```c
/****************************************************************************/
Cleaned up by Huajian Huang on 16:36, March 11th, 2017
/****************************************************************************/


#include "ES_Configure.h"
#include "ES_Framework.h"
#include "LOCMaster.h"

// some includes used in the SPI.c from Lab 8
#include "ES_DeferRecall.h"
//#include "SPI.h"
//#include "BaseControl.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_gpio.h"
#include "inc/hw_sysctl.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"  // Define PART_TM4C123GH6PM in project
#include "driverlib/gpio.h"
//#include "ES_ShortTimer.h"
#include "inc/hw_timer.h"
#include "inc/hw_ssi.h"
#include "inc/hw_nvic.h"
#include "OwnPWM.h"
#include "MasterVehicle.h"
//some testing constants
//#define TEST_LOC_STAGE

/*----------------------------- Module Defines ----------------------------*/
#define GET_STATUS_PERIOD 350 //game status updates at 3Hz, no need to go faster

#define GET_STATUS_COMMAND 0xc0 //=0b11000000,  0b xxxx xxxx means binary with 8 bits
#define REPORT_FRONT_BITS 0x80 //=0b1000 0000, we put the frequency report in the second 4 bit
//#define GET_STATUS_COMMAND 0xAA // fake command, just for testing
//#define GET_RR_RS_COMMAND 0x70 //=0b01110000

#define TWO_HUNDRED_MS 300 //should be 200, but I am faking so it's at least 200ms
//#define TWO_HUNDRED_MS 2000 // this is a fake time for debugging
#define RESPONSE_READY_CODE 0XAA
#define RESPONSE_NOT_READY_CODE 0x00
#define ACK_CODE 0x00 //NEED TO ADJUST AND ONLY GRAB THE 6,7TH BIT
#define NACK_CODE 0xC0
#define REPORT_RESPONSE_COMMAND 112//0x70 =0b 0111 0000
#define GAME_STATUS_SB3_GS_EXTRACT_MASK 0x00000080
#define GAME_STATUS_CSG_MASK 0x00800000
#define GAME_STATUS_CSR_MASK 0X00080000
#define GAME_STATUS_GREEN_ACTIVE_STATUS_MASK 0x00700000
#define GAME_STATUS_RED_ACTIVE_STATUS_MASK   0x00070000
#define GAME_STATUS_GREEN_LOC_NONE_MASK  0x00000000
#define GAME_STATUS_GREEN_LOC_ONE_MASK   0x00100000
#define GAME_STATUS_GREEN_LOC_TWO_MASK   0x00200000
#define GAME_STATUS_GREEN_LOC_THREE_MASK 0x00300000
#define GAME_STATUS_GREEN_LOC_ALL_MASK   0x00400000
#define GAME_STATUS_RED_LOC_NONE_MASK    0x00000000
#define GAME_STATUS_RED_LOC_ONE_MASK     0x00010000
#define GAME_STATUS_RED_LOC_TWO_MASK     0x00020000
#define GAME_STATUS_RED_LOC_THREE_MASK   0x00030000
#define GAME_STATUS_RED_LOC_ALL_MASK     0x00040000
#define GOAL_SCORE_GREEN_MASK 0x00003f00
#define GOAL_SCORE_RED_MASK   0x0000003f
#define GOAL_SCORE_GREEN_OFFSET 8
// these times assume a 1.000mS/tick timing
#define TEN_MSEC 10
#define ONE_SEC 976
#define HALF_SEC (ONE_SEC /2)
#define TWO_SEC (ONE_SEC *2)
#define FIVE_SEC (ONE_SEC *5)
#define BitsPerNibble 4
#define CPSDVSR_PRESCALER 128
#define SCR 60 //date rate should be 1/60 of the Lab 8's data rate, due to Tcy

```

```c
#define RED 0
#define GREEN 1
#define INVALID_LOCATIONS_ACTIVE 5
//input capture realted
#define CHECK_ACTIVE_ALWAYS
//the input would be in ticks, 4*10^7 ticks in one sec, period table is in micro seconds,
//4*10^7 ticks in one sec is 4*10^7 ticks in 10^6 micro
#define TicksToMicroSecDivisor 40
#define FreqErrorThreshold 25

//#define TEAM_GREEN
//#define TEAM_RED

/*---------------------------- Module Functions ----------------------------*/
static ES_Event DuringStateOne( ES_Event Event);
static ES_Event DuringWaitingState( ES_Event Event);
static void enable_SPI_Interupt(void);
static ES_Event During_GAME_STATUS_SENDING_TO_LOC_State( ES_Event Event);
static ES_Event During_GAME_STATUS_RECEIVING_FROM_LOC_State( ES_Event Event);
static ES_Event During_SENDING_TO_LOC_AT_STAGING_State( ES_Event Event);
static ES_Event During_RECEIVING_FROM_LOC_AT_STAGING_State( ES_Event Event);
static ES_Event DuringWait200msState(ES_Event CurrentEvent);
/*---------------------------- 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 LOCMasterState_t CurrentState;
// with the introduction of Gen2, we need a module level Priority var as well
static uint8_t MyPriority;

//our command and response have 4 bytes in them
static uint8_t data1;
static uint8_t data2;
static uint8_t data3;
static uint8_t data4;
static uint8_t data5;
static uint8_t dataBuffer;

//storing the status bytes, so we can detect a change
static uint32_t StatusBytes = 0;
static uint32_t prevStatusBytes=0;
//active shooting or staging location
static uint8_t ActiveLocation=0;
//index used to access the list of frequency to have
static uint8_t frq_index = 0;

//serveral flags indicating which step of the handshake we are doing
static uint8_t FlagSentOneReport=0;
static uint8_t FlagSentTwoReport=0;
static uint8_t FlagSentReportWithin200ms=0;
static uint8_t FlagFirstReportACK=0;
static uint8_t FlagSecondReportACK=0;

//some input capture variables for the hall sensor, make sure to limit their scope
static uint32_t ThisCapture=0;
static uint32_t LastCapture=0;
static uint32_t ThisPeriod=0;
static uint32_t FlagOneShotTimeout=0;

//define the frequency related constants and variable
//the PERIOD_TABLE is in micro seconds
static uint32_t const PERIOD_TABLE[16]={1333,1277,1222,1166,1111,1055,1000,944,889,833,778,722,667,611,556,500};

static uint8_t FreqCode[16]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0X0B,0x0C,0x0D,0x0E,0x0F};
static uint8_t period_table_length=16;

static uint8_t FlagValidFreq=0; //raised to 1 when we detect a frequency in the specified table
static uint8_t FlagFreqConsumed=1; //init to "consumed", no freq needs to be processed, when it's ACKed, it's consumed
static uint8_t measFreqCode=0xf0; //0xf0 means there's no valid frequency
```

```c
static uint8_t prevMeasFreqCode = 0xf0;
static uint8_t stableMeasFreqCode=0xf0;
static uint8_t numOfMeasurementsForStable=10;
static uint8_t stableCounter=0;

//define some variables to sotre the information that would be used/queried by MasterVehicle
static uint8_t ActiveGreenStagingLocation=0;
static uint8_t prevActiveGreenStagingLocation=0;
static uint8_t ActiveGreenShootingLocation=0;
static uint8_t prevActiveGreenShootingLocation=0;
static uint8_t ActiveRedLocation=0;
static uint8_t prevActiveRedLocation=0;

static uint8_t prev_green_score = 0;
static uint8_t prev_red_score = 0;
static uint8_t cur_green_score = 0;
static uint8_t cur_red_score = 0;
//initialize a event to carry information around
static ES_Event EventToPost;
/*----------------------------- Module Code -----------------------------*/
/****************************************************************************

****************************************************************************/
bool InitLOCMasterSM ( uint8_t Priority )
{
  TERMIO_Init();
  ES_Event ThisEvent;

  MyPriority = Priority;  // save our priority

  ThisEvent.EventType = ES_ENTRY;
  //ThisEvent.EventType = ARRIVED_AT_STAGING;//I cannot print to my Mac, so go directly when testing with MAC
  // Start the Master State machine
  SPI_Init();
  InitInputCapture_Hall();
  StartLOCMasterSM( ThisEvent );
  ES_Timer_InitTimer(GET_STATUS_TIMER,GET_STATUS_PERIOD);
  return true;
}

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

****************************************************************************/
bool PostLOCMasterSM( ES_Event ThisEvent )
{
  return ES_PostToService( MyPriority, ThisEvent);
}

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

****************************************************************************/
ES_Event RunLOCMasterSM( ES_Event CurrentEvent )
{
  bool MakeTransition = false;/* are we making a state transition? */
  LOCMasterState_t NextState = CurrentState;
  ES_Event EntryEventKind = { ES_ENTRY, 0 };// default to normal entry to new state
  ES_Event ReturnEvent = { ES_NO_EVENT, 0 }; // assume no error

  //before everything, clear the 200ms flag if it happens
  //regardless of states, I almost want to use oneshot timer for this
  //   if ((CurrentEvent.EventType==ES_TIMEOUT) && (CurrentEvent.EventParam==REPORT_TIMER)){
  //    FlagSentReportWithin200ms=0;
  //   }

  //5 main states, WAITING, GAME_STATUS_SENDING_TO_LOC, GAME_STATUS_RECEIVING_FROM_LOC,
  // SENDING_TO_LOC_AT_STAGING, RECEIVING_FROM_LOC_AT_STAGING, WAITING_FOR_200MS_TIMEOUT

  //separates the regular game status query and the special procedures in staging


  switch ( CurrentState )
```

```
    {
      case WAITING :

      // This state is like a neutral transition state
      //(1) normally wait for timer and keep querying game status
      //(2) when asked to do staging area stuff, move to the corresponding states

      //In this state, I want to keep querying the game status regularly
      CurrentEvent = DuringWaitingState(CurrentEvent);
      //process any events
      if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
      {
        switch (CurrentEvent.EventType)
        {
          case ES_TIMEOUT: //If event is "The GET_STATUS_TIMER" timeout
          if (CurrentEvent.EventParam==GET_STATUS_TIMER){
            //printf("Timeout in waiting\r\n");
            NextState = GAME_STATUS_SENDING_TO_LOC;//Decide what the next state will be
            MakeTransition = true; //mark that we are taking a transition
            //Post the same event to self
            PostLOCMasterSM(CurrentEvent);
            // optionally, consume or re-map this event for the upper
            // level state machine
            ReturnEvent.EventType = ES_NO_EVENT;
          }
          break;


          case ARRIVED_AT_STAGING:
          //printf("Arrive at staging\r\n");
          NextState = SENDING_TO_LOC_AT_STAGING;//Decide what the next state will be
          // for internal transitions, skip changing MakeTransition
          MakeTransition = true; //mark that we are taking a transition
          // optionally, consume or re-map this event for the upper
          // level state machine
          ReturnEvent.EventType = ES_NO_EVENT;
          break;

          default:
          break;

        }
      } // end if "No event"
      else // Current Event is now ES_NO_EVENT. Correction 2/20/17 provided by Prof.Ed
      {     //Probably means that CurrentEvent was consumed by lower level
        ReturnEvent = CurrentEvent; // in that case update ReturnEvent too.
      }
      break;

      case GAME_STATUS_SENDING_TO_LOC :
      //printf("Enter Game status sending to LOC state\n\r");
      // 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 = During_GAME_STATUS_SENDING_TO_LOC_State(CurrentEvent);
      //process any events
      if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
      {
        switch (CurrentEvent.EventType)
        {
          case ES_TIMEOUT: //If event is GET_STATUS timer timeout
          if(CurrentEvent.EventParam==GET_STATUS_TIMER){
            //printf("Enter ES_timerout case for game status sending to loc\r\n");
            // Execute action function for state one
            //time to  query for current game status
            //write the command 0xC0 and followed by 4 bytes of 0x00
            //Pump them into FIFO buffer, when they are all out, we get EOT interrupt
            //printf("Sending GET_STATUS_COMMAND\r\n");
            SPI_Write(GET_STATUS_COMMAND);
            SPI_Write(0X00);
            SPI_Write(0X00);
```

```c
            SPI_Write(0X00);
            SPI_Write(0X00);

            //enable the EOT interrupt
            enable_SPI_Interupt();
            //Decide what the next state will be
            NextState = GAME_STATUS_RECEIVING_FROM_LOC;//neutralize to WAITING, we might need to do staging area stuff anytime
            // for internal transitions, skip changing MakeTransition
            MakeTransition = true; //mark that we are taking a transition
            // optionally, consume or re-map this event for the upper
            // level state machine
            ReturnEvent.EventType = ES_NO_EVENT;
          }
          break;
          // repeat cases as required for relevant events
          case ARRIVED_AT_STAGING:
          ES_Event to_post;
          printf("Repost arrive at staging to LOC Master\r\n");
          to_post.EventType = ARRIVED_AT_STAGING;
          PostLOCMasterSM(to_post);
          ReturnEvent.EventType = ES_NO_EVENT;
          break;


          default:
          break;
        }
      } //end if "no event"
      else // Current Event is now ES_NO_EVENT. Correction 2/20/17
      {     //Probably means that CurrentEvent was consumed by lower level
        ReturnEvent = CurrentEvent; // in that case update ReturnEvent too.
      }
      break;
      // repeat state pattern as required for other states

      case GAME_STATUS_RECEIVING_FROM_LOC :
      //printf("Enter Game status receiving from LOC state\n\r");
      // 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 = During_GAME_STATUS_RECEIVING_FROM_LOC_State(CurrentEvent);
      //process any events
      if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
      {
        switch (CurrentEvent.EventType)
        {
          case ES_EOT: //If event is end of transmission of 5 bytes, provided by EOT interrupt
          // Execute action function for state one
          //printf("Received EOT in receiving from loc state\r\n");
          if (data2==0xff){ //all of the useful commands start with 0xff
            //after ensuring that this data is good by seeing a 0xff byte, assemble the bytes
            StatusBytes = (data2<<24) | (data3<<16) | (data4<<8) | data5 ; //0xff,SB1,SB2,SB3 respectively
            ;


            //                      printf("\n Byte 1 received: 0x%02x\r\n", data1);//put a lot of spaces around this so I can find it easily
            //                      printf("Byte 2 received: 0x%02x\r\n", data2);
            //                      printf("Byte 3 received: 0x%02x\r\n", data3);
            //                      printf("Byte 4 received: 0x%02x\r\n", data4);
            //                      printf("Byte 5 received: 0x%02x\r\n\n", data5);
            //

            //can post an event to Master vehicle telling we updated the game status

            //extract the but corresponding to contruction start and see if it changes to "construction starts"
            if ( (StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)>(prevStatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)){
              EventToPost.EventType=CONSTRUCTION_START;
              PostMasterVehicleSM(EventToPost);
```

```c
            printf("Post CONSTRUCTION_START to master vehicle\n\r");
          }

          //extract the but corresponding to contruction start and see if it changes to "construction ends"
          if ( (StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)<(prevStatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)){
            EventToPost.EventType=CONSTRUCTION_END;
            PostMasterVehicleSM(EventToPost);
            printf("Post CONSTRUCTION_END to master vehicle\n\r");
          }

          //updating current score
          if(get_Team() == GREEN){
            cur_green_score = queryGoalGreen();
            if(cur_green_score > prev_green_score){
              ES_Event score_post;
              score_post.EventType=SCORE_CHANGED;
              PostMasterVehicleSM(score_post);
              printf("Post SCORE_CHANGED to master vehicle\n\r");
            }
            prev_green_score =  cur_green_score;
          }
          else{
            cur_red_score = queryGoalRed();
            if(cur_red_score > prev_red_score){
              ES_Event red_score_post;
              red_score_post.EventType=SCORE_CHANGED;
              PostMasterVehicleSM(red_score_post);
              printf("Post SCORE_CHANGED to master vehicle\n\r");
            }
            prev_red_score=  cur_red_score;
          }
          update_score(queryGoalRed(), queryGoalGreen());

          #ifdef CHECK_ACTIVE_ALWAYS
          if ((StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)>0){//if the game is active

            if(get_Team() == GREEN){
              // Check active staging location
              ActiveGreenStagingLocation=queryActiveStagingGreen();

              if ((ActiveGreenStagingLocation!=0)&&(ActiveGreenStagingLocation!=INVALID_LOCATIONS_ACTIVE)&&(ActiveGreenStagingLocation!=prevActiveGreenStagingLocation)){
                //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE, do this so it doesn't keep posting "STAGE_ACTIVE" events
                //it's a valid location, and it's different from before
                EventToPost.EventType=STAGE_ACTIVE;
                EventToPost.EventParam=ActiveGreenStagingLocation;
                PostMasterVehicleSM(EventToPost);
                printf("\r\nActive Green Staging Location is: %d",ActiveGreenStagingLocation);
              }
              prevActiveGreenStagingLocation=ActiveGreenStagingLocation;

              // Check active shooting location
              ActiveGreenShootingLocation=queryActiveShootingGreen();
              if ((ActiveGreenShootingLocation!=0)&&(ActiveGreenShootingLocation!=INVALID_LOCATIONS_ACTIVE)&&(ActiveGreenShootingLocation!=prevActiveGreenShootingLocation)){
                //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE, do this so it doesn't keep posting "SHOOT_ACTIVE" events
                //it's a valid location and it's different from before

                if(ActiveGreenShootingLocation==4){ //a "4" code implies that all are active
                  printf("Post last 18 second shooting active (SHOOT_ACTIVE_4)\r\n");
                  EventToPost.EventType=SHOOT_ACTIVE_4;
                  PostMasterVehicleSM(EventToPost);
                  printf("\r\nActive Green Shooting Location is: %d",ActiveGreenShootingLocation);
                }
                else{ //it's a normal 1,2,3 shooting location
                  EventToPost.EventType=SHOOT_ACTIVE;
```

```
                    EventToPost.EventParam=ActiveGreenShootingLocation;
                    PostMasterVehicleSM(EventToPost);
                    printf("\r\nActive Green Shooting Location is: %d",ActiveGreenShootingLocation);
                  }
                }
                prevActiveGreenShootingLocation=ActiveGreenShootingLocation;
              }
              //#endif

              //#ifdef TEAM_RED
              if(get_Team() == RED){ //same procedure and reasoning as in green, just different function names

                // Check staging
                ActiveRedLocation=queryActiveStagingRed();

                if ((ActiveRedLocation!=0)&&(ActiveRedLocation!=INVALID_LOCATIONS_ACTIVE)&&(ActiveRedLocation!=prevActiveRedLocation)){
                  //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE
                  //it's a valid location at a valid, and it's different from before
                  EventToPost.EventType=STAGE_ACTIVE;
                  EventToPost.EventParam=ActiveRedLocation;
                  PostMasterVehicleSM(EventToPost);
                  printf("\r\nActive Red stagingLocation is: %d",ActiveRedLocation);

                }
                prevActiveRedLocation=ActiveRedLocation;

                // Check shooting
                ActiveRedLocation=queryActiveShootingRed();
                if ((ActiveRedLocation!=0)&&(ActiveRedLocation!=INVALID_LOCATIONS_ACTIVE)&&(ActiveRedLocation!=prevActiveRedLocation)){
                  //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE
                  //it's a valid location at a valid, and it's different from before
                  if(ActiveRedLocation==4){
                    printf("Post last 18 second shooting active (SHOOT_ACTIVE_4)\r\n");
                    EventToPost.EventType=SHOOT_ACTIVE_4;
                    PostMasterVehicleSM(EventToPost);
                    printf("\r\nActive Red Shooting Location is: %d",ActiveRedLocation);
                  }
                  else{
                    EventToPost.EventType=SHOOT_ACTIVE;
                    EventToPost.EventParam=ActiveRedLocation;
                    PostMasterVehicleSM(EventToPost);
                    printf("\r\nActive Red Shooting Location is: %d",ActiveRedLocation);
                  }

                }
                prevActiveRedLocation=ActiveRedLocation;
              }//end if "red"
              //#endif
            }//end "if the game is active"
            #endif
          }//end if "data is useful, data2=0xff"


          //update prevStatusBytes after this comparison
          prevStatusBytes=StatusBytes;
          //start the timer for the next game status query
          ES_Timer_InitTimer(GET_STATUS_TIMER,GET_STATUS_PERIOD);
          NextState = WAITING;//Decide what the next state will be
          // for internal transitions, skip changing MakeTransition
          MakeTransition = true; //mark that we are taking a transition
          // optionally, consume or re-map this event for the upper
          // level state machine
          ReturnEvent.EventType = ES_NO_EVENT;
          break;
          // repeat cases as required for relevant events


```

```
        case ARRIVED_AT_STAGING: //mainly used when we are stuck, at we are moving around a bit and restarting
        ES_Event to_post;
        printf("Repost arrive at staging to LOC Master\r\n");
        to_post.EventType = ARRIVED_AT_STAGING;
        PostLOCMasterSM(to_post);
        ReturnEvent.EventType = ES_NO_EVENT;
        break;
        default:
        break;
      }
    }// end if "no event"
    else // Current Event is now ES_NO_EVENT. Correction 2/20/17
    {     //Probably means that CurrentEvent was consumed by lower level
      ReturnEvent = CurrentEvent; // in that case update ReturnEvent too.
    }
    break;




    case SENDING_TO_LOC_AT_STAGING :       // 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 = During_SENDING_TO_LOC_AT_STAGING_State(CurrentEvent);
    //process any events
    if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
    {
      switch (CurrentEvent.EventType)
      {


        //early testing code
        #ifdef TEST_LOC_STAGE
        case Input_Freq:
        frq_index++;
        if(frq_index >= period_table_length){
          frq_index = 0;
        }
        printf("Select Period is %d, Code is %d\r\n", PERIOD_TABLE[frq_index],
        FreqCode[frq_index]);
        break;
        case SEND_FIRST_REPORT_WITH_OUR_FAKE_FREQ:
        ES_Event EventToPostFake;
        EventToPostFake.EventType=SEND_FIRST_REPORT;
        EventToPostFake.EventParam=FreqCode[frq_index];
        PostLOCMasterSM(EventToPostFake);
        break;
        case SEND_SECOND_REPORT_WITH_OUR_FAKE_FREQ:
        ES_Event EventToPostFake2;
        EventToPostFake2.EventType=SEND_SECOND_REPORT;
        EventToPostFake2.EventParam=FreqCode[frq_index];
        PostLOCMasterSM(EventToPostFake2);
        break;
        #endif




        case SEND_FIRST_REPORT: //if we are asked to send the first report
        //clear the active location first, we are at a new round
        ActiveLocation=0;
        printf("Enter sending first report\n\r");
        //clear all the flags for previous report sent and ack, might be repeated, but better safe than sorry
        FlagFirstReportACK=0;
        FlagSecondReportACK=0;
        FlagSentOneReport=0;
        FlagSentTwoReport=0;

```

```c
          // Execute action function for state one :
          if (FlagSentReportWithin200ms==1)
          {
            //printf("Waiting for 200ms\r\n");
            NextState = WAITING_FOR_200MS_TIMEOUT;//Decide what the next state will be
            // for internal transitions, skip changing MakeTransition
            MakeTransition = true; //mark that we are taking a transition
            // optionally, consume or re-map this event for the upper
            // level state machine
            ReturnEvent.EventType = ES_NO_EVENT;
          }
          else if(FlagFreqConsumed==0){//we have a freq to use
            //write the frequency to LOC

            printf("Writing first frequency to LOC\r\n");
            printf("Event param is %d\r\n", CurrentEvent.EventParam);
            printf("Frequency report is: %d \n\r",(REPORT_FRONT_BITS)|(CurrentEvent.EventParam));

            SPI_Write((REPORT_FRONT_BITS)|(CurrentEvent.EventParam)); //REPORT_FRONT_BITS=0x80 is 0b10000000, the event parameter is
            //passed from input capture response
            //basically assembling the package with our frequency report
            SPI_Write(0x00);
            SPI_Write(0x00);
            SPI_Write(0x00);
            SPI_Write(0x00);

            //enable the EOT interrupt
            enable_SPI_Interupt();

            //raised the flag, we have sent 1 already
            FlagSentOneReport=1;

            //raise the flag, indicating that we have sent a report, don't
            //send again for another 200 ms
            FlagSentReportWithin200ms=1;
            //start a 200 ms timer to clear this FlagSentReprotWithin200ms
            ES_Timer_InitTimer(REPORT_TIMER,TWO_HUNDRED_MS);
            NextState = RECEIVING_FROM_LOC_AT_STAGING;//Decide what the next state will be
            // for internal transitions, skip changing MakeTransition
            MakeTransition = true; //mark that we are taking a transitio
            // optionally, consume or re-map this event for the upper
            // level state machine
            ReturnEvent.EventType = ES_NO_EVENT;
          }//end the within 200ms check
          break;
          // repeat cases as required for relevant events

          case SEND_SECOND_REPORT: //if we are asked to send the second report
          printf("Enter sending second report\n\r");
          // Execute action function for state one :
          if (FlagSentReportWithin200ms==1)
          {
            NextState = WAITING_FOR_200MS_TIMEOUT;//Decide what the next state will be
            // for internal transitions, skip changing MakeTransition
            MakeTransition = true; //mark that we are taking a transition
            // optionally, consume or re-map this event for the upper
            // level state machine
            ReturnEvent.EventType = ES_NO_EVENT;
          }
          else if(FlagFreqConsumed==0){//if we have a freq to use

            //write the frequency to LOC
            printf("Writing second frequency to LOC\r\n");
            printf("Frequency report is: 0x%02x \n\r",(REPORT_FRONT_BITS)|(CurrentEvent.EventParam));
            SPI_Write((REPORT_FRONT_BITS)|(CurrentEvent.EventParam)); //REPORT_FRONT_BITS=0x80 is 0b10000000,
            //the event parameter is passed from input capture response
            SPI_Write(0x00);
            SPI_Write(0x00);
            SPI_Write(0x00);
            SPI_Write(0x00);
```

```
            //enable the EOT interrupt
            enable_SPI_Interupt();


            //raised the flag, we have sent a "second report" already
            FlagSentTwoReport=1; //only difference with SENT_ONE_REPORT
            //raise the flag, indicating that we have sent a report, don't
            //send again for another 200 ms
            FlagSentReportWithin200ms=1;
            //start a 200 ms timer to clear this Flag
            ES_Timer_InitTimer(REPORT_TIMER,TWO_HUNDRED_MS);
            NextState = RECEIVING_FROM_LOC_AT_STAGING;//Decide what the next state will be
            // for internal transitions, skip changing MakeTransition
            MakeTransition = true; //mark that we are taking a transition
            // optionally, consume or re-map this event for the upper
            // level state machine
            ReturnEvent.EventType = ES_NO_EVENT;
          }//end the within 200ms check
          break;
          // repeat cases as required for relevant events

          case GO_QUERY_REPORT_RESPONSE : //If event is event one
          printf("Asked to go query report response\n\r");
          // Execute action function
          //write the query for report response command to LOC
          SPI_Write(REPORT_RESPONSE_COMMAND);
          //SPI_Write(112);
          SPI_Write(0x00);
          SPI_Write(0x00);
          SPI_Write(0x00);
          SPI_Write(0x00);
          //enable the EOT interrupt
          enable_SPI_Interupt();
          NextState = RECEIVING_FROM_LOC_AT_STAGING;//Decide what the next state will be
          // for internal transitions, skip changing MakeTransition
          MakeTransition = true; //mark that we are taking a transition
          // optionally, consume or re-map this event for the upper
          // level state machine
          ReturnEvent.EventType = ES_NO_EVENT;
          break;
          // repeat cases as required for relevant events


          default:
          break;
        }

      }// end if "not no-event"
      else // Current Event is now ES_NO_EVENT. Correction 2/20/17
      {     //Probably means that CurrentEvent was consumed by lower level
        ReturnEvent = CurrentEvent; // in that case update ReturnEvent too.
      }
      break;

      case RECEIVING_FROM_LOC_AT_STAGING :       // If current state is state one
      printf("Enter receiving from LOC at staging state\n\r");
      // 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 = During_RECEIVING_FROM_LOC_AT_STAGING_State(CurrentEvent);
      //process any events
      if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
      {
        switch (CurrentEvent.EventType)
        {


          //some test code to use keyboard to step through the process
          #ifdef TEST_LOC_STAGE
          case Set_SentOne_Flag_1:
          FlagSentOneReport = 1;
          printf("FlagSentOneReport = 1\n\r");
```

```
            break;
            case Set_SentOne_Flag_0:
            FlagSentOneReport = 0;
            printf("FlagSentOneReport = 0\n\r");
            break;
            case Set_SentTwo_Flag_1:
            FlagSentTwoReport = 0;
            printf("FlagSentTwoReport = 0\n\r");
            break;
            case Set_SentTwo_Flag_0:
            FlagSentTwoReport = 0;
            printf("FlagSentTwoReport = 0\n\r");
            break;
            case Set_first_ack_1:
            FlagFirstReportACK = 1;
            printf("FlagFirstReportACK = 1\n\r");
            break;
            case Set_first_ack_0:
            FlagFirstReportACK = 0;
            printf("FlagFirstReportACK = 0\n\r");
            break;
            case Set_second_ack_1:
            FlagSecondReportACK = 0;
            printf("FlagSecondReportACK = 0\n\r");
            break;
            case Set_second_ack_0:
            FlagSecondReportACK = 0;
            printf("FlagSecondReportACK = 0\n\r");
            break;
            case Set_Data3_Ready:
            data3 = RESPONSE_READY_CODE;
            printf("data3 = RESPONSE_READY_CODE\n\r");
            break;
            case Set_Data3_Notready:
            data3 = RESPONSE_NOT_READY_CODE;
            printf("data3 = RESPONSE_NOT_READY_CODE\n\r");
            break;
            case Set_Data4_ACK:
            data4=ACK_CODE;
            printf("data4== ACK_CODE\n\r");
            break;
            case Set_Data4_NACK:
            data4= NACK_CODE;
            printf("data4= NACK_CODE\n\r");
            break;

            #endif




            case RESTART_VERIFY_FREQ://we are stuck, restart the process
            NextState = WAITING;
            MakeTransition = true;
            ReturnEvent.EventType = ES_NO_EVENT;
            break;


            case GO_QUERY_REPORT_RESPONSE:
            printf("Send QUERY to LOC\r\n");
            SPI_Write(REPORT_RESPONSE_COMMAND);//112 works
            SPI_Write(0x00);
            SPI_Write(0x00);
            SPI_Write(0x00);
            SPI_Write(0x00);
            //enable the EOT interrupt
            enable_SPI_Interupt();
            break;

            case ES_EOT : //If event is ES_EOT, END OF FIVE BYTES, not one, FIVE!
            // Execute action function for state one : event one

```

```c

        //printf("\n B1 received staging: 0x%02x\r\n", data1);//put a lot of spaces around this so I can find it easily
        //printf("B2 received staging: 0x%02x\r\n", data2);
        //printf("B3 received staging: 0x%02x\r\n", data3);
        //printf("B4 received staging: 0x%02x\r\n", data4);
        //printf("B5 received staging: 0x%02x\r\n\n", data5);


        printf("Stable measured freq code: %d\n\n\r",stableMeasFreqCode);


        //if we have sent the first report and not acknowledged
        if ( (FlagSentOneReport==1) && (FlagFirstReportACK==0)){
          //printf("Enter Sent one report, first report NOT ACK\n\r");
          //Extract to see if the response is ready
          if (data3==RESPONSE_READY_CODE){ //data3=RR, response ready code is 0xAA
            //consume the frequency once the reseponse is ready and our frequency is processed
            FlagFreqConsumed=1;
            if ((data4&(BIT7HI|BIT6HI))==ACK_CODE){ //data4 is RS byte, ACK is 0x00
              //printf("First report ACK\n\r");
              //update the flags to encode our current step
              FlagSentOneReport=1;
              FlagSentTwoReport=0;
              FlagFirstReportACK=1;
              FlagSecondReportACK=0;

              NextState=SENDING_TO_LOC_AT_STAGING;//update the next state
              //
              MakeTransition=true;
              ReturnEvent.EventType=ES_NO_EVENT;
            }
            else{ //we get NACK or Inactive, first report failed
              printf("First Report NACK\n\r");
              //update the flags to encode our current step, we go back to the beginning
              FlagSentOneReport=0;
              FlagSentTwoReport=0;
              FlagFirstReportACK=0;
              FlagSecondReportACK=0;
              //just consume the frequency and don't post yet, let the input capture do the posting
              FlagFreqConsumed=1;
              NextState=SENDING_TO_LOC_AT_STAGING;

              MakeTransition=true;
              ReturnEvent.EventType=ES_NO_EVENT;
            }

          }// ends the "checking response ready" if statement
          else{//the reponse code is not ready, keep querying
            printf("Response is NOT READY\n\r");
            NextState=SENDING_TO_LOC_AT_STAGING;
            //keep querying until we get response ready byte
            EventToPost.EventType=GO_QUERY_REPORT_RESPONSE;
            PostLOCMasterSM(EventToPost);

            MakeTransition=true;
            ReturnEvent.EventType=ES_NO_EVENT;
          }
        }
        //end of after first report and dealing with first ACK

        //now deal with after having one successful report
        if ( (FlagSentTwoReport==1) && (FlagSecondReportACK==0)){
          printf("Enter sent two reports, second one NOT ACK\n\r");
          //Extract to see if the response is ready
          if (data3==RESPONSE_READY_CODE){ //data3=RR, response ready code is 0xAA
            //consume the frequency once the reseponse is ready and our frequency is processed
            FlagFreqConsumed=1;
            if ((data4&(BIT7HI|BIT6HI))==ACK_CODE){ //data4 is RS byte, ACK is 0x00
              printf("Second report ACK\n\r");
              //update the flags to encode our current step
              FlagFreqConsumed=1;//consume the frequency after ACKed
```

```
                FlagSentOneReport=1;
                FlagSentTwoReport=1;
                FlagFirstReportACK=1;
                FlagSecondReportACK=1; //second report is successful
                //extract the active location
                ActiveLocation= (data5 & (BIT0HI|BIT1HI|BIT2HI|BIT3HI));
                NextState=WAITING;
                EventToPost.EventType=FINISHED_STAGING;
                //FlagFreqConsumed=0;//consume the frequency after ACKed
                printf("Finished Staging\n\r");
                printf("Posting finished staging to mastervehicle SM\r\n");
                PostMasterVehicleSM(EventToPost);//tell the master vehicle that we are done
                //start the timer for the next game status query otherwise it wouldn't get triggered
                ES_Timer_InitTimer(GET_STATUS_TIMER,GET_STATUS_PERIOD);
                MakeTransition=true;
                ReturnEvent.EventType=ES_NO_EVENT;
              }
              else{ //we get NACK or Inactive
                printf("Failed at 2nd report, NACK\n\r");
                //update the flags to encode our current step
                FlagSentOneReport=0;
                FlagSentTwoReport=0;
                FlagFirstReportACK=0;
                FlagSecondReportACK=0;
                NextState=SENDING_TO_LOC_AT_STAGING;
                //just consume the frequency and don't post yet, let the input capture do the posting
                FlagFreqConsumed=1;
                MakeTransition=true;
                ReturnEvent.EventType=ES_NO_EVENT;
              }

            }
            else{//the reponse code is not ready, keep querying
              printf("Response is NOT READY\n\r");
              NextState=SENDING_TO_LOC_AT_STAGING;
              //keep querying until we get response byte
              EventToPost.EventType=GO_QUERY_REPORT_RESPONSE;
              PostLOCMasterSM(EventToPost);
              MakeTransition=true;
              ReturnEvent.EventType=ES_NO_EVENT;
            }
          } //end of after first report and dealing with first ACK

          break;
          // repeat cases as required for relevant events
          default:
          break;
        }
      } // end if "no event"
      else // Current Event is now ES_NO_EVENT. Correction 2/20/17
      {     //Probably means that CurrentEvent was consumed by lower level
        ReturnEvent = CurrentEvent; // in that case update ReturnEvent too.
      }
      break;

      case WAITING_FOR_200MS_TIMEOUT :       // If current state is waiting for 200ms timeout (due to report)

      // 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 = DuringWait200msState(CurrentEvent);


      //process any events
      if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
      {
        switch (CurrentEvent.EventType)
        {
```

```
910             #ifdef TEST_LOC_STAGE
911             case Set_SentOne_Flag_1:
912             FlagSentOneReport = 1;
913             printf("FlagSentOneReport = 1\n\r");
914             break;
915             case Set_SentOne_Flag_0:
916             FlagSentOneReport = 0;
917             printf("FlagSentOneReport = 0\n\r");
918             break;
919             case Set_SentTwo_Flag_1:
920             FlagSentTwoReport = 0;
921             printf("FlagSentTwoReport = 0\n\r");
922             break;
923             case Set_SentTwo_Flag_0:
924             FlagSentTwoReport = 0;
925             printf("FlagSentTwoReport = 0\n\r");
926             break;
927             case Set_first_ack_1:
928             FlagFirstReportACK = 1;
929             printf("FlagFirstReportACK = 1\n\r");
930             break;
931             case Set_first_ack_0:
932             FlagFirstReportACK = 0;
933             printf("FlagFirstReportACK = 0\n\r");
934             break;
935             case Set_second_ack_1:
936             FlagSecondReportACK = 0;
937             printf("FlagSecondReportACK = 0\n\r");
938             break;
939             case Set_second_ack_0:
940             FlagSecondReportACK = 0;
941             printf("FlagSecondReportACK = 0\n\r");
942             break;
943             case Set_Data3_Ready:
944             data3 = RESPONSE_READY_CODE;
945             printf("data3 = RESPONSE_READY_CODE\n\r");
946             break;
947             case Set_Data3_Notready:
948             data3 = RESPONSE_NOT_READY_CODE;
949             printf("data3 = RESPONSE_NOT_READY_CODE\n\r");
950             break;
951             case Set_Data4_ACK:
952             data4=ACK_CODE;
953             printf("data4== ACK_CODE\n\r");
954             break;
955             case Input_Freq:
956             frq_index++;
957             if(frq_index >= period_table_length){
958               frq_index = 0;
959             }
960             printf("Select Period is %d, Code is %d\r\n", PERIOD_TABLE[frq_index],
961             FreqCode[frq_index]);
962             break;
963
964             #endif
965
966
967
968
969             case RESTART_VERIFY_FREQ://restart when we are stuck
970             NextState = WAITING;
971             MakeTransition = true;
972             ReturnEvent.EventType = ES_NO_EVENT;
973             break;
974
975
976
977             case ES_TIMEOUT : //If event is the 200ms timeout
978             //MAYBE CHECK THAT IT IS ACTUALLY THE 200MS TIMEOUT HERE
979             if (CurrentEvent.EventParam==REPORT_TIMER){
980               FlagSentReportWithin200ms=0;
981               if ((FlagSentOneReport==0)&&
```

```
(FlagFirstReportACK==0)&&(FlagSentTwoReport==0)&&(FlagSecondReportACK==0)){//failed on any attempt, restarting
            // Execute action function for state one : event one
            NextState = SENDING_TO_LOC_AT_STAGING;//Decide what the next state will be
            //the program is sent to this state because when asked to send report, 200ms hasn't expired

            //just consumed the frequency and let input capture post
            FlagFreqConsumed=1;

            #ifdef TEST_LOC_STAGE
            EventToPost.EventParam=FreqCode[frq_index];
            #endif

            // for internal transitions, skip changing MakeTransition
            MakeTransition = true; //mark that we are taking a transition
            // optionally, consume or re-map this event for the upper
            // level state machine
            ReturnEvent.EventType = ES_NO_EVENT;
          }

          else if ((FlagSentOneReport==1)&&(FlagFirstReportACK==1)&&(FlagSentTwoReport==0)&&(FlagSecondReportACK==0)){//sent one report, ACKed first report, sent to this state when attempting to send the second report
            // Execute action function for state one : event one
            NextState = SENDING_TO_LOC_AT_STAGING;//Decide what the next state will be
            //the program is sent to this state because when asked to send report, 200ms hasn't expired

            //just consumed the frequency and let input capture post
            FlagFreqConsumed=1;

            //so we return to where the program came from
            //EventToPost.EventType=SEND_SECOND_REPORT;
            //EventToPost.EventParam=stableMeasFreqCode;
            #ifdef TEST_LOC_STAGE
            EventToPost.EventParam=FreqCode[frq_index];
            #endif
            //PostLOCMasterSM(EventToPost);
            // for internal transitions, skip changing MakeTransition
            MakeTransition = true; //mark that we are taking a transition
            // optionally, consume or re-map this event for the upper
            // level state machine
            ReturnEvent.EventType = ES_NO_EVENT;
          }

        }//end checking the timeout is from the 200ms report timer
        break; //break the timeout case
        // repeat cases as required for relevant events

        //
        default:
        break;//break default, for switching event type
      }//end switch event type


    } //end if "not no event"
    else // Current Event is now ES_NO_EVENT. Correction 2/20/17
    {     //Probably means that CurrentEvent was consumed by lower level
      ReturnEvent = CurrentEvent; // in that case update ReturnEvent too.
    }
    break; //break the waiting 200ms state

    default:
    break;//break default, for switching states
  } //end switch of states
  //   If we are making a state transition
  if (MakeTransition == true)
  {
    //   Execute exit function for current state
    CurrentEvent.EventType = ES_EXIT;
    RunLOCMasterSM(CurrentEvent);

    CurrentState = NextState; //Modify state variable
```

```c

      // Execute entry function for new state
      // this defaults to ES_ENTRY
      RunLOCMasterSM(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);
  }
  /****************************************************************************

  ****************************************************************************/
  void StartLOCMasterSM ( ES_Event CurrentEvent )
  {
    // if there is more than 1 state to the top level machine you will need
    // to initialize the state variable
    //our initial state is WAITING
    CurrentState = WAITING;
    // 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
    RunLOCMasterSM(CurrentEvent);
    return;
  }


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



  static ES_Event DuringWaitingState( ES_Event Event)
  {
    ES_Event 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
      // after that start any lower level machines that run in this state

      //printf("Enter Waiting of LOC Master SM\r\n");

      //set all the flags related to handshaking at staging area to default value
      FlagSentOneReport=0;
      FlagSentTwoReport=0;
      FlagFirstReportACK=0;
      FlagSecondReportACK=0;
      FlagSentReportWithin200ms=0;
      FlagFreqConsumed=1;

    }
    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
      // printf("Exit Waiting State\r\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);
  }

  static ES_Event During_GAME_STATUS_SENDING_TO_LOC_State( ES_Event Event)
  {
    ES_Event 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
      // after that start any lower level machines that run in this state
      //StartLowerLevelSM( Event );
      // printf("Enter GAME_STATUS_SENDING_TO_LOC State\r\n");
      // 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
      // printf("Exit GAME_STATUS_SENDING_TO_LOC State\r\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);
  }

  static ES_Event During_GAME_STATUS_RECEIVING_FROM_LOC_State( ES_Event Event)
  {
    ES_Event 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
      // after that start any lower level machines that run in this state
      //StartLowerLevelSM( Event );
      // printf("Enter GAME_STATUS_RECEIVING_FROM_LOC State\r\n");
      // 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
      // printf("Exit GAME_STATUS_RECEIVING_FROM_LOC State\r\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);
  }

  static ES_Event During_SENDING_TO_LOC_AT_STAGING_State( ES_Event Event)
  {
    ES_Event 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) )
    {   //printf("Enter Sending to Loc at staging state\n\r");
      // implement any entry actions required for this state machine
      // after that start any lower level machines that run in this state
      //StartLowerLevelSM( Event );
      //printf("Enter SENDING_TO_LOC_AT_STAGING State\r\n");
      // 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
      //printf("Exit SENDING_TO_LOC_AT_STAGING State\r\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);
  }

  static ES_Event During_RECEIVING_FROM_LOC_AT_STAGING_State( ES_Event Event)
  {
    ES_Event 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
      // after that start any lower level machines that run in this state
      //StartLowerLevelSM( Event );
      //printf("Enter RECEIVING_FROM_LOC_AT_STAGING State\r\n");
      // 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
      //printf("Exit RECEIVING_FROM_LOC_AT_STAGING State\r\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
```

```c

      // 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 DuringWait200msState( ES_Event Event)
  {
    ES_Event 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) )
    {  printf("Enter WAITING for 200ms time out state\n\r");
      // implement any entry actions required for this state machine
      // after that start any lower level machines that run in this state
      //StartLowerLevelSM( Event );
      //printf("Enter RECEIVING_FROM_LOC_AT_STAGING State\r\n");
      // repeat the StartxxxSM() functions for concurrent state machines
      // on the lower level
    }
    else if ( Event.EventType == ES_EXIT )
    {
      printf("Exiting wait200ms state\n\r");
      // 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
      //printf("Exit RECEIVING_FROM_LOC_AT_STAGING State\r\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);
  }
  /*******************************************SPI FUNCTIONS*******************************************/

  void SPI_Init(void){
    // Enable the clock to the GPIO Port (we are going to use Port A)
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    // Enable clock to SSI - set to SSI Module 0
    HWREG(SYSCTL_RCGCSSI) |= SYSCTL_RCGCSSI_R0;

    // Wait for GPIO Port to be ready by killing a few cycles
    while((HWREG(SYSCTL_RCGCGPIO) & SYSCTL_RCGCGPIO_R0) != SYSCTL_RCGCGPIO_R0){}
    //printf("right supanath 123456789 BANANA abc before while loop\n\r");
    // Program the GPIO to use the alternate functions on the SSI pins PA2,3,4,5
    //  HWREG(GPIO_PORTA_BASE + GPIO_O_AFSEL) = (HWREG(GPIO_PORTA_BASE + GPIO_O_AFSEL)
    //    & 0xffffff0) | (BIT2HI | BIT3HI| BIT4HI| BIT5HI);
    HWREG(GPIO_PORTA_BASE + GPIO_O_AFSEL) |= (BIT2HI | BIT3HI| BIT4HI| BIT5HI);
    // Set Mux position in GPIOPCTL to select the SSI use of the pins
    HWREG(GPIO_PORTA_BASE+GPIO_O_PCTL) =
    (HWREG(GPIO_PORTA_BASE+GPIO_O_PCTL) & 0xff0000ff) + (2<<(5*BitsPerNibble))
    + (2<<(4*BitsPerNibble)) + (2<<(3*BitsPerNibble)) + (2<<(2*BitsPerNibble));
    // Program the port lines for digital I/O
    HWREG(GPIO_PORTA_BASE+GPIO_O_DEN) |= (BIT2HI | BIT3HI | BIT4HI | BIT5HI);
    // Program the required data directions on the port line
    HWREG(GPIO_PORTA_BASE+GPIO_O_DIR) &= BIT4LO; //PA4 is the input (receiver line)
    HWREG(GPIO_PORTA_BASE+GPIO_O_DIR) |= (BIT2HI | BIT3HI | BIT5HI);
    // If using SPI mode 3, program the pull-up on the clock line
    //HWREG(GPIO_PORTD_BASE + GPIO_O_PUR) |= BIT0HI;
    HWREG(GPIO_PORTA_BASE + GPIO_O_PUR) |= BIT2HI;
```

```c
    // Wait for the SSI0 to be ready
    while((HWREG(SYSCTL_RCGCSSI) & SYSCTL_RCGCSSI_R0) != SYSCTL_RCGCSSI_R0){}
    // Make sure that the SSI is disabled before programming mode bits
    HWREG(SSI0_BASE + SSI_O_CR1) = HWREG(SSI0_BASE + SSI_O_CR1) & ~SSI_CR1_SSE;
    // Select Master mode and TXRES indicating EOT
    HWREG(SSI0_BASE + SSI_O_CR1) &= ~SSI_CR1_MS; //Set to 0 for master
    HWREG(SSI0_BASE + SSI_O_CR1) |= SSI_CR1_EOT; //Set EOT to 1 (for interupt)
    // Configure the SSI clock source to the system clock
    HWREG(SSI0_BASE + SSI_O_CC) = (HWREG(SSI0_BASE + SSI_O_CC) & ~SSI_CC_CS_M) | SSI_CC_CS_SYSPLL;
    // Configure the clock pre-scaler: here we want CPSDVSR = , 1+SCR =
    HWREG(SSI0_BASE + SSI_O_CPSR) = (HWREG(SSI0_BASE + SSI_O_CPSR) &
    ~SSI_CPSR_CPSDVSR_M) | CPSDVSR_PRESCALER; //set CPSDVSR = 80
    // Configure clock rate (SCR) - 0, phase (SPH)- 1 and
    // polarity (SPO)- 1 ,mode (FRF) - freescale(0) and datasize (DSS) - 8 bit
    HWREG(SSI0_BASE + SSI_O_CR0) = (HWREG(SSI0_BASE + SSI_O_CR0) & 0xffff0000)
    | ((SCR << 8)| SSI_CR0_SPH | SSI_CR0_SPO | SSI_CR0_DSS_8 | SSI_CR0_FRF_MOTO);
    // Locally Enable Interrupts (TXIM in SSIIM)
    enable_SPI_Interupt();
    // Enable SSI
    HWREG(SSI0_BASE + SSI_O_CR1) = (HWREG(SSI0_BASE + SSI_O_CR1) & ~SSI_CR1_SSE) | SSI_CR1_SSE;
    // Globally enable interupts
    __enable_irq();
    // Enable the NVIC interrupt for the SSI when starting to transmit
    // enable SSI3 interrupt in the NVIC, it is interrupt number 7 so appears in EN0 at bit 7
    HWREG(NVIC_EN0) = BIT7HI;

    //make sure we disable loopback mode
    HWREG(SSI0_BASE + SSI_O_CR1) &= (~SSI_CR1_LBM); //DISABLE LOOP BACK MODE FOR SURE
    printf("finsihed SPI init\n\r");
  }
  // We enable or disable interupt by setting or clearing TXIM
  static void enable_SPI_Interupt(void){
    HWREG(SSI0_BASE+SSI_O_IM) |= SSI_IM_TXIM;
  }

  static void disable_SPI_Interupt(void){
    HWREG(SSI0_BASE+SSI_O_IM) &= ~SSI_IM_TXIM;
  }

  void SPI_Interupt_Response(void){\

    disable_SPI_Interupt(); //disable the interupt
    //we have more bytes now, each read is 8-bit by initialization

    //consecutive read try
    data1=SPI_Read();
    //printf("Passed data1\n\r");
    data2=SPI_Read();
    data3=SPI_Read();
    data4=SPI_Read();
    data5=SPI_Read();
    ES_Event new_event; //finish 5 bytes sequence, post EOT of 5 bytes
    new_event.EventType = ES_EOT;
    PostLOCMasterSM(new_event);


  }

  // Set the data to SPI register
  void SPI_Write(uint8_t data){
    HWREG(SSI0_BASE+SSI_O_DR) = data;
  }

  // Read the data from SPI register
  uint8_t SPI_Read(void){
    uint8_t dataHolder;
    dataHolder = HWREG(SSI0_BASE+SSI_O_DR);
    return dataHolder;
  }
  /******************************Other functions***************/
  uint32_t QueryGameStatus(void){

```

```c
  return (StatusBytes);
}

uint8_t QueryActiveLocation(void){

  return (ActiveLocation);
}



/***********************************Input Capture related to Hall Sensor*****************/
void InitInputCapture_Hall( void ){
  // start by enabling the clock to the timer (Wide Timer 5)
  HWREG(SYSCTL_RCGCWTIMER) |= SYSCTL_RCGCWTIMER_R5;
  // enable the clock to Port D
  HWREG(SYSCTL_RCGCGPIO) |= SYSCTL_RCGCGPIO_R3;
  // since we added this Port D clock init, we can immediately start
  // into configuring the timer, no need for further delay
  // make sure that timer (Timer A) is disabled before configuring
  HWREG(WTIMER5_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEN;
  // set it up in 32bit wide (individual, not concatenated) mode
  // the constant name derives from the 16/32 bit timer, but this is a 32/64
  // bit timer so we are setting the 32bit mode
  HWREG(WTIMER5_BASE+TIMER_O_CFG) = TIMER_CFG_16_BIT;
  // we want to use the full 32 bit count, so initialize the Interval Load
  // register to 0xffff.ffff (its default value :-)
  HWREG(WTIMER5_BASE+TIMER_O_TAILR) = 0xffffffff;
  // set up timer A in capture mode (TAMR=3, TAAMS = 0),
  // for edge time (TACMR = 1) and up-counting (TACDIR = 1)
  HWREG(WTIMER5_BASE+TIMER_O_TAMR) =
  (HWREG(WTIMER5_BASE+TIMER_O_TAMR) & ~TIMER_TAMR_TAAMS) |
  (TIMER_TAMR_TACDIR | TIMER_TAMR_TACMR | TIMER_TAMR_TAMR_CAP);
  // To set the event to rising edge, we need to modify the TAEVENT bits
  // in GPTMCTL. Rising edge = 00, so we clear the TAEVENT bits
  HWREG(WTIMER5_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEVENT_M;
  // Now Set up the port to do the capture (clock was enabled earlier)
  // start by setting the alternate function for Port D bit 6 (WT5CCP0)
  HWREG(GPIO_PORTD_BASE+GPIO_O_AFSEL) |= BIT6HI;
  // Then, map bit 6's alternate function to WT5CCP0
  // 7 is the mux value to select WT0CCP0, 16 to shift it over to the
  // right nibble for bit 6 (4 bits/nibble * 6 bits)
  HWREG(GPIO_PORTD_BASE+GPIO_O_PCTL) =
  (HWREG(GPIO_PORTD_BASE+GPIO_O_PCTL) & 0xf0ffffff) + (7<<24);
  // Enable pin on Port D for digital I/O
  HWREG(GPIO_PORTD_BASE+GPIO_O_DEN) |= BIT6HI;
  // make pin 4 on Port D into an input
  HWREG(GPIO_PORTD_BASE+GPIO_O_DIR) &= BIT6LO;
  // back to the timer to enable a local capture interrupt
  HWREG(WTIMER5_BASE+TIMER_O_IMR) |= TIMER_IMR_CAEIM;
  // enable the Timer A in Wide Timer 0 interrupt in the NVIC
  // it is interrupt number 104 so appears in EN3 at bit 8
  HWREG(NVIC_EN3) |= BIT8HI;
  // make sure interrupts are enabled globally
  __enable_irq();
  // now kick the timer off by enabling it and enabling the timer to
  // stall while stopped by the debugger
  HWREG(WTIMER5_BASE+TIMER_O_CTL) |= (TIMER_CTL_TAEN | TIMER_CTL_TASTALL);
}



void InputCaptureResponse_Hall( void ){
  ES_Event EventToPost_InputCapture;

  // start by clearing the source of the interrupt, the input capture event
  HWREG(WTIMER5_BASE+TIMER_O_ICR) = TIMER_ICR_CAECINT;
  //printf("In input capture ISR after clearing\n\r");

  // now grab the captured value and calculate the period
  ThisCapture = HWREG(WTIMER5_BASE+TIMER_O_TAR);
  ThisPeriod = ThisCapture - LastCapture;
  // update LastCapture to prepare for the next edge
```

```
  LastCapture = ThisCapture;
  //update the FlagValidFreq and see if we get a code

  measFreqCode=frequency_map(ThisPeriod);
  //see if the current measuremented changed
  if (measFreqCode==prevMeasFreqCode){
    stableCounter=stableCounter+1; //if so, increment
  }
  else{
    stableCounter=0; //restart the count
  }

  //if we have read enough times of the same measurement, we know it's stable
  if (stableCounter==numOfMeasurementsForStable){
    stableMeasFreqCode=measFreqCode;
    stableCounter=0;//restart the count

    //only post frequency when we have a valid and stable frequency code
    if ((stableMeasFreqCode!=0xf0)&&(FlagFreqConsumed==1)){
      if (CurrentState==WAITING){
        EventToPost_InputCapture.EventType=ARRIVED_AT_STAGING;

      }

      else if ((CurrentState==SENDING_TO_LOC_AT_STAGING)||(CurrentState==RECEIVING_FROM_LOC_AT_STAGING)){
        //ecide whether that's a code for 1st report or 2nd report
        if (FlagSentOneReport==0){
          FlagFreqConsumed=0;
          EventToPost_InputCapture.EventType=SEND_FIRST_REPORT;
          EventToPost_InputCapture.EventParam=stableMeasFreqCode;
          PostLOCMasterSM( EventToPost_InputCapture);
        }
        else if ((FlagSentTwoReport==0)&&(FlagFirstReportACK==1)){
          FlagFreqConsumed=0;
          EventToPost_InputCapture.EventType=SEND_SECOND_REPORT;
          EventToPost_InputCapture.EventParam=stableMeasFreqCode;
          PostLOCMasterSM( EventToPost_InputCapture);
        }
      }
    }
  }
  //don't forget the update prevMeasFreqCode
  prevMeasFreqCode=measFreqCode;

}//end input capture response



uint8_t frequency_map(uint32_t period){
  //input is in encoder ticks
  uint8_t result = 0xf0;//assume we have no valid frequency to start with
  //the input would be in ticks, 4*10^7 ticks in one sec, period table is in micro seconds,
  //4*10^7 ticks in one sec is 4*10^7 ticks in 10^6 micro
  period=period/TicksToMicroSecDivisor;

  for (int i = 0; i < period_table_length; i++){
    if ((period > PERIOD_TABLE[i]-FreqErrorThreshold) &
    (period < PERIOD_TABLE[i] + FreqErrorThreshold)){
      result = (uint8_t)i; //the code corresponds to the index
      break;
    }
  }

  return result;
}

//--------------------functions used to interact with other modules-------------------
uint32_t queryStatusBytes(void){
  return StatusBytes;
}

uint8_t queryActiveStagingGreen(void){
```

```
  //returns 0 for no active staging (either game hasn't started or in shooting), 1, 2, 3 for 1R,2R,3R
  //5 means error
  uint8_t ReturnResult=0;
  if ((StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)==0){
    ReturnResult=0; //the game hasn't started, you can't do anything
  }
  else if ((StatusBytes & GAME_STATUS_CSG_MASK)>0){//it's shooting, should not be staging
    ReturnResult=INVALID_LOCATIONS_ACTIVE;
  }
  else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_NONE_MASK){
    ReturnResult=0;
  }
  else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_ONE_MASK){
    ReturnResult=1;
  }
  else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_TWO_MASK){
    ReturnResult=2;
  }
  else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_THREE_MASK){
    ReturnResult=3;
  }
  else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)>=GAME_STATUS_GREEN_LOC_ALL_MASK){
    ReturnResult=INVALID_LOCATIONS_ACTIVE;//you cannot have all staging active for check-ins
  }
  return ReturnResult;

}

uint8_t queryActiveStagingRed(void){
  //returns 0 for no active staging (either game hasn't started or in shooting), 1, 2, 3 for 1R,2R,3R
  uint8_t ReturnResult=0;
  if ((StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)==0){
    ReturnResult=0; //the game hasn't started, you can't do anything
  }
  else if ((StatusBytes & GAME_STATUS_CSR_MASK)>0){//it's shooting, should not be staging
    ReturnResult=INVALID_LOCATIONS_ACTIVE;
  }
  else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_NONE_MASK){
    ReturnResult=0;
  }
  else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_ONE_MASK){
    ReturnResult=1;
  }
  else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_TWO_MASK){
    ReturnResult=2;
  }
  else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_THREE_MASK){
    ReturnResult=3;
  }
  else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)>=GAME_STATUS_RED_LOC_ALL_MASK){
    ReturnResult=INVALID_LOCATIONS_ACTIVE;
  }

  return ReturnResult;

}

uint8_t queryActiveShootingGreen(void){
  //returns 0 for no active shooting (either game hasn't started or still checking in), 1, 2, 3 for 1R,2R,3R
  //4 means all goals are active, 5 means error
  uint8_t ReturnResult=0;
  if ((StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)==0){
    ReturnResult=0; //the game hasn't started, you can't do anything
  }
  else if ((StatusBytes & GAME_STATUS_CSG_MASK)==0){//it's checking in, should not be shooting
    ReturnResult=INVALID_LOCATIONS_ACTIVE;
  }
  else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_NONE_MASK){
    ReturnResult=0;
  }
  else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_ONE_MASK){
```

```
      ReturnResult=1;
    }
    else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_TWO_MASK){
      ReturnResult=2;
    }
    else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_THREE_MASK){
      ReturnResult=3;
    }
    else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)>=GAME_STATUS_GREEN_LOC_ALL_MASK){
      printf("Last 18 seconds!!, return shooting stage 4\r\n");
      ReturnResult=4;//you cannot have all staging active for check-ins
    }
    return ReturnResult;

  }

  uint8_t queryActiveShootingRed(void){
    //returns 0 for no active shooting (either game hasn't started or still checking in), 1, 2, 3 for 1R,2R,3R
    //4 means all goals are active, 5 means error
    uint8_t ReturnResult=0;
    if ((StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)==0){
      ReturnResult=0; //the game hasn't started, you can't do anything
    }
    else if ((StatusBytes & GAME_STATUS_CSR_MASK)==0){//it's checking in, should not be shooting
      ReturnResult=INVALID_LOCATIONS_ACTIVE;
    }
    else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_NONE_MASK){
      ReturnResult=0;
    }
    else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_ONE_MASK){
      ReturnResult=1;
    }
    else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_TWO_MASK){
      ReturnResult=2;
    }
    else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_THREE_MASK){
      ReturnResult=3;
    }
    else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)>=GAME_STATUS_RED_LOC_ALL_MASK){
      ReturnResult=4;//you cannot have all staging active for check-ins
    }
    return ReturnResult;

  }
  uint8_t quickQueryActiveLocation(void){
    //instead of waiting for the GAME_STATUS to update,just grab it from the response
    return ActiveLocation;
  }

  uint8_t queryGoalGreen(void){
    return (StatusBytes & GOAL_SCORE_GREEN_MASK) >> GOAL_SCORE_GREEN_OFFSET;
  }

  uint8_t queryGoalRed(void){
    return StatusBytes & GOAL_SCORE_RED_MASK;
  }


  void check_active_event(){
    if(get_Team() == GREEN){
      // Check Staging
      printf("Enter check active green\r\n");
      ActiveGreenStagingLocation=queryActiveStagingGreen();
      //printf("Before the posting if, ActiveGreenStagingLocation is %d\n\r",ActiveGreenStagingLocation);
      if ((ActiveGreenStagingLocation!=0)&&(ActiveGreenStagingLocation!=INVALID_LOCATIONS_ACTIVE)){
        //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE
        //it's a valid location at a valid, and it's different from before
        EventToPost.EventType=STAGE_ACTIVE;
        EventToPost.EventParam=ActiveGreenStagingLocation;
        PostMasterVehicleSM(EventToPost);
        printf("\r\nActive Green Staging Location is: %d",ActiveGreenStagingLocation);
```

```
      }
      prevActiveGreenStagingLocation=ActiveGreenStagingLocation;

      // Check shooting
      ActiveGreenShootingLocation=queryActiveShootingGreen();
      //printf("Before the posting if, ActiveGreenShootingLocation is %d\n\r",ActiveGreenShootingLocation);
      if ((ActiveGreenShootingLocation!=0)&&(ActiveGreenShootingLocation!=INVALID_LOCATIONS_ACTIVE)){
        //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE
        //it's a valid location at a valid, and it's different from before
        EventToPost.EventType=SHOOT_ACTIVE;
        EventToPost.EventParam=ActiveGreenShootingLocation;
        PostMasterVehicleSM(EventToPost);
        printf("\r\nActive Green Shooting Location is: %d",ActiveGreenShootingLocation);
      }
      prevActiveGreenShootingLocation=ActiveGreenShootingLocation;
    }
    //#endif

    //#ifdef TEAM_RED
    if(get_Team() == RED){
      // Check staging
      printf("Enter check active red\r\n");
      ActiveRedLocation=queryActiveStagingRed();
      if ((ActiveRedLocation!=0)&&(ActiveRedLocation!=INVALID_LOCATIONS_ACTIVE)&&(ActiveRedLocation!=prevActiveRedLocation)){
        //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE
        //it's a valid location at a valid, and it's different from before
        EventToPost.EventType=STAGE_ACTIVE;
        EventToPost.EventParam=ActiveRedLocation;
        PostMasterVehicleSM(EventToPost);
        printf("\r\nActive Red stagingLocation is: %d",ActiveRedLocation);

      }
      prevActiveRedLocation=ActiveRedLocation;

      // Check shooting
      ActiveRedLocation=queryActiveShootingRed();
      if ((ActiveRedLocation!=0)&&(ActiveRedLocation!=INVALID_LOCATIONS_ACTIVE)&&(ActiveRedLocation!=prevActiveRedLocation)){
        //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE
        //it's a valid location at a valid, and it's different from before
        EventToPost.EventType=SHOOT_ACTIVE;
        EventToPost.EventParam=ActiveRedLocation;
        PostMasterVehicleSM(EventToPost);
        printf("\r\nActive Red Shooting Location is: %d",ActiveRedLocation);

      }
    }
  }
```