio.h>
#include <stdlib.h>
#include <mc9s12e128.h>
#include <S12e128bits.h>
#include <Bin_Const.h>
#include <termio.h>
#include <hidef.h>
#include "S12eVec.h"

#include "E128_PWM.h"         //has all prescale definitions
#include "E128_SPI.h"
#include "E128_Servo.h"
#include "FAC_FSM.h"
#include "NavigationFSM.h"
#include "AlignPPService.h"
#include "DriveTrainService.h"
#include "ArtilleryFSM.h"
#include "StrategyFSM.h"
#include "ResupplyService.h"

/*---------------------------- Module Defines ----------------------------*/
#define TARGETSHIP 0
#define TARGETBOT 1
#define RELOAD 2
#define TARGETPOWERPLANT 3
#define TARGETDEPOT 4
#define MOVE2LOC  5
#define BLINDSHOT 6


#define MINUTE 60000 //1 ms = 1 tick


/*---------------------------- Module Functions ----------------------------*/
/* prototypes for private functions for this machine.They should be functions
   relevant to the behavior of this state machine
*/
static void PowerDown( void );
/*---------------------------- Module Variables ----------------------------*/
```

```
// everybody needs a state variable, you may need others as well.
// type of state variable should match that of enum in header file
static StrategyState_t CurrentState;

static Coordinates Locations[5];


//Instructions
static unsigned char InstructNum = 0;
static unsigned char Instructions[11] =
{
   MOVE2LOC, TARGETPOWERPLANT, TARGETSHIP, TARGETSHIP, TARGETSHIP, MOVE2LOC,
RELOAD, MOVE2LOC, RELOAD, TARGETSHIP, TARGETSHIP

};

static unsigned char TotalNumInstructions = 11;

//Location of Resupply Depots
signed int DepotX[2] = { 80, 170 };
signed int DepotY[2] = { 128, 129 };

//Game Info Trackers
static unsigned char SelfColor;
static unsigned char EnemyBot;
static boolean Flag_1minute = False;
static boolean Fire_Flag = True;
static char BallCount = 5; //assume we have 5 balls in the beginning

// bounds on travel, so bot doesn't hit walls when turning
static signed int ybound = 25;//clearance to avoid rotating into wall (BotDiagonal/2)
static signed int xbound = 60; //have to be 48 ticks (1 ship length 10" + Shade cover 3" + BotDiagonal/2
8.1")
static signed int xmidbound = 25; //clearance for center wall divide (BotDiagonal/2)
static signed int half_field = 127;
static signed int xlim = 255;
static signed int ylim = 255;
//shoot distances
//static unsigned char ShootDistX_far = 160; //line up 160 ticks (5 feet) away from target to shoot it
static signed int ShootDistX_close = 125; //102 line up 95 ticks (3 feet) away from target to shoot it

//xdesired, ydesired
static signed int Desiredx;
static signed int Desiredy;

// with the introduction of Gen2, we need a module level Priority var as well
static uint8_t MyPriority;


/*------------------------------ Module Code ------------------------------*/
/****************************************************************************
 Function
    InitStrategyFSM

 Parameters
    uint8_t : the priorty of this service

 Returns
    boolean, False if error in initialization, True otherwise
```

```
 Description
    Saves away the priority, sets up the initial transition and does any
    other required initialization for this state machine
 Notes

 Author
    Jina Wang 3/8/2013
****************************************************************************/
boolean InitStrategyFSM ( uint8_t Priority )
{
   ES_Event ThisEvent;


   Locations[0].x = 60;
   Locations[0].y = 47;
   Locations[1].x = 85;
   Locations[1].y = 150;
   Locations[2].x = 60;
   Locations[2].y = 100;
   Locations[3].x = 60;
   Locations[3].y = 127;
   Locations[4].x = 60;
   Locations[4].y = 60;
   //navigate in a box


   MyPriority = Priority;
   // put us into the Initial PseudoState
   CurrentState = Waiting4GameStart;
   // post the initial transition event
   ThisEvent.EventType = ES_INIT;
   if (ES_PostToService( MyPriority, ThisEvent) == True)
      {
         return True;
      }
   else
      {
         return False;
      }
}

/****************************************************************************
 Function
    PostStrategyFSM

 Parameters
    EF_Event ThisEvent , the event to post to the queue

 Returns
    boolean False if the Enqueue operation failed, True otherwise

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

 Author
       Jina Wang 3/8/2013
****************************************************************************/
boolean PostStrategyFSM( ES_Event ThisEvent )
{
```

```
    return ES_PostToService( MyPriority, ThisEvent);
}

/****************************************************************************
 Function
    RunStrategyFSM

 Parameters
   ES_Event : the event to process

 Returns
   ES_Event, ES_NO_EVENT if no error ES_ERROR otherwise

 Description
   add your description here
 Notes
   uses nested switch/case to implement the machine.
 Author
     Jina Wang 3/8/2013
****************************************************************************/
ES_Event RunStrategyFSM( ES_Event ThisEvent )
{
    ES_Event ReturnEvent, NavEvent, MtrEvent, ShootEvent, StrategyEvent;
    static int k = 0;
    static unsigned char TargetNum;

    ReturnEvent.EventType = ES_NO_EVENT; // assume no errors

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

                        BEGIN STATE MACHINE CODE

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

    switch ( CurrentState )
       {
       case Waiting4GameStart :
          if ( ThisEvent.EventType == GAME_START )
             {
                //Turn on Game Active Light and initialize SideLight Port
                DDRU |= SIDELED_DIR;

                DDRU |= ACTIVELED_DIR;
                ACTIVELED_PORT = HI;

                //Initialize 2 min GameTimer
                ES_Timer_InitTimer(GAME_TIMER, MINUTE); //initialize 1 min timer (internal flag to check
for 2 mins)


                CurrentState = GatheringIntel;

             }

          if (ThisEvent.EventType == ES_NEW_KEY)
             {
                CurrentState = GatheringIntel;
             }

          break; //end Waiting4GameStart State
```

```
    case GatheringIntel :       // If current state is state one
      if (ThisEvent.EventType == ES_NEW_KEY)
        {
          // Post to Strategy in order to execute first step in list
          StrategyEvent.EventType = EVAL_INSTRUCTION;
          StrategyEvent.EventParam = 0;
          PostStrategyFSM(StrategyEvent);

          CurrentState = Evaluating_Strategy;
        }

      if ( ThisEvent.EventType == FAC_UPDATED )
        {
          //determine which side we are on
          GameStartIntel();
          SelfColor = QueryColor();

          //Light red/blue side
          if (SelfColor == RED)
            {
              SIDELED_PORT = HI;
            }
          else
            {
              SIDELED_PORT = LO;
            }

          //determine enemy bot #
          EnemyBot = DetermineEnemy();
          InitIRDetectHardware(); //initialize Timer capabilities IR Detection of power plants

          //Set GatherIntelComplete to TRUE to begin execution of Instructions
          //printf("Done gathering intel - Our Color = %i, EnemyBot = %i, \r\n", SelfColor, EnemyBot);

          // Post to Strategy in order to execute first step in list
          StrategyEvent.EventType = EVAL_INSTRUCTION;
          StrategyEvent.EventParam = 0;
          PostStrategyFSM(StrategyEvent);


          CurrentState = Evaluating_Strategy;
        }

      break; //end GatheringIntel State

    case Evaluating_Strategy :
      if (ThisEvent.EventType == EVAL_INSTRUCTION)
        {
          switch (Instructions[InstructNum])
            {
            case ( BLINDSHOT ):
              Desiredx = QueryX(SelfNum);
              Desiredy = QueryY(SelfNum);

              //start flywheel cannon
              //ShootEvent.EventType = FIREUP;
              //ShootEvent.EventParam = SHOOT_SHORT; //may want to specify speed of shot -
may need a helper function to determine which lenght to shoot
              //PostArtilleryFSM(ShootEvent);
```

```
                    //start flywheel cannon
                    ShootEvent.EventType = FIREUP;
                    ShootEvent.EventParam = SHOOT_SHORT; //may want to specify speed of shot -
may need a helper function to determine which lenght to shoot
                    PostArtilleryFSM(ShootEvent);

                    //post navigation event
                    NavEvent.EventType = NEW_DESTINATION;
                    NavEvent.EventParam = 0;
                    PostNavigationFSM(NavEvent);
                    CurrentState = Moving2ShootingPosition;

                    break;

                case ( MOVE2LOC ) :

                    //navigate to T-section of field
                    if (SelfColor == RED)
                      {
                        Desiredx = DepotX[SelfColor];
                      }
                    else if (SelfColor == BLUE)
                      {
                        Desiredx = DepotX[SelfColor];
                      }

                    Desiredy = half_field;

                    Fire_Flag = False;

                    //post navigation event
                    NavEvent.EventType = NEW_DESTINATION;
                    NavEvent.EventParam = 0;
                    PostNavigationFSM(NavEvent);
                    CurrentState = Moving2ShootingPosition;
                    break;

                case ( TARGETSHIP ):
                    /******************************** TESTING *****************
                    Desiredx = Locations[k].x;
                    Desiredy = Locations[k].y;
                    k++;

                    if(k > 4)
                    {
                    k = 0;
                    }
                    ***********************************************************/

                    TargetNum = DetermineShip2Target(SelfColor);


                    //printf("Target is %i, \r\n", TargetNum);

                    //RED FIELD x = [0,127]
                    if ( SelfColor == RED )
                      {
                        Desiredx = QueryX(TargetNum) - ShootDistX_close;
```

```
        //check if too close to x = 0
        if ( Desiredx < xbound )
           {
              Desiredx = xbound;
           }
        //check if too close to midfield
        else if (Desiredx > (half_field - xmidbound))
           {
              Desiredx = (half_field - xmidbound);
           }


        Desiredy = QueryY(TargetNum);

        //check if too close to y = 0
        if (Desiredy < ybound)
           {
              Desiredy = ybound;
           }
        else if (Desiredy > (ylim - ybound))
           {
              Desiredy = (ylim - ybound);
           }

     }

  //BLUE FIELD x = [128,255]
  else
     {
        Desiredx = QueryX(TargetNum) + ShootDistX_close;

        //check if too close to x = 255
        if (Desiredx > (xlim - xbound))
           {
              Desiredx = (xlim - xbound);
           }
        //check if too close to midfield
        else if (Desiredx < (half_field + xmidbound))
           {
              Desiredx = (half_field + xmidbound);
           }

        Desiredy = QueryY(TargetNum);

        //check if too close to y = 0
        if (Desiredy < ybound)
           {
              Desiredy = ybound;
           }
        else if (Desiredy > (ylim - ybound))
           {
              Desiredy = (ylim - ybound);
           }

     }

  if (TargetNum == 0)
     {
        //navigate to T-section of field
        if (SelfColor == RED)
```

```
                {
                    Desiredx = DepotX[SelfColor];
                }
            else if (SelfColor == BLUE)
                {
                    Desiredx = DepotX[SelfColor];
                }

            Desiredy = half_field;
        }

        //start flywheel cannon
        ShootEvent.EventType = FIREUP;
        ShootEvent.EventParam = SHOOT_SHORT; //may want to specify speed of shot - may need a helper function to determine which lenght to shoot
        PostArtilleryFSM(ShootEvent);

        //post navigation event
        NavEvent.EventType = NEW_DESTINATION;
        NavEvent.EventParam = 0;
        PostNavigationFSM(NavEvent);

        CurrentState = Moving2ShootingPosition;
        break;

    case ( TARGETBOT ):
        //post navigation event
        NavEvent.EventType = NEW_DESTINATION;
        NavEvent.EventParam = 0;
        PostNavigationFSM(NavEvent);

        //start flywheel cannon
        ShootEvent.EventType = FIREUP;
        ShootEvent.EventParam = SHOOT_SHORT; //may want to specify speed of shot
        PostArtilleryFSM(ShootEvent);

        CurrentState = Moving2ShootingPosition;
        break;

    case ( RELOAD ) :
        Desiredx = DepotX[SelfColor];
        Desiredy = DepotY[SelfColor];

        //post navigation event
        NavEvent.EventType = NEW_DESTINATION;
        NavEvent.EventParam = 0;
        PostNavigationFSM(NavEvent);

        CurrentState = Reloading;
        break;

    case ( TARGETPOWERPLANT ) :

        //turn on detection hardware
        EnableIRDetection(True);
        FlywheelOff();
        //puts("turned on IR detection ability \r\n");

        //initiate timer to restrict time allowed for alignment detection
        ES_Timer_InitTimer( ALIGN_TIMER, 7000 );
```

```
                //post to motor to rotate CW to align with enemy's powerplant
                MtrEvent.EventType = ALIGNPP;
                MtrEvent.EventParam = 0;
                PostDriveTrainService(MtrEvent);

                CurrentState = Moving2ShootingPosition;
                break;
            }
        }

    if (ThisEvent.EventType == GAME_OVER)
        {
            PowerDown();

            CurrentState = GameOver;
        }

    if ( (ThisEvent.EventType == ES_TIMEOUT) && (ThisEvent.EventParam == GAME_TIMER) )
        {
            if (Flag_1minute == False)
                {
                    Flag_1minute = True;// set that 1 minute has passed
                    ES_Timer_InitTimer(GAME_TIMER, MINUTE);
                }
            else
                {
                    //post to self that game is over
                    StrategyEvent.EventType = GAME_OVER;
                    StrategyEvent.EventParam = 1;
                    PostStrategyFSM(StrategyEvent);

                    PowerDown();

                    CurrentState = GameOver;
                }
        }
    break; //end Evaluating_Strategy


case Moving2ShootingPosition :
    switch ( ThisEvent.EventType )
        {
        case ( DESTINATION_REACHED ) :

            if (Fire_Flag == True)
                {
                    //start flywheel cannon  *********
                    if (Instructions[InstructNum] == TARGETPOWERPLANT)
                        {
                            ShootEvent.EventType = FIREUP;
                            ShootEvent.EventParam = SHOOT_PP;
                            PostArtilleryFSM(ShootEvent);
                        }
                    else
                        {
                            ShootEvent.EventType = FIREUP;
                            ShootEvent.EventParam = SHOOT_SHORT;
                            PostArtilleryFSM(ShootEvent);
                        }
```

```
                    //printf("CurrentPosition is %i, %i, %i \r\n", QueryX(SelfNum), QueryY(SelfNum),
QueryTheta(SelfNum));
                    ShootEvent.EventType = FIRE;
                    ShootEvent.EventParam = 1;
                    PostArtilleryFSM(ShootEvent);


                    CurrentState = ShootingTarget;
                  }
                else
                  {
                    InstructNum++; //instruction complete

                    if (InstructNum > TotalNumInstructions )
                      {
                        InstructNum = 0;
                        //check gamestate and possibly end game
                      }

                    // Post to Strategy in order to execute first step in list
                    StrategyEvent.EventType = EVAL_INSTRUCTION;
                    StrategyEvent.EventParam = 0;
                    PostStrategyFSM(StrategyEvent);

                    //reset Fire_Flag to True
                    Fire_Flag = True;

                    CurrentState = Evaluating_Strategy;
                  }
                break;

              case ( GAME_OVER ) :
                PowerDown();

                CurrentState = GameOver;
                break;

              case ( ES_TIMEOUT ) :
                if (ThisEvent.EventParam == GAME_TIMER)
                  {
                    if (Flag_1minute == False)
                      {
                        Flag_1minute = True;// set that 1 minute has passed
                        ES_Timer_InitTimer(GAME_TIMER, MINUTE);
                      }
                    else
                      {
                        //post to self that game is over
                        StrategyEvent.EventType = GAME_OVER;
                        StrategyEvent.EventParam = 0;
                        PostStrategyFSM(StrategyEvent);

                        PowerDown();

                        CurrentState = GameOver;
                      }
                  }

                //time to allow detection of enemy's power plant timed out
```

```
                //go back to evaluating instruction and pretend that we shot something
                if (ThisEvent.EventParam == ALIGN_TIMER)
                   {
                      EnableIRDetection( False ); //turn off IR detection ability

                      //puts("time to detect IR has timed out and stopped motor. Move on to next instruction
\r\n");

                      ShootEvent.EventType = NO_SHOT;
                      ShootEvent.EventParam = 0;
                      PostArtilleryFSM(ShootEvent);

                      MtrEvent.EventType = STOP_MOTOR;
                      MtrEvent.EventParam = 0;
                      PostDriveTrainService(MtrEvent);

                      InstructNum++; //instruction complete
                      if ( InstructNum > TotalNumInstructions )
                         {
                            InstructNum = 0;
                            //check gamestate andd possibly end game
                         }

                      printf("finished instruction %i \r\n", InstructNum );

                      // Post to Strategy in order to execute next step in list
                      StrategyEvent.EventType = EVAL_INSTRUCTION;
                      StrategyEvent.EventParam = 0;
                      PostStrategyFSM(StrategyEvent);

                      CurrentState = Evaluating_Strategy; //go back to Evaluating_Strategy to determine
next Action
                   }

                break;
             }
          break; //end Moving2Position State

       case ShootingTarget :
          switch ( ThisEvent.EventType )
             {
             case ( BALL_DEPLOYED ) :

                InstructNum++; //instruction complete
                if ( InstructNum > TotalNumInstructions )
                   {
                      InstructNum = 0;
                      //check gamestate andd possibly end game
                   }

                printf("finished instruction %i \r\n", InstructNum );

                // Post to Strategy in order to execute next step in list
                StrategyEvent.EventType = EVAL_INSTRUCTION;
                StrategyEvent.EventParam = 0;
                PostStrategyFSM(StrategyEvent);

                CurrentState = Evaluating_Strategy; //go back to Evaluating_Strategy to determine next
Action
                break;
```

```
            case ( GAME_OVER ) :

                PowerDown();

                CurrentState = GameOver;
                break;

            case ( ES_TIMEOUT ) :
                if (ThisEvent.EventParam == GAME_TIMER)
                    {
                        if (Flag_1minute == False)
                            {
                                Flag_1minute = True;// set that 1 minute has passed
                                ES_Timer_InitTimer(GAME_TIMER, MINUTE);
                            }
                        else
                            {
                                //post to self that game is over
                                StrategyEvent.EventType = GAME_OVER;
                                StrategyEvent.EventParam = 0;
                                PostStrategyFSM(StrategyEvent);

                                PowerDown();

                                CurrentState = GameOver;
                            }
                    }
                break;
            }

        break; //end ShootingTarget

    case Reloading :
        switch (ThisEvent.EventType)
            {
            case ( DESTINATION_REACHED ):

                if (SelfColor == RED)
                    {
                        SetThetaManually(191);
                    }
                else
                    {
                        SetThetaManually(64);
                    }

                //printf("CurrentPosition is %i, %i, %i \r\n", QueryX(SelfNum), QueryY(SelfNum), QueryTheta
(SelfNum));
                MtrEvent.EventType = DRIVE;
                MtrEvent.EventParam = REVERSE;
                PostDriveTrainService(MtrEvent);

                break;

                // Put here provision to stop updating of controller
            case ( RESUPPLY_COVER_REACHED ) :

                //puts("Resupply Cover Reached");
                //Disable Interrupt
```

```
    TurnOffPControl();   //Disable P-Control Interrupt

    break;

case ( READY2RELOAD ) :

    //puts("Ready2Reload Event Captured");
    //InitTimer to stay at reload station for 16.5 seconds
    ES_Timer_InitTimer(RELOAD_TIMER, 7000); //change to 16.5 for real scenario

    //Start the resupply light
    EmitResupplySignal(True);
    break;

case ( ES_TIMEOUT ) : //ES_TIMEOUT is equivalent to reload complete
    if (ThisEvent.EventParam == RELOAD_TIMER)
      {
        //Turn off resupply light
        EmitResupplySignal(False);
        //puts("ir signal should be turned off \r\n");
        //move back into view of FAC because will be in "shade"
        ES_Timer_InitTimer(MOVE_TIMER, 2500);

        //puts("Got ES_Timeout for Reload Timer, setting PWM Duty Cyles");
        //manually control duty cycle to circumvent PI control
        SetDutyCycle(PWM_CHANNEL0, 80);
        SetDutyCycle(PWM_CHANNEL1, 80);

        CurrentState = LeavingShade;
      }


    if (ThisEvent.EventParam == GAME_TIMER)
      {
        if (Flag_1minute == False)
          {
            Flag_1minute = True;// set that 1 minute has passed
            ES_Timer_InitTimer(GAME_TIMER, MINUTE);
          }
        else
          {
            //post to self that game is over
            StrategyEvent.EventType = GAME_OVER;
            StrategyEvent.EventParam = 0;
            PostStrategyFSM(StrategyEvent);

            PowerDown();

            CurrentState = GameOver;
          }

      }


    break;

case ( GAME_OVER ) :

    PowerDown();
    puts("game is over. motors/lights turned off \r\n");
```

```
            CurrentState = GameOver;
            break;

        }
    break; //end Reloading State

case LeavingShade :
    if ( (ThisEvent.EventType == ES_TIMEOUT ) && (ThisEvent.EventParam == MOVE_TIMER) )
        {
            MtrEvent.EventType = STOP_MOTOR;
            MtrEvent.EventParam = 0;
            PostDriveTrainService(MtrEvent);

            //puts("Move Timer Timeout");

            InstructNum++; //instruction complete

            if (InstructNum > TotalNumInstructions )
                {
                    InstructNum = 0;
                    //check gamestate and possibly end game
                }

            // Post to Strategy in order to execute first step in list
            StrategyEvent.EventType = EVAL_INSTRUCTION;
            StrategyEvent.EventParam = 0;
            PostStrategyFSM(StrategyEvent);

            CurrentState = Evaluating_Strategy;
        }

    if ( ThisEvent.EventType == GAME_OVER )
        {
            PowerDown();
            puts("game is over. motors/lights turned off \r\n");

            CurrentState = GameOver;
        }

    if ( (ThisEvent.EventType == ES_TIMEOUT ) && (ThisEvent.EventParam == GAME_TIMER) )

        {
            if (Flag_1minute == False)
                {
                    Flag_1minute = True;// set that 1 minute has passed
                    ES_Timer_InitTimer(GAME_TIMER, MINUTE);
                }
            else
                {
                    //post to self that game is over
                    StrategyEvent.EventType = GAME_OVER;
                    StrategyEvent.EventParam = 0;
                    PostStrategyFSM(StrategyEvent);

                    PowerDown();

                    CurrentState = GameOver;
                }
        }
```

```
            break;

        case GameOver :

            break; //end GameOver State

        }// end switch on Current State
    return ReturnEvent;
}

/****************************************************************************
 Function
    QueryStrategySM

 Parameters
    None

 Returns
    StrategyState_t The current state of the Strategy state machine

 Description
    returns the current state of the Strategy state machine
 Notes

 Author
    Jina Wang 3/8/2013
****************************************************************************/
StrategyState_t QueryStrategyFSM ( void )
{
    return(CurrentState);
}

/****************************************************************************
public functions
 ****************************************************************************/
unsigned char GetDesiredX( void )
{
    return Desiredx;
}

unsigned char GetDesiredY( void )
{
    return Desiredy;
}

unsigned char QueryEnemy( void )
{
    return EnemyBot;
}

static void PowerDown( void )
{
    ES_Event MtrEvent;

    //turn off lights/stop motor/power down etc
    ACTIVELED_PORT = LO; //turn off GameActive Light
    puts("game over");
    FlywheelOff(); //turn off flywheel
    EnableIRDetection(False); //turn off PhotoNPN
    MtrEvent.EventType = STOP_MOTOR; //stop motor
```

```
    MtrEvent.EventParam = 0;
    PostDriveTrainService(MtrEvent);
}
```