```
/****************************************************************************
 Module
   StageSM.c

 Revision
   2.0.1

 Description
   Handles staging.
   This is based on a template file for implementing state machines.

****************************************************************************/
/*----------------------------- Include Files -----------------------------*/
// Basic includes for a program using the Events and Services Framework
#include "ES_Configure.h"
#include "ES_Framework.h"

/* 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 "StageSM.h"
#include "Location.h"
#include "MasterVehicle.h"
#include "LOCMaster.h"
/*----------------------------- Module Defines ----------------------------*/

// define constants for the states for this machine
// and any other local defines
/* Team Definition */
#define RED 0
#define GREEN 1

#define ENTRY_STATE STAGE_WAITING
#define ONE_SEC 976
#define HALF_SEC (ONE_SEC /2)
#define FREQ_TIME HALF_SEC
#define THREE_SEC 3000
#define TWO_SEC 2000
#define STAGE_WAITING_TIME TWO_SEC
#define COMPETITION_FULL_DUTY 75
#define CORRECTION_FULL_DUTY 55
/*---------------------------- Module Functions ---------------------------*/
/* prototypes for private functions for this machine, things like during
   functions, entry & exit functions.They should be functions relevant to the
   behavior of this state machine
*/
static ES_Event DuringWaiting( ES_Event Event);
static ES_Event DuringMoveY( ES_Event Event);
static ES_Event DuringMoveX( ES_Event Event);
static ES_Event DuringFreqMeasurement( ES_Event Event);
static ES_Event DuringWaitingForResponse( ES_Event Event);
/*---------------------------- Module Variables ---------------------------*/
// everybody needs a state variable, you may need others as well
static StagingState_t CurrentState;
static uint32_t CurrentXDestination=0;
static uint32_t CurrentYDestination=0;
/*------------------------------ Module Code ------------------------------*/
/****************************************************************************
 Function
    RunStageSM

 Parameters
   ES_Event: the event to process

 Returns
   ES_Event: an event to return

****************************************************************************/
ES_Event RunStageSM( ES_Event CurrentEvent )
{
   bool MakeTransition = false;/* are we making a state transition? */
   StagingState_t NextState = CurrentState;
```

```
    ES_Event EntryEventKind = { ES_ENTRY, 0 };// default to normal entry to new state
    ES_Event ReturnEvent = CurrentEvent; // assume we are not consuming event

    switch ( CurrentState )
    {
       case STAGE_WAITING :       // If current state is waiting state
          // Execute During function for state one. ES_ENTRY & ES_EXIT are
          // processed here allow the lower level state machines to re-map
          // or consume the event
          CurrentEvent = DuringWaiting(CurrentEvent);
          //process any events
          if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
          {
             switch (CurrentEvent.EventType)
             {
                case STAGE_ACTIVE : //If event is STAGE_ACTIVE
             uint8_t dest_index = 0;
                   // check team selection
                   // if team is GREEN
             if(get_Team() == GREEN){
                      // get stage destination index
            dest_index = queryActiveStagingGreen();
                      // if index is between 0 and 5, which mean staging is active
                      // we can get the coordinate x,y for this location
            if ((dest_index!=0)&&(dest_index!=5)){
               CurrentXDestination = get_Stage_Green_X(dest_index);
             CurrentYDestination = get_Stage_Green_Y(dest_index);
              }
             }
                   // else if team is RED
           else{
                      // get stage destination index
            dest_index = queryActiveStagingRed();
                      // if index is between 0 and 5, which mean staging is active
                      // we can get the coordinate x,y for this location
            if ((dest_index!=0)&&(dest_index!=5)){
               CurrentXDestination = get_Stage_Red_X(dest_index);
               CurrentYDestination = get_Stage_Red_Y(dest_index);
            }
           }

                   // Execute action function for state one : event one
                   NextState = STAGE_MOVE_Y;//Decide what the next state will be
                   // for internal transitions, skip changing MakeTransition
                   MakeTransition = true; //mark that we are taking a transition
                   // if transitioning to a state with history change kind of entry
                   // EntryEventKind.EventType = ES_ENTRY_HISTORY;
                   // optionally, consume or re-map this event for the upper
                   // level state machine
                   ReturnEvent.EventType = ES_NO_EVENT;
                   break;

                default:
             break;
             }
          }
          break;

        case STAGE_MOVE_Y :       // If current state is STAGE_MOVE_Y
          // Execute During function for state one. ES_ENTRY & ES_EXIT are
          // processed here allow the lower level state machines to re-map
          // or consume the event
          CurrentEvent = DuringMoveY(CurrentEvent);
          //process any events
          if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
          {
             switch (CurrentEvent.EventType)
             {

          case SHOOT_ACTIVE_4: // if event is SHOOT_ACTIVE_4 which is shooting in last 18 sec
                   NextState = STAGE_WAITING; // set next state to waiting
                   MakeTransition = true; //mark that we are taking a transition
```

```
                    ReturnEvent.EventType = SHOOT_ACTIVE_4; //return this event to upper SM
                    break;

                 case Y_REACHED : //If event is reached target y location
                    // Execute action function for state one : event one
                    NextState = STAGE_MOVE_X; // set next state to moving in x
                    // for internal transitions, skip changing MakeTransition
                    MakeTransition = true; //mark that we are taking a transition
                    // if transitioning to a state with history change kind of entry
                    // EntryEventKind.EventType = ES_ENTRY_HISTORY;
                    // optionally, consume or re-map this event for the upper
                    // level state machine
                    ReturnEvent.EventType = ES_NO_EVENT;
                    break;
            case CONSTRUCTION_END: //If event is construction end, we want to exit this SM
                    NextState = STAGE_WAITING;//Decide what the next state will be
                    MakeTransition = true; //mark that we are taking a transition
                    ReturnEvent.EventType = CONSTRUCTION_END; // return this event to upper SM
                    break;

                 default:
             break;
                }
            }
            break;

         case STAGE_MOVE_X :       // If current state is STAGE_MOVE_X
            // Execute During function for state one. ES_ENTRY & ES_EXIT are
            // processed here allow the lower level s tate machines to re-map
            // or consume the event
            CurrentEvent = DuringMoveX(CurrentEvent);
            //process any events
            if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
            {
               switch (CurrentEvent.EventType)
               {
            case SHOOT_ACTIVE_4: // if event is SHOOT_ACTIVE_4 which is shooting in last 18 sec
                    NextState = STAGE_WAITING; //set next state to waiting
                    MakeTransition = true; //mark that we are taking a transition
                    ReturnEvent.EventType = SHOOT_ACTIVE_4; // return this event to upper SM
                    break;

                 case X_REACHED: // If event is reaching target x location
                    // we reverify that the current y location is still our target y location
             if(verify_y_location()){
                        // if y location is correct, reset PWM to high speed
                set_PWM_Full_Duty(COMPETITION_FULL_DUTY);
                        // set next state to STAGE_VERIFICATION
                NextState = STAGE_VERIFICATION;
             }
                    // if our y location is incorrect
             else{
                        // set PWM speed to lower correction speed (to avoid overshoot)
              set_PWM_Full_Duty(CORRECTION_FULL_DUTY);
              printf("Y location invalid, call move y again\r\n");
                        // set next state to STAGE_MOVE_Y
              NextState = STAGE_MOVE_Y;
             }
             MakeTransition = true;
               ReturnEvent.EventType = ES_NO_EVENT;
                    break;

                 case CONSTRUCTION_END: //If event is CONSTRUCTION_END
                    NextState = STAGE_WAITING;//Decide what the next state will be
                    MakeTransition = true; //mark that we are taking a transition
                    ReturnEvent.EventType = CONSTRUCTION_END; // return this event to upper SM
                    break;

            default:
             break;
               }
            }
```

```
            break;

         case STAGE_VERIFICATION :       // If current state is STAGE_VERIFICATION
            // Execute During function for state one. ES_ENTRY & ES_EXIT are
            // processed here allow the lower level state machines to re-map
            // or consume the event
            CurrentEvent = DuringFreqMeasurement(CurrentEvent);
            //process any events
            if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
            {
               switch (CurrentEvent.EventType)
               {
            case ES_TIMEOUT:
                     // if we get ES_TIMEOUT from the stage_timer,
                     // we should exit stage SM and enter stage SM again
             if(CurrentEvent.EventParam == STAGE_TIMER){
                NextState = STAGE_WAITING;
              MakeTransition = true; //mark that we are taking a transition
              printf("STAGE SM Timeout, go back to IDLE and restart stage SM\r\n");
              ReturnEvent.EventType = ES_TIMEOUT; //return this event to upper SM
                        // also post RESTART_VERIFY_FREQ to LOCMaster SM
              ES_Event to_post;
              printf("Post to LOCmaster to restart verifying stage\r\n");
              to_post.EventType = RESTART_VERIFY_FREQ;
              PostLOCMasterSM(to_post);
              break;
               }

                  case FINISHED_STAGING: //If event is FINISHED_STAGING
               // Execute action function for state one : event one
             NextState = STAGE_WAITING; // set next state to STAGE_WAITING
             MakeTransition = true; //mark that we are taking a transition
             printf("STAGE SM received FINISHED STAGING - returning to STAGE WAITING\r\n");
             ReturnEvent.EventType = FINISHED_STAGING; // return this event to upper SM
                     break;

            case CONSTRUCTION_END: //If event is CONSTRUCTION_END
                     NextState = STAGE_WAITING; // set next state to waiting
                     MakeTransition = true; //mark that we are taking a transition
                     ReturnEvent.EventType = CONSTRUCTION_END; // return this event to upper SM
                     break;

                  default:
             break;
               }
            }
            break;

    default:
       break;
      }
      // If we are making a state transition
      if (MakeTransition == true)
      {
         // Execute exit function for current state
         CurrentEvent.EventType = ES_EXIT;
         RunStageSM(CurrentEvent);

         CurrentState = NextState; //Modify state variable

         // Execute entry function for new state
         // this defaults to ES_ENTRY
         RunStageSM(EntryEventKind);
       }
       return(ReturnEvent);
  }
  /****************************************************************************
   Function
       StartStageSM

   Parameters
       None
```

```

 Returns
     None

 Description
     Does any required initialization for this state machine

****************************************************************************/
void StartStageSM ( ES_Event CurrentEvent )
{
   // to implement entry to a history state or directly to a substate
   // you can modify the initialization of the CurrentState variable
   // otherwise just start in the entry state every time the state machine
   // is started
   if ( ES_ENTRY_HISTORY != CurrentEvent.EventType )
   {
        CurrentState = ENTRY_STATE;
   }
   // call the entry function (if any) for the ENTRY_STATE
   RunStageSM(CurrentEvent);
}

/****************************************************************************
 Function
     QueryStageSM

 Parameters
     None

 Returns
     StageState_t The current state of the Stage state machine

****************************************************************************/
StagingState_t QueryStageSM ( void )
{
   return(CurrentState);
}

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

static ES_Event DuringWaiting( 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) ){
        // no the lower level
    }
    else if ( Event.EventType == ES_EXIT ){
    }
    else{
    }
    // 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 DuringMoveY( 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
        move_Y(CurrentYDestination);
    }
    else if ( Event.EventType == ES_EXIT ){
        // make sure the motor is stopped when exiting this moving state
    stopMotor();
    }
    else{
    }
```

```
    // 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 DuringMoveX( 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
        move_X(CurrentXDestination); // call move function to target x destination
    }
    else if ( Event.EventType == ES_EXIT ){
        // on exit, give the lower levels a chance to clean up first
        // make sure the motor is stopped when exiting this moving state
    stopMotor();
    }
    else{
    }
    // 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 DuringFreqMeasurement( 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
        // when entering this state, post ARRIVE_AT_STAGING event to
        // LOCMaster to start frequency verification process
    ES_Event to_post;
    printf("Post arrive at staging to LOC Master\r\n");
    to_post.EventType = ARRIVED_AT_STAGING;
    PostLOCMasterSM(to_post);
        // start STAGE_TIMER to limit the maximum time it takes
        // to verify frequencies and send two reports
    ES_Timer_InitTimer(STAGE_TIMER, STAGE_WAITING_TIME);
    }
    else if ( Event.EventType == ES_EXIT ){
    }
    else{
    }
    // 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);
}

```