

```

1  /*****
2  Cleaned up by Huajian Huang on 16:36, March 11th, 2017
3  *****/
4
5
6  #include "ES_Configure.h"
7  #include "ES_Framework.h"
8  #include "LOCMaster.h"
9
10 // some includes used in the SPI.c from Lab 8
11 #include "ES_DeferRecall.h"
12 // #include "SPI.h"
13 // #include "BaseControl.h"
14 #include "inc/hw_memmap.h"
15 #include "inc/hw_types.h"
16 #include "inc/hw_gpio.h"
17 #include "inc/hw_sysctl.h"
18 #include "driverlib/sysctl.h"
19 #include "driverlib/pin_map.h" // Define PART_TM4C123GH6PM in project
20 #include "driverlib/gpio.h"
21 // #include "ES_ShortTimer.h"
22 #include "inc/hw_timer.h"
23 #include "inc/hw_ssi.h"
24 #include "inc/hw_nvic.h"
25 #include "OwnPWM.h"
26 #include "MasterVehicle.h"
27 // some testing constants
28 // #define TEST_LOC_STAGE
29
30 /*----- Module Defines -----*/
31 #define GET_STATUS_PERIOD 350 // game status updates at 3Hz, no need to go faster
32
33 #define GET_STATUS_COMMAND 0xc0 // = 0b11000000, 0b xxxx xxxx means binary with 8 bits
34 #define REPORT_FRONT_BITS 0x80 // = 0b1000 0000, we put the frequency report in the second 4 bit
35 // #define GET_STATUS_COMMAND 0xAA // fake command, just for testing
36 // #define GET_RR_RS_COMMAND 0x70 // = 0b01110000
37
38 #define TWO_HUNDRED_MS 300 // should be 200, but I am faking so it's at least 200ms
39 // #define TWO_HUNDRED_MS 2000 // this is a fake time for debugging
40 #define RESPONSE_READY_CODE 0xAA
41 #define RESPONSE_NOT_READY_CODE 0x00
42 #define ACK_CODE 0x00 // NEED TO ADJUST AND ONLY GRAB THE 6,7TH BIT
43 #define NACK_CODE 0xC0
44 #define REPORT_RESPONSE_COMMAND 112 // 0x70 = 0b 0111 0000
45 #define GAME_STATUS_SB3_GS_EXTRACT_MASK 0x00000080
46 #define GAME_STATUS_CSG_MASK 0x00800000
47 #define GAME_STATUS_CSR_MASK 0x00080000
48 #define GAME_STATUS_GREEN_ACTIVE_STATUS_MASK 0x00700000
49 #define GAME_STATUS_RED_ACTIVE_STATUS_MASK 0x00070000
50 #define GAME_STATUS_GREEN_LOC_NONE_MASK 0x00000000
51 #define GAME_STATUS_GREEN_LOC_ONE_MASK 0x00100000
52 #define GAME_STATUS_GREEN_LOC_TWO_MASK 0x00200000
53 #define GAME_STATUS_GREEN_LOC_THREE_MASK 0x00300000
54 #define GAME_STATUS_GREEN_LOC_ALL_MASK 0x00400000
55 #define GAME_STATUS_RED_LOC_NONE_MASK 0x00000000
56 #define GAME_STATUS_RED_LOC_ONE_MASK 0x00010000
57 #define GAME_STATUS_RED_LOC_TWO_MASK 0x00020000
58 #define GAME_STATUS_RED_LOC_THREE_MASK 0x00030000
59 #define GAME_STATUS_RED_LOC_ALL_MASK 0x00040000
60 #define GOAL_SCORE_GREEN_MASK 0x00003f00
61 #define GOAL_SCORE_RED_MASK 0x0000003f
62 #define GOAL_SCORE_GREEN_OFFSET 8
63 // these times assume a 1.000mS/tick timing
64 #define TEN_MSEC 10
65 #define ONE_SEC 976
66 #define HALF_SEC (ONE_SEC / 2)
67 #define TWO_SEC (ONE_SEC * 2)
68 #define FIVE_SEC (ONE_SEC * 5)
69 #define BitsPerNibble 4
70 #define CPSDVSR_PRESCALER 128
71 #define SCR 60 // date rate should be 1/60 of the Lab 8's data rate, due to Tcy
72

```

```

73 #define RED 0
74 #define GREEN 1
75 #define INVALID_LOCATIONS_ACTIVE 5
76 //input capture realted
77 #define CHECK_ACTIVE_ALWAYS
78 //the input would be in ticks, 4*10^7 ticks in one sec, period table is in micro seconds,
79 //4*10^7 ticks in one sec is 4*10^7 ticks in 10^6 micro
80 #define TicksToMicroSecDivisor 40
81 #define FreqErrorThreshold 25
82
83 // #define TEAM_GREEN
84 // #define TEAM_RED
85
86 /*----- Module Functions -----*/
87 static ES_Event DuringStateOne( ES_Event Event);
88 static ES_Event DuringWaitingState( ES_Event Event);
89 static void enable_SPI_Interrupt(void);
90 static ES_Event During_GAME_STATUS_SENDING_TO_LOC_State( ES_Event Event);
91 static ES_Event During_GAME_STATUS_RECEIVING_FROM_LOC_State( ES_Event Event);
92 static ES_Event During_SENDING_TO_LOC_AT_STAGING_State( ES_Event Event);
93 static ES_Event During_RECEIVING_FROM_LOC_AT_STAGING_State( ES_Event Event);
94 static ES_Event DuringWait200msState(ES_Event CurrentEvent);
95 /*----- Module Variables -----*/
96 // everybody needs a state variable, though if the top level state machine
97 // is just a single state container for orthogonal regions, you could get
98 // away without it
99 static LOCMasterState_t CurrentState;
100 // with the introduction of Gen2, we need a module level Priority var as well
101 static uint8_t MyPriority;
102
103 //our command and response have 4 bytes in them
104 static uint8_t data1;
105 static uint8_t data2;
106 static uint8_t data3;
107 static uint8_t data4;
108 static uint8_t data5;
109 static uint8_t dataBuffer;
110
111 //storing the status bytes, so we can detect a change
112 static uint32_t StatusBytes = 0;
113 static uint32_t prevStatusBytes=0;
114 //active shooting or staging location
115 static uint8_t ActiveLocation=0;
116 //index used to access the list of frequency to have
117 static uint8_t frq_index = 0;
118
119 //serveral flags indicating which step of the handshake we are doing
120 static uint8_t FlagSentOneReport=0;
121 static uint8_t FlagSentTwoReport=0;
122 static uint8_t FlagSentReportWithin200ms=0;
123 static uint8_t FlagFirstReportACK=0;
124 static uint8_t FlagSecondReportACK=0;
125
126 //some input capture variables for the hall sensor, make sure to limit their scope
127 static uint32_t ThisCapture=0;
128 static uint32_t LastCapture=0;
129 static uint32_t ThisPeriod=0;
130 static uint32_t FlagOneShotTimeout=0;
131
132 //define the frequency related constants and variable
133 //the PERIOD_TABLE is in micro seconds
134 static uint32_t const
PERIOD_TABLE[16]={1333,1277,1222,1166,1111,1055,1000,944,889,833,778,722,667,611,556,500};
135
136 static uint8_t
FreqCode[16]={0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F};
137 static uint8_t period_table_length=16;
138
139 static uint8_t FlagValidFreq=0; //raised to 1 when we detect a frequency in the specified table
140 static uint8_t FlagFreqConsumed=1; //init to "consumed", no freq needs to be processed, when it's
ACKed, it's consumed
141 static uint8_t measFreqCode=0xf0; //0xf0 means there's no valid frequency

```

```

142 static uint8_t prevMeasFreqCode = 0xf0;
143 static uint8_t stableMeasFreqCode=0xf0;
144 static uint8_t numOfMeasurementsForStable=10;
145 static uint8_t stableCounter=0;
146
147 //define some variables to store the information that would be used/queried by MasterVehicle
148 static uint8_t ActiveGreenStagingLocation=0;
149 static uint8_t prevActiveGreenStagingLocation=0;
150 static uint8_t ActiveGreenShootingLocation=0;
151 static uint8_t prevActiveGreenShootingLocation=0;
152 static uint8_t ActiveRedLocation=0;
153 static uint8_t prevActiveRedLocation=0;
154
155 static uint8_t prev_green_score = 0;
156 static uint8_t prev_red_score = 0;
157 static uint8_t cur_green_score = 0;
158 static uint8_t cur_red_score = 0;
159 //initialize a event to carry information around
160 static ES_Event EventToPost;
161 /*----- Module Code -----*/
162 /*****
163
164 *****/
165 bool InitLOCMasterSM ( uint8_t Priority )
166 {
167     TERMIO_Init();
168     ES_Event ThisEvent;
169
170     MyPriority = Priority; // save our priority
171
172     ThisEvent.EventType = ES_ENTRY;
173     //ThisEvent.EventType = ARRIVED_AT_STAGING;//I cannot print to my Mac, so go directly when testing
with MAC
174     // Start the Master State machine
175     SPI_Init();
176     InitInputCapture_Hall();
177     StartLOCMasterSM( ThisEvent );
178     ES_Timer_InitTimer(GET_STATUS_TIMER,GET_STATUS_PERIOD);
179     return true;
180 }
181
182 /*****
183
184 *****/
185 bool PostLOCMasterSM( ES_Event ThisEvent )
186 {
187     return ES_PostToService( MyPriority, ThisEvent);
188 }
189
190 /*****
191
192 *****/
193 ES_Event RunLOCMasterSM( ES_Event CurrentEvent )
194 {
195     bool MakeTransition = false; /* are we making a state transition? */
196     LOCMasterState_t NextState = CurrentState;
197     ES_Event EntryEventKind = { ES_ENTRY, 0 }; // default to normal entry to new state
198     ES_Event ReturnEvent = { ES_NO_EVENT, 0 }; // assume no error
199
200     //before everything, clear the 200ms flag if it happens
201     //regardless of states, I almost want to use oneshot timer for this
202     // if ((CurrentEvent.EventType==ES_TIMEOUT) && (CurrentEvent.EventParam==REPORT_TIMER)){
203     //     FlagSentReportWithin200ms=0;
204     // }
205
206     //5 main states, WAITING, GAME_STATUS_SENDING_TO_LOC, GAME_STATUS_RECEIVING_FROM_LOC,
207     // SENDING_TO_LOC_AT_STAGING, RECEIVING_FROM_LOC_AT_STAGING, WAITING_FOR_200MS_TIMEOUT
208
209     //separates the regular game status query and the special procedures in staging
210
211
212     switch ( CurrentState )

```

```

213 {
214     case WAITING :
215
216         // This state is like a neutral transition state
217         //(1) normally wait for timer and keep querying game status
218         //(2) when asked to do staging area stuff, move to the corresponding states
219
220         //In this state, I want to keep querying the game status regularly
221         CurrentEvent = DuringWaitingState(CurrentEvent);
222         //process any events
223         if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
224         {
225             switch (CurrentEvent.EventType)
226             {
227                 case ES_TIMEOUT: //If event is "The GET_STATUS_TIMER" timeout
228                 if (CurrentEvent.EventParam==GET_STATUS_TIMER){
229                     //printf("Timeout in waiting\r\n");
230                     NextState = GAME_STATUS_SENDING_TO_LOC;//Decide what the next state will be
231                     MakeTransition = true; //mark that we are taking a transition
232                     //Post the same event to self
233                     PostLOCMasterSM(CurrentEvent);
234                     // optionally, consume or re-map this event for the upper
235                     // level state machine
236                     ReturnEvent.EventType = ES_NO_EVENT;
237                 }
238                 break;
239
240
241                 case ARRIVED_AT_STAGING:
242                     //printf("Arrive at staging\r\n");
243                     NextState = SENDING_TO_LOC_AT_STAGING;//Decide what the next state will be
244                     // for internal transitions, skip changing MakeTransition
245                     MakeTransition = true; //mark that we are taking a transition
246                     // optionally, consume or re-map this event for the upper
247                     // level state machine
248                     ReturnEvent.EventType = ES_NO_EVENT;
249                     break;
250
251                 default:
252                     break;
253             }
254         } // end if "No event"
255     else // Current Event is now ES_NO_EVENT. Correction 2/20/17 provided by Prof.Ed
256     {
257         //Probably means that CurrentEvent was consumed by lower level
258         ReturnEvent = CurrentEvent; // in that case update ReturnEvent too.
259     }
260     break;
261
262     case GAME_STATUS_SENDING_TO_LOC :
263         //printf("Enter Game status sending to LOC state\n\r");
264         // Execute During function for state one. ES_ENTRY & ES_EXIT are
265         // processed here allow the lowere level state machines to re-map
266         // or consume the event
267
268         CurrentEvent = During_GAME_STATUS_SENDING_TO_LOC_State(CurrentEvent);
269         //process any events
270         if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
271         {
272             switch (CurrentEvent.EventType)
273             {
274                 case ES_TIMEOUT: //If event is GET_STATUS timer timeout
275                 if(CurrentEvent.EventParam==GET_STATUS_TIMER){
276                     //printf("Enter ES_timeout case for game status sending to loc\r\n");
277                     // Execute action function for state one
278                     //time to query for current game status
279                     //write the command 0xC0 and followed by 4 bytes of 0x00
280                     //Pump them into FIFO buffer, when they are all out, we get EOT interrupt
281                     //printf("Sending GET_STATUS_COMMAND\r\n");
282                     SPI_Write(GET_STATUS_COMMAND);
283                     SPI_Write(0X00);
284                     SPI_Write(0X00);

```

```

285     SPI_Write(0X00);
286     SPI_Write(0X00);
287
288     //enable the EOT interrupt
289     enable_SPI_Interupt();
290     //Decide what the next state will be
291     nextState = GAME_STATUS_RECEIVING_FROM_LOC; //neutralize to WAITING, we might need to do
staging area stuff anytime
292     // for internal transitions, skip changing MakeTransition
293     MakeTransition = true; //mark that we are taking a transition
294     // optionally, consume or re-map this event for the upper
295     // level state machine
296     ReturnEvent.EventType = ES_NO_EVENT;
297 }
298 break;
299 // repeat cases as required for relevant events
300 case ARRIVED_AT_STAGING:
301     ES_Event to_post;
302     printf("Repost arrive at staging to LOC Master\r\n");
303     to_post.EventType = ARRIVED_AT_STAGING;
304     PostLOCMasterSM(to_post);
305     ReturnEvent.EventType = ES_NO_EVENT;
306     break;
307
308
309     default:
310     break;
311 }
312 } //end if "no event"
313 else // Current Event is now ES_NO_EVENT. Correction 2/20/17
314 {
315     //Probably means that CurrentEvent was consumed by lower level
316     ReturnEvent = CurrentEvent; // in that case update ReturnEvent too.
317 }
318 break;
319 // repeat state pattern as required for other states
320
321 case GAME_STATUS_RECEIVING_FROM_LOC :
322     //printf("Enter Game status receiving from LOC state\r\n");
323     // Execute During function for state one. ES_ENTRY & ES_EXIT are
324     // processed here allow the lower level state machines to re-map
325     // or consume the event
326     CurrentEvent = During_GAME_STATUS_RECEIVING_FROM_LOC_State(CurrentEvent);
327     //process any events
328     if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
329     {
330         switch (CurrentEvent.EventType)
331         {
332             case ES_EOT: //If event is end of transmission of 5 bytes, provided by EOT interrupt
333                 // Execute action function for state one
334                 //printf("Received EOT in receiving from loc state\r\n");
335                 if (data2==0xff){ //all of the useful commands start with 0xff
336                     //after ensuring that this data is good by seeing a 0xff byte, assemble the bytes
337                     StatusBytes = (data2<<24) | (data3<<16) | (data4<<8) | data5 ; //0xff,SB1,SB2,SB3 respectively
338                     ;
339
340                     //
341                     //
342                     //
343                     //
344                     //
345                     //
346
347                     //can post an event to Master vehicle telling we updated the game status
348
349                     //extract the but corresponding to contruction start and see if it changes to "construction
starts"
350                     if ( (StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)>(prevStatusBytes &
GAME_STATUS_SB3_GS_EXTRACT_MASK)) {
351                         EventToPost.EventType=CONSTRUCTION_START;
352                         PostMasterVehicleSM(EventToPost);

```

```

353         printf("Post CONSTRUCTION_START to master vehicle\n\r");
354     }
355
356     //extract the but corresponding to contruction start and see if it changes to "construction
ends"
357     if ( (StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)<(prevStatusBytes &
GAME_STATUS_SB3_GS_EXTRACT_MASK)) {
358         EventToPost.EventType=CONSTRUCTION_END;
359         PostMasterVehicleSM(EventToPost);
360         printf("Post CONSTRUCTION_END to master vehicle\n\r");
361     }
362
363     //updating current score
364     if(get_Team() == GREEN){
365         cur_green_score = queryGoalGreen();
366         if(cur_green_score > prev_green_score){
367             ES_Event score_post;
368             score_post.EventType=SCORE_CHANGED;
369             PostMasterVehicleSM(score_post);
370             printf("Post SCORE_CHANGED to master vehicle\n\r");
371         }
372         prev_green_score = cur_green_score;
373     }
374     else{
375         cur_red_score = queryGoalRed();
376         if(cur_red_score > prev_red_score){
377             ES_Event red_score_post;
378             red_score_post.EventType=SCORE_CHANGED;
379             PostMasterVehicleSM(red_score_post);
380             printf("Post SCORE_CHANGED to master vehicle\n\r");
381         }
382         prev_red_score= cur_red_score;
383     }
384     update_score(queryGoalRed(), queryGoalGreen());
385
386     #ifdef CHECK_ACTIVE_ALWAYS
387     if ((StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)>0){//if the game is active
388
389         if(get_Team() == GREEN){
390             // Check active staging location
391             ActiveGreenStagingLocation=queryActiveStagingGreen();
392
393             if
((ActiveGreenStagingLocation!=0)&&(ActiveGreenStagingLocation!=INVALID_LOCATIONS_ACTIVE)&&(ActiveGreenSta
gingLocation!=prevActiveGreenStagingLocation)){
394                 //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE, do this so it doesn't keep posting
"STAGE_ACTIVE" events
395                 //it's a valid location, and it's different from before
396                 EventToPost.EventType=STAGE_ACTIVE;
397                 EventToPost.EventParam=ActiveGreenStagingLocation;
398                 PostMasterVehicleSM(EventToPost);
399                 printf("\r\nActive Green Staging Location is: %d",ActiveGreenStagingLocation);
400             }
401             prevActiveGreenStagingLocation=ActiveGreenStagingLocation;
402
403             // Check active shooting location
404             ActiveGreenShootingLocation=queryActiveShootingGreen();
405             if
((ActiveGreenShootingLocation!=0)&&(ActiveGreenShootingLocation!=INVALID_LOCATIONS_ACTIVE)&&(ActiveGreenS
hootingLocation!=prevActiveGreenShootingLocation)){
406                 //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE, do this so it doesn't keep posting
"SHOOT_ACTIVE" events
407                 //it's a valid location and it's different from before
408
409                 if(ActiveGreenShootingLocation==4){ //a "4" code implies that all are active
410                     printf("Post last 18 second shooting active (SHOOT_ACTIVE_4)\r\n");
411                     EventToPost.EventType=SHOOT_ACTIVE_4;
412                     PostMasterVehicleSM(EventToPost);
413                     printf("\r\nActive Green Shooting Location is: %d",ActiveGreenShootingLocation);
414                 }
415                 else{ //it's a normal 1,2,3 shooting location
416                     EventToPost.EventType=SHOOT_ACTIVE;

```

```

417         EventToPost.EventParam=ActiveGreenShootingLocation;
418         PostMasterVehicleSM(EventToPost);
419         printf("\r\nActive Green Shooting Location is: %d",ActiveGreenShootingLocation);
420     }
421 }
422     prevActiveGreenShootingLocation=ActiveGreenShootingLocation;
423 }
424 //endif
425
426 //ifdef TEAM_RED
427 if(get_Team() == RED){ //same procedure and reasoning as in green, just different function
names
428
429     // Check staging
430     ActiveRedLocation=queryActiveStagingRed();
431
432     if
433 ((ActiveRedLocation!=0)&&(ActiveRedLocation!=INVALID_LOCATIONS_ACTIVE)&&(ActiveRedLocation!=prevActiveRed
Location)){
434         //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE
435         //it's a valid location at a valid, and it's different from before
436         EventToPost.EventType=STAGE_ACTIVE;
437         EventToPost.EventParam=ActiveRedLocation;
438         PostMasterVehicleSM(EventToPost);
439         printf("\r\nActive Red stagingLocation is: %d",ActiveRedLocation);
440     }
441     prevActiveRedLocation=ActiveRedLocation;
442
443     // Check shooting
444     ActiveRedLocation=queryActiveShootingRed();
445     if
446 ((ActiveRedLocation!=0)&&(ActiveRedLocation!=INVALID_LOCATIONS_ACTIVE)&&(ActiveRedLocation!=prevActiveRed
Location)){
447         //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE
448         //it's a valid location at a valid, and it's different from before
449         if(ActiveRedLocation==4){
450             printf("Post last 18 second shooting active (SHOOT_ACTIVE_4)\r\n");
451             EventToPost.EventType=SHOOT_ACTIVE_4;
452             PostMasterVehicleSM(EventToPost);
453             printf("\r\nActive Red Shooting Location is: %d",ActiveRedLocation);
454         }
455         else{
456             EventToPost.EventType=SHOOT_ACTIVE;
457             EventToPost.EventParam=ActiveRedLocation;
458             PostMasterVehicleSM(EventToPost);
459             printf("\r\nActive Red Shooting Location is: %d",ActiveRedLocation);
460         }
461     }
462     prevActiveRedLocation=ActiveRedLocation;
463 }//end if "red"
464 //endif
465 }//end "if the game is active"
466 #endif
467 }//end if "data is useful, data2=0xff"
468
469
470 //update prevStatusBytes after this comparison
471 prevStatusBytes=StatusBytes;
472 //start the timer for the next game status query
473 ES_Timer_InitTimer(GET_STATUS_TIMER,GET_STATUS_PERIOD);
474 NextState = WAITING;//Decide what the next state will be
475 // for internal transitions, skip changing MakeTransition
476 MakeTransition = true; //mark that we are taking a transition
477 // optionally, consume or re-map this event for the upper
478 // level state machine
479 ReturnEvent.EventType = ES_NO_EVENT;
480 break;
481 // repeat cases as required for relevant events
482
483

```



```

484         case ARRIVED_AT_STAGING: //mainly used when we are stuck, at we are moving around a bit and
restarting
485             ES_Event to_post;
486             printf("Repost arrive at staging to LOC Master\r\n");
487             to_post.EventType = ARRIVED_AT_STAGING;
488             PostLOCMasterSM(to_post);
489             ReturnEvent.EventType = ES_NO_EVENT;
490             break;
491             default:
492             break;
493     }
494 } // end if "no event"
495 else // Current Event is now ES_NO_EVENT. Correction 2/20/17
496 {
497     //Probably means that CurrentEvent was consumed by lower level
498     ReturnEvent = CurrentEvent; // in that case update ReturnEvent too.
499 }
500 break;
501
502
503
504 case SENDING_TO_LOC_AT_STAGING :           // If current state is state one
505
506 // Execute During function for state one. ES_ENTRY & ES_EXIT are
507 // processed here allow the lower level state machines to re-map
508 // or consume the event
509 CurrentEvent = During_SENDING_TO_LOC_AT_STAGING_State(CurrentEvent);
510 //process any events
511 if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
512 {
513     switch (CurrentEvent.EventType)
514     {
515
516
517         //early testing code
518         #ifdef TEST_LOC_STAGE
519         case Input_Freq:
520             frq_index++;
521             if (frq_index >= period_table_length){
522                 frq_index = 0;
523             }
524             printf("Select Period is %d, Code is %d\r\n", PERIOD_TABLE[frq_index],
525                 FreqCode[frq_index]);
526             break;
527         case SEND_FIRST_REPORT_WITH_OUR_FAKE_FREQ:
528             ES_Event EventToPostFake;
529             EventToPostFake.EventType=SEND_FIRST_REPORT;
530             EventToPostFake.EventParam=FreqCode[frq_index];
531             PostLOCMasterSM(EventToPostFake);
532             break;
533         case SEND_SECOND_REPORT_WITH_OUR_FAKE_FREQ:
534             ES_Event EventToPostFake2;
535             EventToPostFake2.EventType=SEND_SECOND_REPORT;
536             EventToPostFake2.EventParam=FreqCode[frq_index];
537             PostLOCMasterSM(EventToPostFake2);
538             break;
539         #endif
540
541
542
543
544         case SEND_FIRST_REPORT: //if we are asked to send the first report
545             //clear the active location first, we are at a new round
546             ActiveLocation=0;
547             printf("Enter sending first report\r\n");
548             //clear all the flags for previous report sent and ack, might be repeated, but better safe than
sorry
549             FlagFirstReportACK=0;
550             FlagSecondReportACK=0;
551             FlagSentOneReport=0;
552             FlagSentTwoReport=0;
553

```



```

554 // Execute action function for state one :
555 if (FlagSentReportWithin200ms==1)
556 {
557     //printf("Waiting for 200ms\r\n");
558     NextState = WAITING_FOR_200MS_TIMEOUT;//Decide what the next state will be
559     // for internal transitions, skip changing MakeTransition
560     MakeTransition = true; //mark that we are taking a transition
561     // optionally, consume or re-map this event for the upper
562     // level state machine
563     ReturnEvent.EventType = ES_NO_EVENT;
564 }
565 else if(FlagFreqConsumed==0){//we have a freq to use
566     //write the frequency to LOC
567
568     printf("Writing first frequency to LOC\r\n");
569     printf("Event param is %d\r\n", CurrentEvent.EventParam);
570     printf("Frequency report is: %d \n\r", (REPORT_FRONT_BITS) | (CurrentEvent.EventParam));
571
572     SPI_Write((REPORT_FRONT_BITS) | (CurrentEvent.EventParam)); //REPORT_FRONT_BITS=0x80 is 0b1000
0000, the event parameter is
573     //passed from input capture response
574     //basically assembling the package with our frequency report
575     SPI_Write(0x00);
576     SPI_Write(0x00);
577     SPI_Write(0x00);
578     SPI_Write(0x00);
579
580     //enable the EOT interrupt
581     enable_SPI_Interupt();
582
583     //raised the flag, we have sent 1 already
584     FlagSentOneReport=1;
585
586     //raise the flag, indicating that we have sent a report, don't
587     //send again for another 200 ms
588     FlagSentReportWithin200ms=1;
589     //start a 200 ms timer to clear this FlagSentReprotWithin200ms
590     ES_Timer_InitTimer(REPORT_TIMER,TWO_HUNDRED_MS);
591     NextState = RECEIVING_FROM_LOC_AT_STAGING;//Decide what the next state will be
592     // for internal transitions, skip changing MakeTransition
593     MakeTransition = true; //mark that we are taking a transitio
594     // optionally, consume or re-map this event for the upper
595     // level state machine
596     ReturnEvent.EventType = ES_NO_EVENT;
597 }//end the within 200ms check
598 break;
599 // repeat cases as required for relevant events
600
601 case SEND_SECOND_REPORT: //if we are asked to send the second report
602 printf("Enter sending second report\n\r");
603 // Execute action function for state one :
604 if (FlagSentReportWithin200ms==1)
605 {
606     NextState = WAITING_FOR_200MS_TIMEOUT;//Decide what the next state will be
607     // for internal transitions, skip changing MakeTransition
608     MakeTransition = true; //mark that we are taking a transition
609     // optionally, consume or re-map this event for the upper
610     // level state machine
611     ReturnEvent.EventType = ES_NO_EVENT;
612 }
613 else if(FlagFreqConsumed==0){//if we have a freq to use
614
615     //write the frequency to LOC
616     printf("Writing second frequency to LOC\r\n");
617     printf("Frequency report is: 0x%02x \n\r", (REPORT_FRONT_BITS) | (CurrentEvent.EventParam));
618     SPI_Write((REPORT_FRONT_BITS) | (CurrentEvent.EventParam)); //REPORT_FRONT_BITS=0x80 is 0b1000
0000,
619     //the event parameter is passed from input capture response
620     SPI_Write(0x00);
621     SPI_Write(0x00);
622     SPI_Write(0x00);
623     SPI_Write(0x00);

```

```

624         //enable the EOT interrupt
625         enable_SPI_Interupt();
626
627
628         //raised the flag, we have sent a "second report" already
629         FlagSentTwoReport=1; //only difference with SENT_ONE_REPORT
630         //raise the flag, indicating that we have sent a report, don't
631         //send again for another 200 ms
632         FlagSentReportWithin200ms=1;
633         //start a 200 ms timer to clear this Flag
634         ES_Timer_InitTimer(REPORT_TIMER,TWO_HUNDRED_MS);
635         NextState = RECEIVING_FROM_LOC_AT_STAGING; //Decide what the next state will be
636         // for internal transitions, skip changing MakeTransition
637         MakeTransition = true; //mark that we are taking a transition
638         // optionally, consume or re-map this event for the upper
639         // level state machine
640         ReturnEvent.EventType = ES_NO_EVENT;
641     } //end the within 200ms check
642     break;
643     // repeat cases as required for relevant events
644
645     case GO_QUERY_REPORT_RESPONSE : //If event is event one
646     printf("Asked to go query report response\n\r");
647     // Execute action function
648     //write the query for report response command to LOC
649     SPI_Write(REPORT_RESPONSE_COMMAND);
650     //SPI_Write(112);
651     SPI_Write(0x00);
652     SPI_Write(0x00);
653     SPI_Write(0x00);
654     SPI_Write(0x00);
655     //enable the EOT interrupt
656     enable_SPI_Interupt();
657     NextState = RECEIVING_FROM_LOC_AT_STAGING; //Decide what the next state will be
658     // for internal transitions, skip changing MakeTransition
659     MakeTransition = true; //mark that we are taking a transition
660     // optionally, consume or re-map this event for the upper
661     // level state machine
662     ReturnEvent.EventType = ES_NO_EVENT;
663     break;
664     // repeat cases as required for relevant events
665
666
667     default:
668     break;
669 }
670
671 } // end if "not no-event"
672 else // Current Event is now ES_NO_EVENT. Correction 2/20/17
673 {
674     //Probably means that CurrentEvent was consumed by lower level
675     ReturnEvent = CurrentEvent; // in that case update ReturnEvent too.
676 }
677 break;
678
679 case RECEIVING_FROM_LOC_AT_STAGING : // If current state is state one
680 printf("Enter receiving from LOC at staging state\n\r");
681 // Execute During function for state one. ES_ENTRY & ES_EXIT are
682 // processed here allow the lower level state machines to re-map
683 // or consume the event
684 CurrentEvent = During_RECEIVING_FROM_LOC_AT_STAGING_State(CurrentEvent);
685 //process any events
686 if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
687 {
688     switch (CurrentEvent.EventType)
689     {
690
691         //some test code to use keyboard to step through the process
692         #ifdef TEST_LOC_STAGE
693         case Set_SentOne_Flag_1:
694             FlagSentOneReport = 1;
695             printf("FlagSentOneReport = 1\n\r");

```

```

696     break;
697     case Set_SentOne_Flag_0:
698     FlagSentOneReport = 0;
699     printf("FlagSentOneReport = 0\n\r");
700     break;
701     case Set_SentTwo_Flag_1:
702     FlagSentTwoReport = 0;
703     printf("FlagSentTwoReport = 0\n\r");
704     break;
705     case Set_SentTwo_Flag_0:
706     FlagSentTwoReport = 0;
707     printf("FlagSentTwoReport = 0\n\r");
708     break;
709     case Set_first_ack_1:
710     FlagFirstReportACK = 1;
711     printf("FlagFirstReportACK = 1\n\r");
712     break;
713     case Set_first_ack_0:
714     FlagFirstReportACK = 0;
715     printf("FlagFirstReportACK = 0\n\r");
716     break;
717     case Set_second_ack_1:
718     FlagSecondReportACK = 0;
719     printf("FlagSecondReportACK = 0\n\r");
720     break;
721     case Set_second_ack_0:
722     FlagSecondReportACK = 0;
723     printf("FlagSecondReportACK = 0\n\r");
724     break;
725     case Set_Data3_Ready:
726     data3 = RESPONSE_READY_CODE;
727     printf("data3 = RESPONSE_READY_CODE\n\r");
728     break;
729     case Set_Data3_Notready:
730     data3 = RESPONSE_NOT_READY_CODE;
731     printf("data3 = RESPONSE_NOT_READY_CODE\n\r");
732     break;
733     case Set_Data4_ACK:
734     data4=ACK_CODE;
735     printf("data4== ACK_CODE\n\r");
736     break;
737     case Set_Data4_NACK:
738     data4= NACK_CODE;
739     printf("data4= NACK_CODE\n\r");
740     break;
741
742     #endif
743
744
745
746
747     case RESTART_VERIFY_FREQ://we are stuck, restart the process
748     NextState = WAITING;
749     MakeTransition = true;
750     ReturnEvent.EventType = ES_NO_EVENT;
751     break;
752
753
754     case GO_QUERY_REPORT_RESPONSE:
755     printf("Send QUERY to LOC\r\n");
756     SPI_Write(REPORT_RESPONSE_COMMAND); //112 works
757     SPI_Write(0x00);
758     SPI_Write(0x00);
759     SPI_Write(0x00);
760     SPI_Write(0x00);
761     //enable the EOT interrupt
762     enable_SPI_Interupt();
763     break;
764
765     case ES_EOT : //If event is ES_EOT, END OF FIVE BYTES, not one, FIVE!
766     // Execute action function for state one : event one
767

```

```

768
769     //printf("\n B1 received staging: 0x%02x\r\n", data1); //put a lot of spaces around this so I
can find it easily
770     //printf("B2 received staging: 0x%02x\r\n", data2);
771     //printf("B3 received staging: 0x%02x\r\n", data3);
772     //printf("B4 received staging: 0x%02x\r\n", data4);
773     //printf("B5 received staging: 0x%02x\r\n", data5);
774
775
776     printf("Stable measured freq code: %d\n\n\r", stableMeasFreqCode);
777
778
779     //if we have sent the first report and not acknowledged
780     if ( (FlagSentOneReport==1) && (FlagFirstReportACK==0)) {
781         //printf("Enter Sent one report, first report NOT ACK\n\r");
782         //Extract to see if the response is ready
783         if (data3==RESPONSE_READY_CODE) { //data3=RR, response ready code is 0xAA
784             //consume the frequency once the response is ready and our frequency is processed
785             FlagFreqConsumed=1;
786             if ((data4&(BIT7HI|BIT6HI))==ACK_CODE) { //data4 is RS byte, ACK is 0x00
787                 //printf("First report ACK\n\r");
788                 //update the flags to encode our current step
789                 FlagSentOneReport=1;
790                 FlagSentTwoReport=0;
791                 FlagFirstReportACK=1;
792                 FlagSecondReportACK=0;
793
794                 NextState=SENDING_TO_LOC_AT_STAGING; //update the next state
795                 //
796                 MakeTransition=true;
797                 ReturnEvent.EventType=ES_NO_EVENT;
798             }
799             else { //we get NACK or Inactive, first report failed
800                 printf("First Report NACK\n\r");
801                 //update the flags to encode our current step, we go back to the beginning
802                 FlagSentOneReport=0;
803                 FlagSentTwoReport=0;
804                 FlagFirstReportACK=0;
805                 FlagSecondReportACK=0;
806                 //just consume the frequency and don't post yet, let the input capture do the posting
807                 FlagFreqConsumed=1;
808                 NextState=SENDING_TO_LOC_AT_STAGING;
809
810                 MakeTransition=true;
811                 ReturnEvent.EventType=ES_NO_EVENT;
812             }
813
814             } // ends the "checking response ready" if statement
815         else { //the response code is not ready, keep querying
816             printf("Response is NOT READY\n\r");
817             NextState=SENDING_TO_LOC_AT_STAGING;
818             //keep querying until we get response ready byte
819             EventToPost.EventType=GO_QUERY_REPORT_RESPONSE;
820             PostLOCMasterSM(EventToPost);
821
822             MakeTransition=true;
823             ReturnEvent.EventType=ES_NO_EVENT;
824         }
825     }
826     //end of after first report and dealing with first ACK
827
828     //now deal with after having one successful report
829     if ( (FlagSentTwoReport==1) && (FlagSecondReportACK==0)) {
830         printf("Enter sent two reports, second one NOT ACK\n\r");
831         //Extract to see if the response is ready
832         if (data3==RESPONSE_READY_CODE) { //data3=RR, response ready code is 0xAA
833             //consume the frequency once the response is ready and our frequency is processed
834             FlagFreqConsumed=1;
835             if ((data4&(BIT7HI|BIT6HI))==ACK_CODE) { //data4 is RS byte, ACK is 0x00
836                 printf("Second report ACK\n\r");
837                 //update the flags to encode our current step
838                 FlagFreqConsumed=1; //consume the frequency after ACKed

```

```

839         FlagSentOneReport=1;
840         FlagSentTwoReport=1;
841         FlagFirstReportACK=1;
842         FlagSecondReportACK=1; //second report is successful
843         //extract the active location
844         ActiveLocation= (data5 & (BIT0HI|BIT1HI|BIT2HI|BIT3HI));
845         NextState=WAITING;
846         EventToPost.EventType=FINISHED_STAGING;
847         //FlagFreqConsumed=0;//consume the frequency after ACKed
848         printf("Finished Staging\n\r");
849         printf("Posting finished staging to mastervehicle SM\r\n");
850         PostMasterVehicleSM(EventToPost);//tell the master vehicle that we are done
851         //start the timer for the next game status query otherwise it wouldn't get triggered
852         ES_Timer_InitTimer(GET_STATUS_TIMER,GET_STATUS_PERIOD);
853         MakeTransition=true;
854         ReturnEvent.EventType=ES_NO_EVENT;
855     }
856     else{ //we get NACK or Inactive
857         printf("Failed at 2nd report, NACK\n\r");
858         //update the flags to encode our current step
859         FlagSentOneReport=0;
860         FlagSentTwoReport=0;
861         FlagFirstReportACK=0;
862         FlagSecondReportACK=0;
863         NextState=SENDING_TO_LOC_AT_STAGING;
864         //just consume the frequency and don't post yet, let the input capture do the posting
865         FlagFreqConsumed=1;
866         MakeTransition=true;
867         ReturnEvent.EventType=ES_NO_EVENT;
868     }
869 }
870 }
871 else{//the reponse code is not ready, keep querying
872     printf("Response is NOT READY\n\r");
873     NextState=SENDING_TO_LOC_AT_STAGING;
874     //keep querying until we get response byte
875     EventToPost.EventType=GO_QUERY_REPORT_RESPONSE;
876     PostLOCMasterSM(EventToPost);
877     MakeTransition=true;
878     ReturnEvent.EventType=ES_NO_EVENT;
879 }
880 } //end of after first report and dealing with first ACK
881
882 break;
883 // repeat cases as required for relevant events
884 default:
885     break;
886 }
887 } // end if "no event"
888 else // Current Event is now ES_NO_EVENT. Correction 2/20/17
889 {
890     //Probably means that CurrentEvent was consumed by lower level
891     ReturnEvent = CurrentEvent; // in that case update ReturnEvent too.
892 }
893 break;
894
895 case WAITING_FOR_200MS_TIMEOUT : // If current state is waiting for 200ms timeout (due to
report)
896
897 // Execute During function for state one. ES_ENTRY & ES_EXIT are
898 // processed here allow the lower level state machines to re-map
899 // or consume the event
900 CurrentEvent = DuringWait200msState(CurrentEvent);
901
902 //process any events
903 if ( CurrentEvent.EventType != ES_NO_EVENT ) //If an event is active
904 {
905     switch (CurrentEvent.EventType)
906     {
907
908
909

```

```

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
982     // Execute action function for state one : event one
983     NextState = SENDING_TO_LOC_AT_STAGING; //Decide what the next state will be
984     //the program is sent to this state because when asked to send report, 200ms hasn't expired
985
986     //just consumed the frequency and let input capture post
987     FlagFreqConsumed=1;
988
989     #ifdef TEST_LOC_STAGE
990     EventToPost.EventParam=FreqCode[frq_index];
991     #endif
992
993     // for internal transitions, skip changing MakeTransition
994     MakeTransition = true; //mark that we are taking a transition
995     // optionally, consume or re-map this event for the upper
996     // level state machine
997     ReturnEvent.EventType = ES_NO_EVENT;
998 }
999
1000     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
1001     // Execute action function for state one : event one
1002     NextState = SENDING_TO_LOC_AT_STAGING; //Decide what the next state will be
1003     //the program is sent to this state because when asked to send report, 200ms hasn't expired
1004
1005     //just consumed the frequency and let input capture post
1006     FlagFreqConsumed=1;
1007
1008     //so we return to where the program came from
1009     //EventToPost.EventType=SEND_SECOND_REPORT;
1010     //EventToPost.EventParam=stableMeasFreqCode;
1011     #ifdef TEST_LOC_STAGE
1012     EventToPost.EventParam=FreqCode[frq_index];
1013     #endif
1014     //PostLOCMasterSM(EventToPost);
1015     // for internal transitions, skip changing MakeTransition
1016     MakeTransition = true; //mark that we are taking a transition
1017     // optionally, consume or re-map this event for the upper
1018     // level state machine
1019     ReturnEvent.EventType = ES_NO_EVENT;
1020 }
1021
1022 } //end checking the timeout is from the 200ms report timer
1023 break; //break the timeout case
1024 // repeat cases as required for relevant events
1025
1026 //
1027 default:
1028     break; //break default, for switching event type
1029 } //end switch event type
1030
1031
1032 } //end if "not no event"
1033 else // Current Event is now ES_NO_EVENT. Correction 2/20/17
1034 {
1035     //Probably means that CurrentEvent was consumed by lower level
1036     ReturnEvent = CurrentEvent; // in that case update ReturnEvent too.
1037 }
1038 break; //break the waiting 200ms state
1039
1040 default:
1041     break; //break default, for switching states
1042 } //end switch of states
1043 // If we are making a state transition
1044 if (MakeTransition == true)
1045 {
1046     // Execute exit function for current state
1047     CurrentEvent.EventType = ES_EXIT;
1048     RunLOCMasterSM(CurrentEvent);
1049
1050     CurrentState = NextState; //Modify state variable

```



```

1050
1051     // Execute entry function for new state
1052     // this defaults to ES_ENTRY
1053     RunLOCMasterSM(EntryEventKind);
1054 }
1055
1056 // in the absence of an error the top level state machine should
1057 // always return ES_NO_EVENT, which we initialized at the top of func
1058 return(ReturnEvent);
1059 }
1060 /*****
1061
1062 *****/
1063 void StartLOCMasterSM ( ES_Event CurrentEvent )
1064 {
1065     // if there is more than 1 state to the top level machine you will need
1066     // to initialize the state variable
1067     //our initial state is WAITING
1068     CurrentState = WAITING;
1069     // now we need to let the Run function init the lower level state machines
1070     // use LocalEvent to keep the compiler from complaining about unused var
1071     RunLOCMasterSM(CurrentEvent);
1072     return;
1073 }
1074
1075
1076 /*****
1077 private functions
1078 *****/
1079
1080
1081
1082 static ES_Event DuringWaitingState( ES_Event Event)
1083 {
1084     ES_Event ReturnEvent = Event; // assume no re-mapping or consumption
1085
1086     // process ES_ENTRY, ES_ENTRY_HISTORY & ES_EXIT events
1087     if ( (Event.EventType == ES_ENTRY) ||
1088         (Event.EventType == ES_ENTRY_HISTORY) )
1089     {
1090         // implement any entry actions required for this state machine
1091         // after that start any lower level machines that run in this state
1092
1093         //printf("Enter Waiting of LOC Master SM\r\n");
1094
1095         //set all the flags related to handshaking at staging area to default value
1096         FlagSentOneReport=0;
1097         FlagSentTwoReport=0;
1098         FlagFirstReportACK=0;
1099         FlagSecondReportACK=0;
1100         FlagSentReportWithin200ms=0;
1101         FlagFreqConsumed=1;
1102     }
1103     else if ( Event.EventType == ES_EXIT )
1104     {
1105         // on exit, give the lower levels a chance to clean up first
1106         //RunLowerLevelSM(Event);
1107         // repeat for any concurrently running state machines
1108         // now do any local exit functionality
1109         // printf("Exit Waiting State\r\n");
1110     }else
1111         // do the 'during' function for this state
1112     {
1113         // run any lower level state machine
1114         // ReturnEvent = RunLowerLevelSM(Event);
1115
1116         // repeat for any concurrent lower level machines
1117
1118         // do any activity that is repeated as long as we are in this state
1119     }
1120 }
1121 // return either Event, if you don't want to allow the lower level machine

```

```

1122 // to remap the current event, or ReturnEvent if you do want to allow it.
1123 return(ReturnEvent);
1124 }
1125
1126 static ES_Event During_GAME_STATUS_SENDING_TO_LOC_State( ES_Event Event)
1127 {
1128     ES_Event ReturnEvent = Event; // assume no re-mapping or consumption
1129
1130     // process ES_ENTRY, ES_ENTRY_HISTORY & ES_EXIT events
1131     if ( (Event.EventType == ES_ENTRY) ||
1132         (Event.EventType == ES_ENTRY_HISTORY) )
1133     {
1134         // implement any entry actions required for this state machine
1135         // after that start any lower level machines that run in this state
1136         //StartLowerLevelSM( Event );
1137         // printf("Enter GAME_STATUS_SENDING_TO_LOC State\r\n");
1138         // repeat the StartxxxSM() functions for concurrent state machines
1139         // on the lower level
1140     }
1141     else if ( Event.EventType == ES_EXIT )
1142     {
1143         // on exit, give the lower levels a chance to clean up first
1144         //RunLowerLevelSM(Event);
1145         // repeat for any concurrently running state machines
1146         // now do any local exit functionality
1147         // printf("Exit GAME_STATUS_SENDING_TO_LOC State\r\n");
1148     }else
1149     // do the 'during' function for this state
1150     {
1151         // run any lower level state machine
1152         // ReturnEvent = RunLowerLevelSM(Event);
1153
1154         // repeat for any concurrent lower level machines
1155
1156         // do any activity that is repeated as long as we are in this state
1157     }
1158     // return either Event, if you don't want to allow the lower level machine
1159     // to remap the current event, or ReturnEvent if you do want to allow it.
1160     return(ReturnEvent);
1161 }
1162
1163 static ES_Event During_GAME_STATUS_RECEIVING_FROM_LOC_State( ES_Event Event)
1164 {
1165     ES_Event ReturnEvent = Event; // assume no re-mapping or consumption
1166
1167     // process ES_ENTRY, ES_ENTRY_HISTORY & ES_EXIT events
1168     if ( (Event.EventType == ES_ENTRY) ||
1169         (Event.EventType == ES_ENTRY_HISTORY) )
1170     {
1171         // implement any entry actions required for this state machine
1172         // after that start any lower level machines that run in this state
1173         //StartLowerLevelSM( Event );
1174         // printf("Enter GAME_STATUS_RECEIVING_FROM_LOC State\r\n");
1175         // repeat the StartxxxSM() functions for concurrent state machines
1176         // on the lower level
1177     }
1178     else if ( Event.EventType == ES_EXIT )
1179     {
1180         // on exit, give the lower levels a chance to clean up first
1181         //RunLowerLevelSM(Event);
1182         // repeat for any concurrently running state machines
1183         // now do any local exit functionality
1184         // printf("Exit GAME_STATUS_RECEIVING_FROM_LOC State\r\n");
1185     }else
1186     // do the 'during' function for this state
1187     {
1188         // run any lower level state machine
1189         // ReturnEvent = RunLowerLevelSM(Event);
1190
1191         // repeat for any concurrent lower level machines
1192
1193         // do any activity that is repeated as long as we are in this state

```

```

1194     }
1195     // return either Event, if you don't want to allow the lower level machine
1196     // to remap the current event, or ReturnEvent if you do want to allow it.
1197     return(ReturnEvent);
1198 }
1199
1200 static ES_Event During_SENDING_TO_LOC_AT_STAGING_State( ES_Event Event)
1201 {
1202     ES_Event ReturnEvent = Event; // assume no re-mapping or consumption
1203
1204     // process ES_ENTRY, ES_ENTRY_HISTORY & ES_EXIT events
1205     if ( (Event.EventType == ES_ENTRY) ||
1206         (Event.EventType == ES_ENTRY_HISTORY) )
1207     {
1208         //printf("Enter Sending to Loc at staging state\n\r");
1209         // implement any entry actions required for this state machine
1210         // after that start any lower level machines that run in this state
1211         //StartLowerLevelSM( Event );
1212         //printf("Enter SENDING_TO_LOC_AT_STAGING State\r\n");
1213         // repeat the StartxxxSM() functions for concurrent state machines
1214         // on the lower level
1215     }
1216     else if ( Event.EventType == ES_EXIT )
1217     {
1218         // on exit, give the lower levels a chance to clean up first
1219         //RunLowerLevelSM(Event);
1220         // repeat for any concurrently running state machines
1221         // now do any local exit functionality
1222         //printf("Exit SENDING_TO_LOC_AT_STAGING State\r\n");
1223     }else
1224     {
1225         // do the 'during' function for this state
1226
1227         // run any lower level state machine
1228         // ReturnEvent = RunLowerLevelSM(Event);
1229
1230         // repeat for any concurrent lower level machines
1231
1232         // do any activity that is repeated as long as we are in this state
1233     }
1234     // return either Event, if you don't want to allow the lower level machine
1235     // to remap the current event, or ReturnEvent if you do want to allow it.
1236     return(ReturnEvent);
1237 }
1238
1239 static ES_Event During_RECEIVING_FROM_LOC_AT_STAGING_State( ES_Event Event)
1240 {
1241     ES_Event ReturnEvent = Event; // assume no re-mapping or consumption
1242
1243     // process ES_ENTRY, ES_ENTRY_HISTORY & ES_EXIT events
1244     if ( (Event.EventType == ES_ENTRY) ||
1245         (Event.EventType == ES_ENTRY_HISTORY) )
1246     {
1247         // implement any entry actions required for this state machine
1248         // after that start any lower level machines that run in this state
1249         //StartLowerLevelSM( Event );
1250         //printf("Enter RECEIVING_FROM_LOC_AT_STAGING State\r\n");
1251         // repeat the StartxxxSM() functions for concurrent state machines
1252         // on the lower level
1253     }
1254     else if ( Event.EventType == ES_EXIT )
1255     {
1256         // on exit, give the lower levels a chance to clean up first
1257         //RunLowerLevelSM(Event);
1258         // repeat for any concurrently running state machines
1259         // now do any local exit functionality
1260         //printf("Exit RECEIVING_FROM_LOC_AT_STAGING State\r\n");
1261     }else
1262     {
1263         // do the 'during' function for this state
1264
1265         // run any lower level state machine
1266         // ReturnEvent = RunLowerLevelSM(Event);
1267
1268         // repeat for any concurrent lower level machines

```

```

1266
1267     // do any activity that is repeated as long as we are in this state
1268 }
1269 // return either Event, if you don't want to allow the lower level machine
1270 // to remap the current event, or ReturnEvent if you do want to allow it.
1271 return(ReturnEvent);
1272 }
1273
1274 static ES_Event DuringWait200msState( ES_Event Event)
1275 {
1276     ES_Event ReturnEvent = Event; // assume no re-mapping or consumption
1277
1278     // process ES_ENTRY, ES_ENTRY_HISTORY & ES_EXIT events
1279     if ( (Event.EventType == ES_ENTRY) ||
1280         (Event.EventType == ES_ENTRY_HISTORY) )
1281     { printf("Enter WAITING for 200ms time out state\n\r");
1282       // implement any entry actions required for this state machine
1283       // after that start any lower level machines that run in this state
1284       //StartLowerLevelSM( Event );
1285       //printf("Enter RECEIVING_FROM_LOC_AT_STAGING State\r\n");
1286       // repeat the StartxxxSM() functions for concurrent state machines
1287       // on the lower level
1288     }
1289     else if ( Event.EventType == ES_EXIT )
1290     {
1291         printf("Exiting wait200ms state\n\r");
1292         // on exit, give the lower levels a chance to clean up first
1293         //RunLowerLevelSM(Event);
1294         // repeat for any concurrently running state machines
1295         // now do any local exit functionality
1296         //printf("Exit RECEIVING_FROM_LOC_AT_STAGING State\r\n");
1297     }else
1298         // do the 'during' function for this state
1299     {
1300         // run any lower level state machine
1301         // ReturnEvent = RunLowerLevelSM(Event);
1302
1303         // repeat for any concurrent lower level machines
1304
1305         // do any activity that is repeated as long as we are in this state
1306     }
1307     // return either Event, if you don't want to allow the lower level machine
1308     // to remap the current event, or ReturnEvent if you do want to allow it.
1309     return(ReturnEvent);
1310 }
1311 /*****SPI FUNCTIONS*****/
1312
1313 void SPI_Init(void){
1314     // Enable the clock to the GPIO Port (we are going to use Port A)
1315     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
1316     // Enable clock to SSI - set to SSI Module 0
1317     HWREG(SYSCTL_RCGCSSI) |= SYSCTL_RCGCSSI_R0;
1318
1319     // Wait for GPIO Port to be ready by killing a few cycles
1320     while((HWREG(SYSCTL_RCGCGPIO) & SYSCTL_RCGCGPIO_R0) != SYSCTL_RCGCGPIO_R0){}
1321     //printf("right supanath 123456789 BANANA abc before while loop\n\r");
1322     // Program the GPIO to use the alternate functions on the SSI pins PA2,3,4,5
1323     // HWREG(GPIO_PORTA_BASE + GPIO_O_AFSEL) = (HWREG(GPIO_PORTA_BASE + GPIO_O_AFSEL)
1324     //      & 0xfffffff0) | (BIT2HI | BIT3HI | BIT4HI | BIT5HI);
1325     HWREG(GPIO_PORTA_BASE + GPIO_O_AFSEL) |= (BIT2HI | BIT3HI | BIT4HI | BIT5HI);
1326     // Set Mux position in GPIOCTL to select the SSI use of the pins
1327     HWREG(GPIO_PORTA_BASE+GPIO_O_PCTL) =
1328     (HWREG(GPIO_PORTA_BASE+GPIO_O_PCTL) & 0xff0000ff) + (2<<(5*BitsPerNibble))
1329     + (2<<(4*BitsPerNibble)) + (2<<(3*BitsPerNibble)) + (2<<(2*BitsPerNibble));
1330     // Program the port lines for digital I/O
1331     HWREG(GPIO_PORTA_BASE+GPIO_O_DEN) |= (BIT2HI | BIT3HI | BIT4HI | BIT5HI);
1332     // Program the required data directions on the port line
1333     HWREG(GPIO_PORTA_BASE+GPIO_O_DIR) &= BIT4LO; //PA4 is the input (receiver line)
1334     HWREG(GPIO_PORTA_BASE+GPIO_O_DIR) |= (BIT2HI | BIT3HI | BIT5HI);
1335     // If using SPI mode 3, program the pull-up on the clock line
1336     //HWREG(GPIO_PORTD_BASE + GPIO_O_PUR) |= BIT0HI;
1337     HWREG(GPIO_PORTA_BASE + GPIO_O_PUR) |= BIT2HI;

```

```

1338 // Wait for the SSI0 to be ready
1339 while((HWREG(SYSCTL_RCGCSSI) & SYSCTL_RCGCSSI_R0) != SYSCTL_RCGCSSI_R0){}
1340 // Make sure that the SSI is disabled before programming mode bits
1341 HWREG(SSIO_BASE + SSI_O_CR1) = HWREG(SSIO_BASE + SSI_O_CR1) & ~SSI_CR1_SSE;
1342 // Select Master mode and TXRES indicating EOT
1343 HWREG(SSIO_BASE + SSI_O_CR1) &= ~SSI_CR1_MS; //Set to 0 for master
1344 HWREG(SSIO_BASE + SSI_O_CR1) |= SSI_CR1_EOT; //Set EOT to 1 (for interrupt)
1345 // Configure the SSI clock source to the system clock
1346 HWREG(SSIO_BASE + SSI_O_CC) = (HWREG(SSIO_BASE + SSI_O_CC) & ~SSI_CC_CS_M) | SSI_CC_CS_SYSPLL;
1347 // Configure the clock pre-scaler: here we want CPSDVSR = , 1+SCR =
1348 HWREG(SSIO_BASE + SSI_O_CPSR) = (HWREG(SSIO_BASE + SSI_O_CPSR) &
1349 ~SSI_CPSR_CPSDVSR_M) | CPSDVSR_PRESCALER; //set CPSDVSR = 80
1350 // Configure clock rate (SCR) - 0, phase (SPH)- 1 and
1351 // polarity (SPO)- 1 ,mode (FRF) - freescale(0) and datasize (DSS) - 8 bit
1352 HWREG(SSIO_BASE + SSI_O_CR0) = (HWREG(SSIO_BASE + SSI_O_CR0) & 0xffff0000)
1353 | ((SCR << 8) | SSI_CR0_SPH | SSI_CR0_SPO | SSI_CR0_DSS_8 | SSI_CR0_FRF_MOTO);
1354 // Locally Enable Interrupts (TXIM in SSIIM)
1355 enable_SPI_Interrupt();
1356 // Enable SSI
1357 HWREG(SSIO_BASE + SSI_O_CR1) = (HWREG(SSIO_BASE + SSI_O_CR1) & ~SSI_CR1_SSE) | SSI_CR1_SSE;
1358 // Globally enable interrupts
1359 __enable_irq();
1360 // Enable the NVIC interrupt for the SSI when starting to transmit
1361 // enable SSI3 interrupt in the NVIC, it is interrupt number 7 so appears in EN0 at bit 7
1362 HWREG(NVIC_EN0) = BIT7HI;
1363
1364 //make sure we disable loopback mode
1365 HWREG(SSIO_BASE + SSI_O_CR1) &= (~SSI_CR1_LBM); //DISABLE LOOP BACK MODE FOR SURE
1366 printf("finsihed SPI init\n\r");
1367 }
1368 // We enable or disable interrupt by setting or clearing TXIM
1369 static void enable_SPI_Interrupt(void){
1370     HWREG(SSIO_BASE+SSI_O_IM) |= SSI_IM_TXIM;
1371 }
1372
1373 static void disable_SPI_Interrupt(void){
1374     HWREG(SSIO_BASE+SSI_O_IM) &= ~SSI_IM_TXIM;
1375 }
1376
1377 void SPI_Interrupt_Response(void){\
1378
1379     disable_SPI_Interrupt(); //disable the interrupt
1380     //we have more bytes now, each read is 8-bit by initialization
1381
1382     //consecutive read try
1383     data1=SPI_Read();
1384     //printf("Passed data1\n\r");
1385     data2=SPI_Read();
1386     data3=SPI_Read();
1387     data4=SPI_Read();
1388     data5=SPI_Read();
1389     ES_Event new_event; //finish 5 bytes sequence, post EOT of 5 bytes
1390     new_event.EventType = ES_EOT;
1391     PostLOCMasterSM(new_event);
1392
1393
1394 }
1395
1396 // Set the data to SPI register
1397 void SPI_Write(uint8_t data){
1398     HWREG(SSIO_BASE+SSI_O_DR) = data;
1399 }
1400
1401 // Read the data from SPI register
1402 uint8_t SPI_Read(void){
1403     uint8_t dataHolder;
1404     dataHolder = HWREG(SSIO_BASE+SSI_O_DR);
1405     return dataHolder;
1406 }
1407 /*****Other functions*****/
1408 uint32_t QueryGameStatus(void){
1409

```

```

1410     return (StatusBytes);
1411 }
1412
1413 uint8_t QueryActiveLocation(void){
1414
1415     return (ActiveLocation);
1416 }
1417
1418
1419
1420 /*****Input Capture related to Hall Sensor*****/
1421 void InitInputCapture_Hall( void ){
1422     // start by enabling the clock to the timer (Wide Timer 5)
1423     HWREG(SYSCTL_RCGCTIMER) |= SYSCTL_RCGCTIMER_R5;
1424     // enable the clock to Port D
1425     HWREG(SYSCTL_RCGCGPIO) |= SYSCTL_RCGCGPIO_R3;
1426     // since we added this Port D clock init, we can immediately start
1427     // into configuring the timer, no need for further delay
1428     // make sure that timer (Timer A) is disabled before configuring
1429     HWREG(WTIMER5_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEN;
1430     // set it up in 32bit wide (individual, not concatenated) mode
1431     // the constant name derives from the 16/32 bit timer, but this is a 32/64
1432     // bit timer so we are setting the 32bit mode
1433     HWREG(WTIMER5_BASE+TIMER_O_CFG) = TIMER_CFG_16_BIT;
1434     // we want to use the full 32 bit count, so initialize the Interval Load
1435     // register to 0xffff.ffff (its default value :-))
1436     HWREG(WTIMER5_BASE+TIMER_O_TAILR) = 0xffffffff;
1437     // set up timer A in capture mode (TAMR=3, TAAMS = 0),
1438     // for edge time (TACMR = 1) and up-counting (TACDIR = 1)
1439     HWREG(WTIMER5_BASE+TIMER_O_TAMR) =
1440     (HWREG(WTIMER5_BASE+TIMER_O_TAMR) & ~TIMER_TAMR_TAAMS) |
1441     (TIMER_TAMR_TACDIR | TIMER_TAMR_TACMR | TIMER_TAMR_TAMR_CAP);
1442     // To set the event to rising edge, we need to modify the TAEVENT bits
1443     // in GPTMCTL. Rising edge = 00, so we clear the TAEVENT bits
1444     HWREG(WTIMER5_BASE+TIMER_O_CTL) &= ~TIMER_CTL_TAEVENT_M;
1445     // Now Set up the port to do the capture (clock was enabled earlier)
1446     // start by setting the alternate function for Port D bit 6 (WT5CCP0)
1447     HWREG(GPIO_PORTD_BASE+GPIO_O_AFSEL) |= BIT6HI;
1448     // Then, map bit 6's alternate function to WT5CCP0
1449     // 7 is the mux value to select WT0CCP0, 16 to shift it over to the
1450     // right nibble for bit 6 (4 bits/nibble * 6 bits)
1451     HWREG(GPIO_PORTD_BASE+GPIO_O_PCTL) =
1452     (HWREG(GPIO_PORTD_BASE+GPIO_O_PCTL) & 0xf0ffffff) + (7<<24);
1453     // Enable pin on Port D for digital I/O
1454     HWREG(GPIO_PORTD_BASE+GPIO_O_DEN) |= BIT6HI;
1455     // make pin 4 on Port D into an input
1456     HWREG(GPIO_PORTD_BASE+GPIO_O_DIR) &= BIT6LO;
1457     // back to the timer to enable a local capture interrupt
1458     HWREG(WTIMER5_BASE+TIMER_O_IMR) |= TIMER_IMR_CAEIM;
1459     // enable the Timer A in Wide Timer 0 interrupt in the NVIC
1460     // it is interrupt number 104 so appears in EN3 at bit 8
1461     HWREG(NVIC_EN3) |= BIT8HI;
1462     // make sure interrupts are enabled globally
1463     __enable_irq();
1464     // now kick the timer off by enabling it and enabling the timer to
1465     // stall while stopped by the debugger
1466     HWREG(WTIMER5_BASE+TIMER_O_CTL) |= (TIMER_CTL_TAEN | TIMER_CTL_TASTALL);
1467 }
1468
1469
1470
1471 void InputCaptureResponse_Hall( void ){
1472     ES_Event EventToPost_InputCapture;
1473
1474     // start by clearing the source of the interrupt, the input capture event
1475     HWREG(WTIMER5_BASE+TIMER_O_ICR) = TIMER_ICR_CAECINT;
1476     //printf("In input capture ISR after clearing\n\r");
1477
1478     // now grab the captured value and calculate the period
1479     ThisCapture = HWREG(WTIMER5_BASE+TIMER_O_TAR);
1480     ThisPeriod = ThisCapture - LastCapture;
1481     // update LastCapture to prepare for the next edge

```

```

1482     LastCapture = ThisCapture;
1483     //update the FlagValidFreq and see if we get a code
1484
1485     measFreqCode=frequency_map(ThisPeriod);
1486     //see if the current measurement changed
1487     if (measFreqCode==prevMeasFreqCode){
1488         stableCounter=stableCounter+1; //if so, increment
1489     }
1490     else{
1491         stableCounter=0; //restart the count
1492     }
1493
1494     //if we have read enough times of the same measurement, we know it's stable
1495     if (stableCounter==numOfMeasurementsForStable){
1496         stableMeasFreqCode=measFreqCode;
1497         stableCounter=0; //restart the count
1498
1499         //only post frequency when we have a valid and stable frequency code
1500         if ((stableMeasFreqCode!=0xf0)&&(FlagFreqConsumed==1)){
1501             if (CurrentState==WAITING){
1502                 EventToPost_InputCapture.EventType=ARRIVED_AT_STAGING;
1503
1504             }
1505
1506             else if ((CurrentState==SENDING_TO_LOC_AT_STAGING)|| (CurrentState==RECEIVING_FROM_LOC_AT_STAGING)){
1507                 //ecide whether that's a code for 1st report or 2nd report
1508                 if (FlagSentOneReport==0){
1509                     FlagFreqConsumed=0;
1510                     EventToPost_InputCapture.EventType=SEND_FIRST_REPORT;
1511                     EventToPost_InputCapture.EventParam=stableMeasFreqCode;
1512                     PostLOCMasterSM( EventToPost_InputCapture);
1513                 }
1514                 else if ((FlagSentTwoReport==0)&&(FlagFirstReportACK==1)){
1515                     FlagFreqConsumed=0;
1516                     EventToPost_InputCapture.EventType=SEND_SECOND_REPORT;
1517                     EventToPost_InputCapture.EventParam=stableMeasFreqCode;
1518                     PostLOCMasterSM( EventToPost_InputCapture);
1519                 }
1520             }
1521         }
1522     }
1523     //don't forget the update prevMeasFreqCode
1524     prevMeasFreqCode=measFreqCode;
1525
1526 } //end input capture response
1527
1528
1529
1530 uint8_t frequency_map(uint32_t period){
1531     //input is in encoder ticks
1532     uint8_t result = 0xf0; //assume we have no valid frequency to start with
1533     //the input would be in ticks, 4*10^7 ticks in one sec, period table is in micro seconds,
1534     //4*10^7 ticks in one sec is 4*10^7 ticks in 10^6 micro
1535     period=period/TicksToMicroSecDivisor;
1536
1537     for (int i = 0; i < period_table_length; i++){
1538         if ((period > PERIOD_TABLE[i]-FreqErrorThreshold) &
1539             (period < PERIOD_TABLE[i] + FreqErrorThreshold)){
1540             result = (uint8_t)i; //the code corresponds to the index
1541             break;
1542         }
1543     }
1544
1545     return result;
1546 }
1547
1548 //-----functions used to interact with other modules-----
1549 uint32_t queryStatusBytes(void){
1550     return StatusBytes;
1551 }
1552
1553 uint8_t queryActiveStagingGreen(void){

```



```

1554 //returns 0 for no active staging (either game hasn't started or in shooting), 1, 2, 3 for 1R,2R,3R
1555 //5 means error
1556 uint8_t ReturnResult=0;
1557 if ((StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)==0){
1558     ReturnResult=0; //the game hasn't started, you can't do anything
1559 }
1560 else if ((StatusBytes & GAME_STATUS_CSG_MASK)>0){//it's shooting, should not be staging
1561     ReturnResult=INVALID_LOCATIONS_ACTIVE;
1562 }
1563 else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_NONE_MASK){
1564     ReturnResult=0;
1565 }
1566 else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_ONE_MASK){
1567     ReturnResult=1;
1568 }
1569 else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_TWO_MASK){
1570     ReturnResult=2;
1571 }
1572 else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_THREE_MASK){
1573     ReturnResult=3;
1574 }
1575 else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)>=GAME_STATUS_GREEN_LOC_ALL_MASK){
1576     ReturnResult=INVALID_LOCATIONS_ACTIVE; //you cannot have all staging active for check-ins
1577 }
1578 return ReturnResult;
1579 }
1580 }
1581
1582 uint8_t queryActiveStagingRed(void){
1583     //returns 0 for no active staging (either game hasn't started or in shooting), 1, 2, 3 for 1R,2R,3R
1584     uint8_t ReturnResult=0;
1585     if ((StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)==0){
1586         ReturnResult=0; //the game hasn't started, you can't do anything
1587     }
1588     else if ((StatusBytes & GAME_STATUS_CSG_MASK)>0){//it's shooting, should not be staging
1589         ReturnResult=INVALID_LOCATIONS_ACTIVE;
1590     }
1591     else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_NONE_MASK){
1592         ReturnResult=0;
1593     }
1594     else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_ONE_MASK){
1595         ReturnResult=1;
1596     }
1597     else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_TWO_MASK){
1598         ReturnResult=2;
1599     }
1600     else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_THREE_MASK){
1601         ReturnResult=3;
1602     }
1603     else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)>=GAME_STATUS_RED_LOC_ALL_MASK){
1604         ReturnResult=INVALID_LOCATIONS_ACTIVE;
1605     }
1606     return ReturnResult;
1607 }
1608 }
1609
1610 uint8_t queryActiveShootingGreen(void){
1611     //returns 0 for no active shooting (either game hasn't started or still checking in), 1, 2, 3 for
1612     1R,2R,3R
1613     //4 means all goals are active, 5 means error
1614     uint8_t ReturnResult=0;
1615     if ((StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)==0){
1616         ReturnResult=0; //the game hasn't started, you can't do anything
1617     }
1618     else if ((StatusBytes & GAME_STATUS_CSG_MASK)==0){//it's checking in, should not be shooting
1619         ReturnResult=INVALID_LOCATIONS_ACTIVE;
1620     }
1621     else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_NONE_MASK){
1622         ReturnResult=0;
1623     }
1624     else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_ONE_MASK){

```

```

1625     ReturnResult=1;
1626 }
1627 else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_TWO_MASK){
1628     ReturnResult=2;
1629 }
1630 else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)==GAME_STATUS_GREEN_LOC_THREE_MASK){
1631     ReturnResult=3;
1632 }
1633 else if((StatusBytes & GAME_STATUS_GREEN_ACTIVE_STATUS_MASK)>=GAME_STATUS_GREEN_LOC_ALL_MASK){
1634     printf("Last 18 seconds!!, return shooting stage 4\r\n");
1635     ReturnResult=4;//you cannot have all staging active for check-ins
1636 }
1637 return ReturnResult;
1638 }
1639 }
1640
1641 uint8_t queryActiveShootingRed(void){
1642     //returns 0 for no active shooting (either game hasn't started or still checking in), 1, 2, 3 for
1643     1R,2R,3R
1644     //4 means all goals are active, 5 means error
1645     uint8_t ReturnResult=0;
1646     if ((StatusBytes & GAME_STATUS_SB3_GS_EXTRACT_MASK)==0){
1647         ReturnResult=0; //the game hasn't started, you can't do anything
1648     }
1649     else if ((StatusBytes & GAME_STATUS_CSR_MASK)==0){//it's checking in, should not be shooting
1650         ReturnResult=INVALID_LOCATIONS_ACTIVE;
1651     }
1652     else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_NONE_MASK){
1653         ReturnResult=0;
1654     }
1655     else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_ONE_MASK){
1656         ReturnResult=1;
1657     }
1658     else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_TWO_MASK){
1659         ReturnResult=2;
1660     }
1661     else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)==GAME_STATUS_RED_LOC_THREE_MASK){
1662         ReturnResult=3;
1663     }
1664     else if((StatusBytes & GAME_STATUS_RED_ACTIVE_STATUS_MASK)>=GAME_STATUS_RED_LOC_ALL_MASK){
1665         ReturnResult=4;//you cannot have all staging active for check-ins
1666     }
1667     return ReturnResult;
1668 }
1669
1670 uint8_t quickQueryActiveLocation(void){
1671     //instead of waiting for the GAME_STATUS to update, just grab it from the response
1672     return ActiveLocation;
1673 }
1674
1675 uint8_t queryGoalGreen(void){
1676     return (StatusBytes & GOAL_SCORE_GREEN_MASK) >> GOAL_SCORE_GREEN_OFFSET;
1677 }
1678
1679 uint8_t queryGoalRed(void){
1680     return StatusBytes & GOAL_SCORE_RED_MASK;
1681 }
1682
1683 void check_active_event(){
1684     if(get_Team() == GREEN){
1685         // Check Staging
1686         printf("Enter check active green\r\n");
1687         ActiveGreenStagingLocation=queryActiveStagingGreen();
1688         //printf("Before the posting if, ActiveGreenStagingLocation is %d\r\n",ActiveGreenStagingLocation);
1689         if ((ActiveGreenStagingLocation!=0)&&(ActiveGreenStagingLocation!=INVALID_LOCATIONS_ACTIVE)){
1690             //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE
1691             //it's a valid location at a valid, and it's different from before
1692             EventToPost.EventType=STAGE_ACTIVE;
1693             EventToPost.EventParam=ActiveGreenStagingLocation;
1694             PostMasterVehicleSM(EventToPost);
1695             printf("\r\nActive Green Staging Location is: %d",ActiveGreenStagingLocation);

```

```
1696     }
1697     prevActiveGreenStagingLocation=ActiveGreenStagingLocation;
1698
1699     // Check shooting
1700     ActiveGreenShootingLocation=queryActiveShootingGreen();
1701     //printf("Before the posting if, ActiveGreenShootingLocation is %d\n\r",ActiveGreenShootingLocation);
1702     if ((ActiveGreenShootingLocation!=0)&&(ActiveGreenShootingLocation!=INVALID_LOCATIONS_ACTIVE)){
1703         //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE
1704         //it's a valid location at a valid, and it's different from before
1705         EventToPost.EventType=SHOOT_ACTIVE;
1706         EventToPost.EventParam=ActiveGreenShootingLocation;
1707         PostMasterVehicleSM(EventToPost);
1708         printf("\r\nActive Green Shooting Location is: %d",ActiveGreenShootingLocation);
1709     }
1710     prevActiveGreenShootingLocation=ActiveGreenShootingLocation;
1711 }
1712 //endif
1713
1714 //ifdef TEAM_RED
1715 if(get_Team() == RED){
1716     // Check staging
1717     printf("Enter check active red\r\n");
1718     ActiveRedLocation=queryActiveStagingRed();
1719     if
1720 ((ActiveRedLocation!=0)&&(ActiveRedLocation!=INVALID_LOCATIONS_ACTIVE)&&(ActiveRedLocation!=prevActiveRed
Location)){
1721         //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE
1722         //it's a valid location at a valid, and it's different from before
1723         EventToPost.EventType=STAGE_ACTIVE;
1724         EventToPost.EventParam=ActiveRedLocation;
1725         PostMasterVehicleSM(EventToPost);
1726         printf("\r\nActive Red stagingLocation is: %d",ActiveRedLocation);
1727     }
1728     prevActiveRedLocation=ActiveRedLocation;
1729
1730     // Check shooting
1731     ActiveRedLocation=queryActiveShootingRed();
1732     if
1733 ((ActiveRedLocation!=0)&&(ActiveRedLocation!=INVALID_LOCATIONS_ACTIVE)&&(ActiveRedLocation!=prevActiveRed
Location)){
1734         //ASSUMPTION: ASSUMING STAGING AREAS ALWAYS CHANGE
1735         //it's a valid location at a valid, and it's different from before
1736         EventToPost.EventType=SHOOT_ACTIVE;
1737         EventToPost.EventParam=ActiveRedLocation;
1738         PostMasterVehicleSM(EventToPost);
1739         printf("\r\nActive Red Shooting Location is: %d",ActiveRedLocation);
1740     }
1741 }
1742 }
1743
1744
```