```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Boat Status Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#include "BoatStatus.h"
#include "defs.h"

/* Variables */
	unsigned char waterloggedFlag;
	unsigned int numberOfCrabsOnboard;
	unsigned char myFleet;
	unsigned char XOR_Cypher;
	unsigned char lastImportantEvent;
	unsigned char isItOpenSeason;
	unsigned char lastCrabPotAccumulationRate;
	unsigned char lastCrabPotOrDockNumber;
	unsigned char lastNumberOfCrabsTransfered;
	unsigned char boatState;
	unsigned int numberOfCrabsUnloadedToDock;
	unsigned int longTimerTargetValue;
	unsigned char areWeTimingSomething;
/* Prototypes */

/* Functions */

//this will set the boat status to 0's
void BOAT_STATUS_initalizeBoatStatus(void) {
	waterloggedFlag = 0;  //we aren't waterlogged
	numberOfCrabsOnboard = 0;
	myFleet = WHITE_FLEET;
	XOR_Cypher = 0x00;
	isItOpenSeason = 0;
	lastCrabPotAccumulationRate = 0;
	lastCrabPotOrDockNumber = 0;
	lastNumberOfCrabsTransfered = 0;
	numberOfCrabsUnloadedToDock = 0;
	longTimerTargetValue = 0;
	areWeTimingSomething = 0;
	lastImportantEvent = IMPT_EVNT_NO_EVENT;
	boatState = BOAT_STATE_WAITING_FOR_TEAM_ASSIGNMENT;
}

unsigned char BOAT_STATUS_getWaterloggedFlag(void) {
	return waterloggedFlag;
}
```

```
unsigned int BOAT_STATUS_NumberOfCrabsOnboard(void) {
	return numberOfCrabsOnboard;
}

unsigned char BOAT_STATUS_getLastCrabPotAccumulationRate(void) {
	return lastCrabPotAccumulationRate;
}

unsigned char BOAT_STATUS_getLastCrabPotOrDockNumber(void) {
	return lastCrabPotOrDockNumber;
}

unsigned char BOAT_STATUS_getLastNumberOfCrabsTransfered(void) {
	return lastNumberOfCrabsTransfered;
}

unsigned char BOAT_STATUS_getXOR_Cypher(void) {
	return XOR_Cypher;
}

unsigned char BOAT_STATUS_getIsItOpenSeason(void) {
	return isItOpenSeason;
}

unsigned char BOAT_STATUS_getMyFleet(void) {
	return myFleet;
}

unsigned char BOAT_STATUS_getState(void) {
	return boatState;
}

unsigned char BOAT_STATUS_getLastImportantEvent(void) {
	return lastImportantEvent;
}

unsigned int BOAT_STATUS_getLongTimerTargetValue(void) {
	return		longTimerTargetValue;
}

unsigned char BOAT_STATUS_getAreWeTimingSomething(void) {
	return		areWeTimingSomething;
}

//this will assign the value of the boat's waterlogged flag
void BOAT_STATUS_assignStateOfWaterloggedFlag(unsigned char flagState) {
	 waterloggedFlag = flagState;
}

//this will assign the value of the boat's fleet
void BOAT_STATUS_assignFleet(unsigned char fleet) {
	 myFleet = fleet;
}

//this will assign the value of the xor cypher
void BOAT_STATUS_assignXORCypher(unsigned char cypher) {
```

```
        XOR_Cypher = cypher;
}

//this will assign the value of the crab count
void BOAT_STATUS_assignNumberOfCrabsOnboard(unsigned int count) {
        if(count>MAX_NUMBER_OF_CRABS_ON_DCH) { //this is protection to make sure we don't have
more than 300 crabs on board (an maybe break the stepper motor flag)
         numberOfCrabsOnboard = MAX_NUMBER_OF_CRABS_ON_DCH;
        } else {
           numberOfCrabsOnboard = count;
        }
}

//this will assign the value of the crab count
void BOAT_STATUS_addToNumberOfCrabsOnboard(unsigned char crabs) {
        if((numberOfCrabsOnboard + crabs) > MAX_NUMBER_OF_CRABS_ON_DCH) {
           numberOfCrabsOnboard = MAX_NUMBER_OF_CRABS_ON_DCH;
        } else {
           numberOfCrabsOnboard += crabs;
        }
}

//this will assign the value of the open season flag
void BOAT_STATUS_assignOpenSeasonStatus(unsigned char flagState) {
        isItOpenSeason = flagState;
}

//this will assign the value of the lastCrabPotAccumulationRate
void BOAT_STATUS_assignLastCrabPotAccumulationRate(unsigned char crabs) {
        lastCrabPotAccumulationRate = crabs;
}

//this will assign the value of the lastCrabPotOrDockNumber
void BOAT_STATUS_assignLastCrabPotOrDockNumber(unsigned char potNumber) {
        lastCrabPotOrDockNumber = potNumber;
}

//this will assign the value of the open season flag
void BOAT_STATUS_assignLastNumberOfCrabsTransfered(unsigned char crabs) {
        lastNumberOfCrabsTransfered = crabs;
}

//this will assign the value of the open season flag
void BOAT_STATUS_assignBoatState(unsigned char state) {
        boatState = state;
}

//this will assign the value of the open season flag
void BOAT_STATUS_assignTotalCrabsUnloadedToDock(unsigned int crabs) {
        if((crabs) > MAX_NUMBER_OF_CRABS_ON_DCH) {
           numberOfCrabsUnloadedToDock = MAX_NUMBER_OF_CRABS_ON_DCH;
        } else {
           numberOfCrabsUnloadedToDock = crabs;
        }
}
```

```
//this will assign the value of the open season flag
void BOAT_STATUS_addToTheNumberOfCrabsUnloadedToDock(unsigned int crabs) {
        numberOfCrabsUnloadedToDock += crabs;
}

//this will assign the value of the timer target value
void BOAT_STATUS_assignLongTimerTargetValue(unsigned int targetTime) {
        longTimerTargetValue = targetTime;
}

//this will assign the value of the timingSomething flag
void BOAT_STATUS_assignAreWeTimingSomething(unsigned char flagState) {
        areWeTimingSomething = flagState;
}

//this will assign the value of the timingSomething flag
void BOAT_STATUS_assignLastImportantEvent(unsigned char theEvent) {
        lastImportantEvent = theEvent;
}
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Boat Status Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#ifndef _BOAT_STATUS_H_
#define _BOAT_STATUS_H_

//Public Functions
//this will set the boat status to 0's and the team to white
void BOAT_STATUS_initalizeBoatStatus(void);
unsigned char BOAT_STATUS_getWaterloggedFlag(void);
unsigned int BOAT_STATUS_NumberOfCrabsOnboard(void);
unsigned char BOAT_STATUS_getWaterloggedFlag(void);
unsigned char BOAT_STATUS_getLastCrabPotAccumulationRate(void);
unsigned char BOAT_STATUS_getLastCrabPotOrDockNumber(void);
unsigned char BOAT_STATUS_getLastNumberOfCrabsTransfered(void);
unsigned char BOAT_STATUS_getXOR_Cypher(void);
unsigned char BOAT_STATUS_getIsItOpenSeason(void);
unsigned char BOAT_STATUS_getMyFleet(void);
unsigned char BOAT_STATUS_getState(void);
unsigned char BOAT_STATUS_getLastImportantEvent(void);
unsigned int BOAT_STATUS_getLongTimerTargetValue(void);
unsigned char BOAT_STATUS_getAreWeTimingSomething(void);

void BOAT_STATUS_assignStateOfWaterloggedFlag(unsigned char flagState);
void BOAT_STATUS_assignFleet(unsigned char fleet);
void BOAT_STATUS_assignXORCypher(unsigned char cypher);
void BOAT_STATUS_assignNumberOfCrabsOnboard(unsigned int count);
void BOAT_STATUS_addToNumberOfCrabsOnboard(unsigned char crabs);
void BOAT_STATUS_assignOpenSeasonStatus(unsigned char flagState);
void BOAT_STATUS_assignLastCrabPotAccumulationRate(unsigned char crabs);
void BOAT_STATUS_assignLastCrabPotOrDockNumber(unsigned char potNumber);
void BOAT_STATUS_assignLastNumberOfCrabsTransfered(unsigned char crabs);
void BOAT_STATUS_assignBoatState(unsigned char state);
void BOAT_STATUS_assignTotalCrabsUnloadedToDock(unsigned int crabs);
void BOAT_STATUS_addToTheNumberOfCrabsUnloadedToDock(unsigned int crabs);
void BOAT_STATUS_assignLongTimerTargetValue(unsigned int targetTime);
void BOAT_STATUS_assignAreWeTimingSomething(unsigned char flagState);
void BOAT_STATUS_assignLastImportantEvent(unsigned char theEvent);

#endif
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Definitions
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#ifndef _DEFS_H_
#define _DEFS_H_

#include "BITDEFS.H" // BIT0HI, BIT0LO....definitions
#include <htc.h>

/* Port A bits: */
#define RFID_SS_CONTROL_BIT 0
#define RFID_SS_CONTROL_DDR TRISA
#define RFID_SS_CONTROL_PORT PORTA

#define SC_SS_CONTROL_BIT 1
#define SC_SS_CONTROL_DDR TRISA
#define SC_SS_CONTROL_PORT PORTA

#define SERVO_OUTPUT_BIT 2
#define SERVO_OUTPUT_DDR TRISA
#define SERVO_OUTPUT_PORT PORTA

/* Port B bits: */
#define SDI_BIT 4
#define SDI_DDR TRISB
#define SDI_PORT PORTB

#define SCK_BIT 6
#define SCK_DDR TRISB
#define SCK_PORT PORTB


#define ASYNC_TX_BIT 7
#define ASYNC_TX_DDR TRISB
#define ASYNC_TX_PORT PORTB

#define ASYNC_RX_BIT 5
#define ASYNC_RX_DDR TRISB
#define ASYNC_RX_PORT PORTB

/*Port C bits*/
#define CollectRFIDDataTrigger_BIT 0
#define CollectRFIDDataTrigger_DDR TRISC
#define CollectRFIDDataTrigger_PORT PORTC
```

```
#define  CollectSCDataTrigger_BIT 1
#define  CollectSCDataTrigger_DDR TRISC
#define  CollectSCDataTrigger_PORT PORTC

#define WATERLOGGED_INPUT_FROM_SC_BIT 2
#define WATERLOGGED_INPUT_FROM_SC_DDR TRISC
#define WATERLOGGED_INPUT_FROM_SC_PORT PORTC

#define STEPPER_INPUT_1_BIT 3
#define STEPPER_INPUT_1_DDR TRISC
#define STEPPER_INPUT_1_PORT PORTC

#define STEPPER_INPUT_2_BIT 4
#define STEPPER_INPUT_2_DDR TRISC
#define STEPPER_INPUT_2_PORT PORTC

#define STEPPER_INPUT_3_BIT 5
#define STEPPER_INPUT_3_DDR TRISC
#define STEPPER_INPUT_3_PORT PORTC

#define STEPPER_INPUT_4_BIT 6
#define STEPPER_INPUT_4_DDR TRISC
#define STEPPER_INPUT_4_PORT PORTC

#define SDO_BIT 7
#define SDO_DDR TRISC
#define SDO_PORT PORTC


/* Convenience */
#define CRAB_POT_0 0xA0
#define CRAB_POT_1 0xA1
#define CRAB_POT_2 0xA2
#define CRAB_POT_3 0xA3
#define CRAB_POT_4 0xA4
#define CRAB_POT_5 0xA5
#define CRAB_POT_6 0xA6
#define CRAB_POT_7 0xA7
#define CRAB_POT_8 0xA8
#define CRAB_POT_9 0xA9
#define CRAB_POT_10 0xAA
#define CRAB_POT_11 0xAB
#define CRAB_POT_12 0xAC
#define LOADING_DOCK 0xAD
#define DRY_DOCK 0xAE
#define RED_TEAM_CARD 0x22
#define GREEN_TEAM_CARD 0x44
#define WHITE_TEAM_CARD 0x11
#define NO_MATCH 0x55    //ANDI ADDED THIS TO THE BIGGER LIST

#define RED_FLEET 0x11
#define GREEN_FLEET  0x22
#define WHITE_FLEET 0x44

#define FLEET_LEADER_TIMEOUT_TIME 0x0100
```

```
#define BOAT_STATE_WAITING_FOR_TEAM_ASSIGNMENT 0
#define BOAT_STATE_PLAYING_GAME 1
#define BOAT_STATE_TEAM_LEADER 2
#define BOAT_STATE_NOT_TEAM_LEADER 3

#define IMPT_EVNT_NO_EVENT 0x00
#define IMPT_EVNT_WHITE_TEAM_ASSIGNED 0x01
#define IMPT_EVNT_RED_LEADER 0x02
#define IMPT_EVNT_RED_NOT_LEADER 0x03
#define IMPT_EVNT_GREEN_LEADER 0x04
#define IMPT_EVNT_GREEN_NOT_LEADER 0x05
#define IMPT_EVNT_DRY_DOCK_CARD_FIXED 0x06
#define IMPT_EVNT_UNLOADING_DOCK_UNLOADED 0x07
#define IMPT_EVNT_CRAB_POT_HAULLED 0x08
#define IMPT_EVNT_DCH_WATERLOGGED 0x09
#define IMPT_EVNT_DCH_UNWATERLOGGED 0x0A
#define IMPT_EVNT_OPEN_SEASON 0x0B

#define MAX_NUMBER_OF_CRABS_ON_DCH 300

#define SOS_FLAG_SERVO_UP_HIGH_VALUE 5
#define SOS_FLAG_SERVO_UP_LOW_VALUE 0

#define SOS_FLAG_SERVO_DOWN_HIGH_VALUE 2
#define SOS_FLAG_SERVO_DOWN_LOW_VALUE 0

#define FALSE 0
#define TRUE 1

#define LOW 0
#define HIGH 1

#define LEFT 0
#define RIGHT 1

#define CLOCKWISE 0
#define COUNTERCLOCKWISE 1

#define SETBIT(ADDRESS,BIT) (ADDRESS |= (1<<BIT)) //set bit (high)      //output
#define CLEARBIT(ADDRESS,BIT) (ADDRESS &= ~(1<<BIT)) //clear bit (low)  //input
#define CHECKBIT(ADDRESS,BIT) (ADDRESS & (1<<BIT)) //test single bit in I/O location
#define TOGGLEBIT(ADDRESS,BIT) (ADDRESS ^= (1<<BIT)) //toggle single bit

void MAIN_resetLongTimer(void);
unsigned int MAIN_getLongTimerCount(void);
void MAIN_IncrementLongTimer(void);
void MAIN_StopLongTimer(void);

/*This Function Initalizes the Serial Data functionality*/
void initSSPasMaster(void);
void BOAT_SendMessage(unsigned char Message);
void init_BOAT_SDO_Array(void);
void CondenseHexDataToSend(void);
void BOAT_SetupTimer1(void);
//void blockingDelay(unsigned int delayTimeH);
```

```
unsigned char MAIN_CreateXORCypher(void);
unsigned char MAIN_EncodeOrDecodeByteUsingCyper(unsigned char dataByte);

void MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(unsigned char flagState);
void MAIN_assignDoWeNeedToProccessANewXbeeMessageFlag(unsigned char flagState);
void MAIN_assignDoWeNeedToSendDataToNeptuneFlag(unsigned char flagState);
void MAIN_assignDoWeNeedToSendDataToFleetFlag(unsigned char flagState);
void MAIN_assignDoWeNeedToSendCypherToTeamMateFlag(unsigned char flagState);
void MAIN_assignDoWeNeedToSendTeamIdentificationFlag(unsigned char flagState);
void MAIN_assignDoWeNeedToGetRFIDDataFromPicFlag(unsigned char flagState);
void MAIN_assignDoWeNeedToGetDataFromSCFlag(unsigned char flagState);
void MAIN_assignDoWeNeedToSendRFIDDataToSCFlag(unsigned char flagState);
void MAIN_assignDoWeNeedToProccessANewXbeeMessageFlag(unsigned char flagState);
void MAIN_assignDoWeNeedToUpdateHCWithCypherFlag(unsigned char flagState);

#endif
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Events Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#include <htc.h>
#include "defs.h"
#include "events.h"
#include "xbee.h"
#include "PirateLCD.h"
#include "xbee.h"
#include "BoatStatus.h"
#include "ServoController.h"
#include "rfid.h"
#include "RFID_ReadAndCategorize.h"
#include "stepper.h"

//static unsigned char crabsTransfered;
//static unsigned char crabRate;
//unsigned char tempLastPotNumber;

unsigned char tempTime;

/* Functions */
static void EVENTS_runStateWaitingEventHandler(Event_t event);  //this will be a useful debugging state
static void EVENTS_runStatePlayingEventHandler(Event_t event);  //this is for when we are playing the
game
static void EVENTS_runStateLeaderEventHandler(Event_t event);
static void EVENTS_runStateNotLeaderEventHandler(Event_t event);

/* Checks and returns any events that have occurred. */
Event_t EVENTS_CheckForEvents(void) {
        Event_t event = NO_EVENT;

        if(XBEE_CheckNeptuneOpenSeasonFlag() != 0) {
            event = NEPTUNE_OPEN_SEASON_BROADCAST_EVENT;
            return event;
        }

        if(XBEE_HasAFullMessageBeenReceived() != 0) {
            event = XBEE_FULL_MESSAGE_RECEIVED_EVENT;
            return event;
        }

        if(XBEE_hasNeptuneResponded() != 0) {
            return EVENTS_whatDidNeptuneRespondWith();
        }
```

```
        if(XBEE_CheckNewDriveCommandFromHCFlag() != 0) {
            event = NEW_DRIVE_COMMAND_FROM_HC;
            return event;
        }

        if(CHECKBIT(WATERLOGGED_INPUT_FROM_SC_PORT,
WATERLOGGED_INPUT_FROM_SC_BIT) != 0  && BOAT_STATUS_getWaterloggedFlag() == 0) {
            //if HI = Waterlogged
            //we've had a water-logged event from the SC
            event = DCH_WATERLOGGED_EVENT;
            return event;
        }

        if(BOAT_STATUS_getAreWeTimingSomething() != 0) { //are we currently timing something?

        //BOAT_STATUS_assignNumberOfCrabsOnboard(BOAT_STATUS_getLongTimerTargetValue());
            //if we are, have we gotten to the target time?
            tempTime = BOAT_STATUS_getLongTimerTargetValue();

            if(BOAT_STATUS_getLongTimerTargetValue() < MAIN_getLongTimerCount()) {
                    //yes, we have reached the time that we were counting towards
                    event = LONG_TIMER_EXPIRED;
                    return event;
            }
        }

        if(XBEE_CheckHaveWeGottenANewXORCypher() != 0) { //we've gotten a XOR cypher from our
team
            event = TEAMMATE_GIVING_CYPHER;
            return event;
        }

        if(XBEE_CheckHaveWeHadATeamMateRequestFlag() != 0) { //we've had a request from our team
for XOR cypher
            event = TEAMMATE_REQUESTING_CYPHER;
            return event;
        }

        if(CHECKBIT(CollectRFIDDataTrigger_PORT, CollectRFIDDataTrigger_BIT) != 0 &&
RFID_CheckReceivingARFIDMessageFlag() == 0) {
            //if the trigger line from the RFID is high, and we aren't currently reciving an RFID message
            //then this is a new event from the RFID reader, and we need to do something about it
            event = RFID_PIC_SIGNAL_LINE_HIGH_EVENT;
            return event;
        }

        if(CHECKBIT(CollectSCDataTrigger_PORT, CollectSCDataTrigger_BIT) != 0 &&
RFID_CheckReceivingASCMessageFlag() == 0) {
            //if the trigger line from the SC is high, and we aren't currently reciving an SC message
            //then this is a new event from the SC, and we need to do something about it
            event = SC_PIC_SIGNAL_LINE_HIGH_EVENT;
            return event;
        }

        if(RFID_haveWeReceivedANewRFIDScanFlag() != 0) {
```

```
            //we just got data from the RFID pic!!! woot!
            event = NEW_RFID_SCAN_DATA;
            return event;
        }

        if(RFID_haveWeReceivedNewSCDataFlag() != 0) {
            //we just got data from the SC pic!!! woot!
            event = NEW_SC_DATA;
            return event;
        }

        //if (event != NO_EVENT)
            //return event;

        return event;
}

Event_t EVENTS_whatDidNeptuneRespondWith(void) {
        Event_t event = NO_EVENT;

        switch (XBEE_getNeptuneResponseType()) {
            case NEP_RESP_CRAB_POT_DATA:
                        event = NEPTUNE_CRAB_POT_DATA_RETURNED_EVENT;
                        break;
            case NEP_RESP_SHIP_UNLOADED:
                        event = NEPTUNE_DCH_UNLOADED_EVENT;
                        break;
            case NEP_RESP_SHIP_REPAIRED:
                        event = NEPTUNE_DCH_REPAIRED_EVENT;
                        break;
            case NEP_RESP_INVALID_KEY:
                        event = NEPTUNE_INVALID_KEY_EVENT;
                        break;
            default:
                        break;
        } //end switch
        return event;
}

void EVENTS_RunStateMachine(Event_t event) { //this is nesting things too deeply!!
        //eventually put states in here (in boat status)
        //if/switch/nested functions(but not too many!!)

        switch(BOAT_STATUS_getState()) {
            case BOAT_STATE_WAITING_FOR_TEAM_ASSIGNMENT:   //we are assigned to the
"WHITE FLEET"
                        EVENTS_runStateWaitingEventHandler(event);
                        break;
            case BOAT_STATE_PLAYING_GAME:  //we have gotten the open season command and are
now activily playing the game
                        EVENTS_runStatePlayingEventHandler(event);
                        break;
            case BOAT_STATE_TEAM_LEADER:  //we were swiped with a "RED" or "GREEN" team card,
sent out a team cypher request, but didn't get a responce within 1 second
                        EVENTS_runStateLeaderEventHandler(event);
                        break;
```

```
            case BOAT_STATE_NOT_TEAM_LEADER:  //we were swiped with a "RED" or "GREEN"
team card and are either waiting for a cypher responce or have already gotten one
                        EVENTS_runStateNotLeaderEventHandler(event);
                        break;
            default:
                        break;
        }
}

static void EVENTS_runStateWaitingEventHandler(Event_t event) { //this will be a useful debugging state
        switch(event) {
            case NO_EVENT:
                        break;
            case XBEE_FULL_MESSAGE_RECEIVED_EVENT:
                        //we've gotten a message over the radio, now parse that data
                        //XBEE_InterperateReceivedMessage();  //<-- this is causing a stack overflow, i'm
moving it
                        MAIN_assignDoWeNeedToProccessANewXbeeMessageFlag(1);
                        break;
            case RFID_PIC_SIGNAL_LINE_HIGH_EVENT:
                        //read in the RFID data from the rfid pic
                        //start receiving the rfid stuff from the rfid pic
                        MAIN_assignDoWeNeedToGetRFIDDataFromPicFlag(1);
                        break;
            case NEW_RFID_SCAN_DATA:
                        //now, do something with that new RFID scan data
                        //MAIN_assignDoWeNeedToSendRFIDDataToSCFlag(1);
                        //get the card type (if it is a crab pot or unloading dock do something with it)

        BOAT_STATUS_assignLastCrabPotOrDockNumber(RFID_WhatWasTheRFIDCard(RFID_getLast
S1(), RFID_getLastS2(), RFID_getLastS3()));
                        //SERVO_CONTROLLER_setSOSFlagDOWN();      //debug
                        switch (BOAT_STATUS_getLastCrabPotOrDockNumber()) {
                                    case NO_MATCH:
                                                //SERVO_CONTROLLER_setSOSFlagUP();   //debug
                                                break;
                                    case WHITE_TEAM_CARD:
                                                //if it is a team card, we need to do something else with it
                                                BOAT_STATUS_initalizeBoatStatus();
                                                STEPPER_SetCrabCountFlagHeight(0);

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_WHITE_TEAM_ASSIGNED);
                                                MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update
the crab count to HC
                                                SERVO_CONTROLLER_setSOSFlagDOWN();
                                                break;
                                    case GREEN_TEAM_CARD:
                                                //if it is a team card, we need to do something else with it
                                                //BOAT_STATUS_assignFleet(GREEN_FLEET);
                                                BOAT_STATUS_assignFleet(GREEN_FLEET);

        BOAT_STATUS_assignLastCrabPotOrDockNumber(GREEN_FLEET);
                                                 //FLEET_LEADER_TIMEOUT_TIME
                                                SERVO_CONTROLLER_setSOSFlagUP();

        BOAT_STATUS_assignLongTimerTargetValue(FLEET_LEADER_TIMEOUT_TIME);
```

```
                                    BOAT_STATUS_assignAreWeTimingSomething(1);
                                    //TMR1IE=0;
                                    MAIN_resetLongTimer();

        BOAT_STATUS_assignBoatState(BOAT_STATE_NOT_TEAM_LEADER);
                                    MAIN_assignDoWeNeedToSendTeamIdentificationFlag(1);
//broadcast a we are on team xx message

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_GREEN_NOT_LEADER);
                                    //set the timer target
                                    //FLEET_LEADER_TIMEOUT_TIME
                                    //set the time
                                    //send a message to the fleet
                                    //set the boat state
                                    //if the timer expires, we become team captain  which we have to
indicate to the fleet, use the SOS flag...
                                    //don't use the events below, use this instead
                                    STEPPER_SetCrabCountFlagHeight(100);  //debug
                                    MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update
the crab count to HC
                                    break;
                                case RED_TEAM_CARD:
                                //case CRAB_POT_9:
                                    //if it is a team card, we need to do something else with it
                                    BOAT_STATUS_assignFleet(RED_FLEET);
                                    BOAT_STATUS_assignLastCrabPotOrDockNumber(RED_FLEET);
                                     //FLEET_LEADER_TIMEOUT_TIME
                                    SERVO_CONTROLLER_setSOSFlagUP();

        BOAT_STATUS_assignLongTimerTargetValue(FLEET_LEADER_TIMEOUT_TIME);
                                    BOAT_STATUS_assignAreWeTimingSomething(1);
                                    //TMR1IE=0;
                                    MAIN_resetLongTimer();

        BOAT_STATUS_assignBoatState(BOAT_STATE_NOT_TEAM_LEADER);
                                    MAIN_assignDoWeNeedToSendTeamIdentificationFlag(1);
//broadcast a we are on team xx message

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_RED_NOT_LEADER);
                                    //set the timer target
                                    //FLEET_LEADER_TIMEOUT_TIME
                                    //set the time
                                    //send a message to the fleet
                                    //set the boat state
                                    //if the timer expires, we become team captain  which we have to
indicate to the fleet, use the SOS flag...
                                    //don't use the events below, use this instead
                                    STEPPER_SetCrabCountFlagHeight(100);  //debug
                                    MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update
the crab count to HC
                                    break;
                                default:  //it's a crab pot
                                    //this time we will send it to the SC pic
                                    BOAT_STATUS_assignLastImportantEvent(NO_EVENT);
                                    MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update
the crab count to HC
```

```
                                        //MAIN_assignDoWeNeedToSendRFIDDataToSCFlag(1);
                                break;
                        }
                        break;
                case NEW_SC_DATA:
                        //send a message to neptune
                        MAIN_assignDoWeNeedToSendDataToNeptuneFlag(1);
                        break;
                case SC_PIC_SIGNAL_LINE_HIGH_EVENT:
                        //read in the SC data from the SC pic
                        MAIN_assignDoWeNeedToGetDataFromSCFlag(1);
                        break;
                case NEPTUNE_OPEN_SEASON_BROADCAST_EVENT:
                        //do nothing with open season broadcast...(perhaps debugging)
                        //start playing the game (even though we don't have a team)
                        BOAT_STATUS_assignBoatState(BOAT_STATE_PLAYING_GAME);
                        SERVO_CONTROLLER_setSOSFlagDOWN();
                        BOAT_STATUS_assignLastCrabPotAccumulationRate(0);
                        BOAT_STATUS_assignLastNumberOfCrabsTransfered(0);
                        BOAT_STATUS_assignNumberOfCrabsOnboard(0);
                        BOAT_STATUS_assignStateOfWaterloggedFlag(0);
                        STEPPER_SetCrabCountFlagHeight(BOAT_STATUS_NumberOfCrabsOnboard());
                        BOAT_STATUS_assignOpenSeasonStatus(1);
                        BOAT_STATUS_assignLongTimerTargetValue(0);
                        BOAT_STATUS_assignAreWeTimingSomething(0);
                        BOAT_STATUS_assignLastCrabPotOrDockNumber(CRAB_POT_0);
                        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_OPEN_SEASON);
                        MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);
                        break;
                default:
                        break;
        }
}

static void EVENTS_runStatePlayingEventHandler(Event_t event) { //this is for when we are playing the
game
        switch (event) {
                case NO_EVENT:
                        break;
                case XBEE_FULL_MESSAGE_RECEIVED_EVENT:
                        //we've gotten a message over the radio, now parse that data
                        //XBEE_InterperateReceivedMessage();  //<-- this is causing a stack overflow, i'm
moving it
                        MAIN_assignDoWeNeedToProccessANewXbeeMessageFlag(1);
                        break;
                case RFID_PIC_SIGNAL_LINE_HIGH_EVENT:
                        //read in the RFID data from the rfid pic
                        //start receiving the rfid stuff from the rfid pic
                        MAIN_assignDoWeNeedToGetRFIDDataFromPicFlag(1);
                        break;
                case NEW_RFID_SCAN_DATA:
                        //now, do something with that new RFID scan data
                        //MAIN_assignDoWeNeedToSendRFIDDataToSCFlag(1);
                        //get the card type (if it is a crab pot or unloading dock do something with it)

        BOAT_STATUS_assignLastCrabPotOrDockNumber(RFID_WhatWasTheRFIDCard(RFID_getLast
```

```
S1(), RFID_getLastS2(), RFID_getLastS3()));
                    //SERVO_CONTROLLER_setSOSFlagDOWN();     //debug
                    switch (BOAT_STATUS_getLastCrabPotOrDockNumber()) {
                            case NO_MATCH:
                                    //SERVO_CONTROLLER_setSOSFlagUP();   //debug
                                    break;
                            case LOADING_DOCK:
                                    //this time we will send it to the SC pic
                                    MAIN_assignDoWeNeedToSendRFIDDataToSCFlag(1);
                                    //STEPPER_SetCrabCountFlagHeight(0);
                                    break;
                            case DRY_DOCK:
                                    //this time we will send it to the SC pic
                                    MAIN_assignDoWeNeedToSendRFIDDataToSCFlag(1);
                                    break;
                            case WHITE_TEAM_CARD:
                                    //if it is a team card, we need to do something else with it
                                    BOAT_STATUS_initalizeBoatStatus();
                                    STEPPER_SetCrabCountFlagHeight(0);

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_WHITE_TEAM_ASSIGNED);
                                    MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update
the crab count to HC
                                    SERVO_CONTROLLER_setSOSFlagDOWN();
                                    break;
                            case GREEN_TEAM_CARD:
                                    //if it is a team card, we need to do nothing
                                    break;
                            case RED_TEAM_CARD:
                                    //if it is a team card, we need to do nothing
                                    break;
                            default:   //it's a crab pot
                                    //this time we will send it to the SC pic
                                    //BOAT_STATUS_assignLastImportantEvent(NO_EVENT);
                                    //MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);
//update the crab count to HC
                                    MAIN_assignDoWeNeedToSendRFIDDataToSCFlag(1);
                                    break;
                    }
                    break;
            case LONG_TIMER_EXPIRED:
                    MAIN_StopLongTimer();
                    BOAT_STATUS_assignAreWeTimingSomething(0);
                    break;
            case NEW_SC_DATA:
                    //send a message to neptune
                    MAIN_assignDoWeNeedToSendDataToNeptuneFlag(1);
                    break;
            case SC_PIC_SIGNAL_LINE_HIGH_EVENT:
                    //read in the SC data from the SC pic
                    MAIN_assignDoWeNeedToGetDataFromSCFlag(1);
                    break;
            case NEPTUNE_OPEN_SEASON_BROADCAST_EVENT:
                    //do nothing with open season broadcast...(perhaps debugging)
                    //start playing the game (even though we don't have a team)
                    BOAT_STATUS_assignBoatState(BOAT_STATE_PLAYING_GAME);
```

```
                SERVO_CONTROLLER_setSOSFlagDOWN();
                BOAT_STATUS_assignLastCrabPotAccumulationRate(0);
                BOAT_STATUS_assignLastNumberOfCrabsTransfered(0);
                BOAT_STATUS_assignNumberOfCrabsOnboard(0);
                BOAT_STATUS_assignStateOfWaterloggedFlag(0);
                STEPPER_SetCrabCountFlagHeight(BOAT_STATUS_NumberOfCrabsOnboard());
                BOAT_STATUS_assignOpenSeasonStatus(1);
                BOAT_STATUS_assignLongTimerTargetValue(0);
                BOAT_STATUS_assignAreWeTimingSomething(0);
                BOAT_STATUS_assignLastCrabPotOrDockNumber(CRAB_POT_0);
                BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_OPEN_SEASON);
                MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);
                break;
        case NEPTUNE_CRAB_POT_DATA_RETURNED_EVENT:
                //we have gathered some crab units from a pot

    BOAT_STATUS_assignLastCrabPotAccumulationRate(XBEE_getLastCrabPotAccumulationRate())
;

    BOAT_STATUS_assignLastNumberOfCrabsTransfered(XBEE_getLastAmountOfCrabTransfered())
;

    BOAT_STATUS_addToNumberOfCrabsOnboard(BOAT_STATUS_getLastNumberOfCrabsTransfe
red());  //<--this is nested too deeply, and is causing a stack overflow

                //STEPPER_SetCrabCountFlagHeight(BOAT_STATUS_NumberOfCrabsOnboard());
                STEPPER_SetCrabCountFlagHeight(BOAT_STATUS_NumberOfCrabsOnboard());
                BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_CRAB_POT_HAULLED);

                //tempLastPotNumber = RFID_WhatWasTheRFIDCard(RFID_getLastS1(),
RFID_getLastS2(), RFID_getLastS3());
                //BOAT_STATUS_assignLastCrabPotOrDockNumber(tempLastPotNumber);
                //THIS WILL UPDATE THE CRAB AMOUNT STEPPER MOTOR
                //STEPPER_ResetFlagHeight(300);
                //setFlagHeight(BOAT_STATUS_NumberOfCrabsOnboard())
                //Broadcast it to the fleet!!!!
                //....do that here....
                MAIN_assignDoWeNeedToSendDataToFleetFlag(1);
                MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update the crab count to
HC
                break;
        case NEPTUNE_DCH_UNLOADED_EVENT:
                //update the number of crabs that we have given to the dock

    BOAT_STATUS_addToTheNumberOfCrabsUnloadedToDock(BOAT_STATUS_NumberOfCrabsO
nboard());
                //reset the current number of crabs on the boat to zero
                BOAT_STATUS_assignNumberOfCrabsOnboard(0);

    BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_UNLOADING_DOCK_UNLOADED);
                STEPPER_SetCrabCountFlagHeight(0);
                STEPPER_SetCrabCountFlagHeight(BOAT_STATUS_NumberOfCrabsOnboard());
                MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update the crab count to
HC
                break;
        case NEPTUNE_DCH_REPAIRED_EVENT:
```

```
                //we've been repared
                //set the state of the boat's waterlogged flag to zero
                BOAT_STATUS_assignStateOfWaterloggedFlag(0);

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_DRY_DOCK_CARD_FIXED);
                //actually set the physical waterlogged flag to lowered

        //SERVO_CONTROLLER_setServoPosition(SERVO_POSITION_SOS_FLAG_DOWN);
                SERVO_CONTROLLER_setSOSFlagDOWN();
                //set the current crab count to zero just to make sure it happened
                BOAT_STATUS_assignNumberOfCrabsOnboard(0);
                STEPPER_SetCrabCountFlagHeight(BOAT_STATUS_NumberOfCrabsOnboard());
                MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update the crab count to
HC
                break;
            case NEPTUNE_INVALID_KEY_EVENT:
                //I don't think that neptune actually transmits this, ever.
                break;
            case DCH_WATERLOGGED_EVENT:
                //waterlogged event
                //set the boat status (software, not hardware) flag to let everyone know we're waterlogged
                //SERVO_CONTROLLER_setServoPosition(SERVO_POSITION_SOS_FLAG_UP);
                SERVO_CONTROLLER_setSOSFlagUP();
                //set the state of the boat's software waterlogged flag to high
                BOAT_STATUS_assignStateOfWaterloggedFlag(1);
                //reset the current number of crabs on the boat to zero
                BOAT_STATUS_assignNumberOfCrabsOnboard(0);
                STEPPER_SetCrabCountFlagHeight(BOAT_STATUS_NumberOfCrabsOnboard());
                //send an update to the HC
                BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_DCH_WATERLOGGED);
                MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update the crab count to
HC
                break;
            case NEW_DRIVE_COMMAND_FROM_HC:
                break;
            default:
                break;
        }
}

static void EVENTS_runStateLeaderEventHandler(Event_t event) {
        switch(event) {
            case NO_EVENT:
                break;
            case XBEE_FULL_MESSAGE_RECEIVED_EVENT:
                //we've gotten a message over the radio, now parse that data
                //XBEE_InterperateReceivedMessage();  //<-- this is causing a stack overflow, i'm
moving it
                MAIN_assignDoWeNeedToProccessANewXbeeMessageFlag(1);
                break;
            case RFID_PIC_SIGNAL_LINE_HIGH_EVENT:
                //read in the RFID data from the rfid pic
                //start receiving the rfid stuff from the rfid pic
                MAIN_assignDoWeNeedToGetRFIDDataFromPicFlag(1);
                break;
            case NEW_RFID_SCAN_DATA:
```

```
                    //now, do something with that new RFID scan data
                    //MAIN_assignDoWeNeedToSendRFIDDataToSCFlag(1);
                    //get the card type (if it is a crab pot or unloading dock do something with it)

        BOAT_STATUS_assignLastCrabPotOrDockNumber(RFID_WhatWasTheRFIDCard(RFID_getLast
S1(), RFID_getLastS2(), RFID_getLastS3()));
                    //SERVO_CONTROLLER_setSOSFlagDOWN();     //debug
                    switch (BOAT_STATUS_getLastCrabPotOrDockNumber()) {
                              case NO_MATCH:
                                        //SERVO_CONTROLLER_setSOSFlagUP();   //debug
                                        break;
                              case WHITE_TEAM_CARD:
                                        //if it is a team card, we need to do something else with it
                                        BOAT_STATUS_initalizeBoatStatus();
                                        STEPPER_SetCrabCountFlagHeight(0);  //debug

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_WHITE_TEAM_ASSIGNED);
                                        MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update
the crab count to HC
                                        SERVO_CONTROLLER_setSOSFlagDOWN();
                                        break;
                              case GREEN_TEAM_CARD:
                                        //if it is a team card, we need to do something else with it
                                        //BOAT_STATUS_assignFleet(GREEN_FLEET);
                                        BOAT_STATUS_assignFleet(GREEN_FLEET);

        BOAT_STATUS_assignLastCrabPotOrDockNumber(GREEN_FLEET);
                                         //FLEET_LEADER_TIMEOUT_TIME
                                        SERVO_CONTROLLER_setSOSFlagUP();

        BOAT_STATUS_assignLongTimerTargetValue(FLEET_LEADER_TIMEOUT_TIME);
                                        BOAT_STATUS_assignAreWeTimingSomething(1);
                                        //TMR1IE=0;
                                        MAIN_resetLongTimer();

        BOAT_STATUS_assignBoatState(BOAT_STATE_NOT_TEAM_LEADER);
                                        MAIN_assignDoWeNeedToSendTeamIdentificationFlag(1);
//broadcast a we are on team xx message

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_GREEN_NOT_LEADER);
                                        //set the timer target
                                        //FLEET_LEADER_TIMEOUT_TIME
                                        //set the time
                                        //send a message to the fleet
                                        //set the boat state
                                        //if the timer expires, we become team captain  which we have to
indicate to the fleet, use the SOS flag...
                                        //don't use the events below, use this instead
                                        STEPPER_SetCrabCountFlagHeight(100);  //debug
                                        MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update
the crab count to HC
                                        break;
                              case RED_TEAM_CARD:
                              //case CRAB_POT_9:
                                        //if it is a team card, we need to do something else with it
                                        BOAT_STATUS_assignFleet(RED_FLEET);
```

```
                                    BOAT_STATUS_assignLastCrabPotOrDockNumber(RED_FLEET);
                                     //FLEET_LEADER_TIMEOUT_TIME
                                    SERVO_CONTROLLER_setSOSFlagUP();

        BOAT_STATUS_assignLongTimerTargetValue(FLEET_LEADER_TIMEOUT_TIME);
                                    BOAT_STATUS_assignAreWeTimingSomething(1);
                                    //TMR1IE=0;
                                    MAIN_resetLongTimer();

        BOAT_STATUS_assignBoatState(BOAT_STATE_NOT_TEAM_LEADER);
                                    MAIN_assignDoWeNeedToSendTeamIdentificationFlag(1);
//broadcast a we are on team xx message

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_RED_NOT_LEADER);
                                    //set the timer target
                                    //FLEET_LEADER_TIMEOUT_TIME
                                    //set the time
                                    //send a message to the fleet
                                    //set the boat state
                                    //if the timer expires, we become team captain  which we have to
indicate to the fleet, use the SOS flag...
                                    //don't use the events below, use this instead
                                    STEPPER_SetCrabCountFlagHeight(100);  //debu
                                    MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update
the crab count to HC
                                    break;
                                default:  //it's a crab pot
                                    //this time we will send it to the SC pic
                                    //MAIN_assignDoWeNeedToSendRFIDDataToSCFlag(1);

                                    //send fake crab pot data to teammates
                                    BOAT_STATUS_assignLastCrabPotOrDockNumber(CRAB_POT_0);

        BOAT_STATUS_assignLastCrabPotAccumulationRate(RFID_getLastS3());

                                    MAIN_assignDoWeNeedToSendDataToFleetFlag(1);
                                break;
                }
                break;
            case TEAMMATE_GIVING_CYPHER:
                //stop timer
                //MAIN_StopLongTimer();
                //BOAT_STATUS_assignAreWeTimingSomething(0);
                //store cypher
                BOAT_STATUS_assignXORCypher(XBEE_getNewXORCypher());
                BOAT_STATUS_assignNumberOfCrabsOnboard(XBEE_getNewXORCypher());
                BOAT_STATUS_assignBoatState(BOAT_STATE_NOT_TEAM_LEADER);
                //let hc know what the cypher is   ...do this....
                if(BOAT_STATUS_getMyFleet() == RED_FLEET) {

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_RED_NOT_LEADER);
                } else if (BOAT_STATUS_getMyFleet() == GREEN_FLEET) {

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_GREEN_NOT_LEADER);
                }
                MAIN_assignDoWeNeedToUpdateHCWithCypherFlag(1);
```

```
                STEPPER_SetCrabCountFlagHeight(200);  //debug
                //MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update the crab count
to HC
                break;
        case TEAMMATE_REQUESTING_CYPHER:
                //give the teammate our cypher
                MAIN_assignDoWeNeedToSendCypherToTeamMateFlag(1);
                break;
        case NEPTUNE_OPEN_SEASON_BROADCAST_EVENT:
                //do nothing with open season broadcast...(perhaps debugging)
                //go to state playing game
                BOAT_STATUS_assignBoatState(BOAT_STATE_PLAYING_GAME);
                SERVO_CONTROLLER_setSOSFlagDOWN();
                BOAT_STATUS_assignLastCrabPotAccumulationRate(0);
                BOAT_STATUS_assignLastNumberOfCrabsTransfered(0);
                BOAT_STATUS_assignNumberOfCrabsOnboard(0);
                BOAT_STATUS_assignStateOfWaterloggedFlag(0);
                STEPPER_SetCrabCountFlagHeight(BOAT_STATUS_NumberOfCrabsOnboard());
                BOAT_STATUS_assignOpenSeasonStatus(1);
                BOAT_STATUS_assignLongTimerTargetValue(0);
                BOAT_STATUS_assignAreWeTimingSomething(0);
                BOAT_STATUS_assignLastCrabPotOrDockNumber(CRAB_POT_0);
                BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_OPEN_SEASON);
                MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update the crab count to
HC
                break;
        case NEW_DRIVE_COMMAND_FROM_HC:
                //i'm using this to trigger a response to the HC in the form a status message
                //in this way, if we fail to send a status message then we can still get the info
                //to the HC   ...robust or prepared for to fail?
                //MAIN_setWeNeedToSendAnUpdateToTheHCFlag();
                //actually no, this slows things way way down...
                break;
        default:
                break;
    }
}

static void EVENTS_runStateNotLeaderEventHandler(Event_t event) {
    switch(event) {
        case NO_EVENT:
                break;
        case XBEE_FULL_MESSAGE_RECEIVED_EVENT:
                //we've gotten a message over the radio, now parse that data
                //XBEE_InterperateReceivedMessage();  //<-- this is causing a stack overflow, i'm
moving it
                MAIN_assignDoWeNeedToProccessANewXbeeMessageFlag(1);
                break;
        case RFID_PIC_SIGNAL_LINE_HIGH_EVENT:
                //read in the RFID data from the rfid pic
                //start receiving the rfid stuff from the rfid pic
                MAIN_assignDoWeNeedToGetRFIDDataFromPicFlag(1);
                break;
        case NEW_RFID_SCAN_DATA:
                //now, do something with that new RFID scan data
                //MAIN_assignDoWeNeedToSendRFIDDataToSCFlag(1);
```

```
                //get the card type (if it is a crab pot or unloading dock do something with it)

        BOAT_STATUS_assignLastCrabPotOrDockNumber(RFID_WhatWasTheRFIDCard(RFID_getLast
S1(), RFID_getLastS2(), RFID_getLastS3()));
                //SERVO_CONTROLLER_setSOSFlagDOWN();     //debug
                switch (BOAT_STATUS_getLastCrabPotOrDockNumber()) {
                        case NO_MATCH:
                                //SERVO_CONTROLLER_setSOSFlagUP();   //debug
                                break;
                        case WHITE_TEAM_CARD:
                                //if it is a team card, we need to do something else with it
                                BOAT_STATUS_initalizeBoatStatus();
                                STEPPER_SetCrabCountFlagHeight(0);  //debug


        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_WHITE_TEAM_ASSIGNED);
                                MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update
the crab count to HC
                                SERVO_CONTROLLER_setSOSFlagDOWN();
                                break;
                        case GREEN_TEAM_CARD:
                                //if it is a team card, we need to do something else with it
                                //BOAT_STATUS_assignFleet(GREEN_FLEET);
                                BOAT_STATUS_assignFleet(GREEN_FLEET);

        BOAT_STATUS_assignLastCrabPotOrDockNumber(GREEN_FLEET);
                                 //FLEET_LEADER_TIMEOUT_TIME
                                SERVO_CONTROLLER_setSOSFlagUP();

        BOAT_STATUS_assignLongTimerTargetValue(FLEET_LEADER_TIMEOUT_TIME);
                                BOAT_STATUS_assignAreWeTimingSomething(1);
                                //TMR1IE=0;
                                MAIN_resetLongTimer();

        BOAT_STATUS_assignBoatState(BOAT_STATE_NOT_TEAM_LEADER);
                                MAIN_assignDoWeNeedToSendTeamIdentificationFlag(1);
//broadcast a we are on team xx message

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_GREEN_NOT_LEADER);
                                //set the timer target
                                //FLEET_LEADER_TIMEOUT_TIME
                                //set the time
                                //send a message to the fleet
                                //set the boat state
                                //if the timer expires, we become team captain  which we have to
indicate to the fleet, use the SOS flag...
                                //don't use the events below, use this instead
                                STEPPER_SetCrabCountFlagHeight(100);  //debug
                                MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update
the crab count to HC
                                break;
                        case RED_TEAM_CARD:
                        //case CRAB_POT_9:
                                //if it is a team card, we need to do something else with it
                                BOAT_STATUS_assignFleet(RED_FLEET);
                                BOAT_STATUS_assignLastCrabPotOrDockNumber(RED_FLEET);
```

```
                                    //FLEET_LEADER_TIMEOUT_TIME
                                    SERVO_CONTROLLER_setSOSFlagUP();

        BOAT_STATUS_assignLongTimerTargetValue(FLEET_LEADER_TIMEOUT_TIME);
                                    BOAT_STATUS_assignAreWeTimingSomething(1);
                                    //TMR1IE=0;
                                    MAIN_resetLongTimer();

        BOAT_STATUS_assignBoatState(BOAT_STATE_NOT_TEAM_LEADER);
                                    MAIN_assignDoWeNeedToSendTeamIdentificationFlag(1);
//broadcast a we are on team xx message

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_RED_NOT_LEADER);
                                    //set the timer target
                                    //FLEET_LEADER_TIMEOUT_TIME
                                    //set the time
                                    //send a message to the fleet
                                    //set the boat state
                                    //if the timer expires, we become team captain  which we have to
indicate to the fleet, use the SOS flag...
                                    //don't use the events below, use this instead
                                    STEPPER_SetCrabCountFlagHeight(100);  //debu
                                    MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update
the crab count to HC
                                    break;
                                default:  //it's a crab pot
                                    //this time we will send it to the SC pic
                                    //MAIN_assignDoWeNeedToSendRFIDDataToSCFlag(1);

                                    //send fake crab pot data to teammates
                                    BOAT_STATUS_assignLastCrabPotOrDockNumber(CRAB_POT_0);

        BOAT_STATUS_assignLastCrabPotAccumulationRate(RFID_getLastS3());
                                    MAIN_assignDoWeNeedToSendDataToFleetFlag(1);
                                    break;
                }
                break;
            case TEAMMATE_GIVING_CYPHER:
                //stop timer
                MAIN_StopLongTimer();
                BOAT_STATUS_assignAreWeTimingSomething(0);
                //store cypher
                BOAT_STATUS_assignXORCypher(XBEE_getNewXORCypher());
                BOAT_STATUS_assignNumberOfCrabsOnboard(XBEE_getNewXORCypher());
                //let hc know what the cypher is   ...do this....

                STEPPER_SetCrabCountFlagHeight(200);  //debug
                //MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update the crab count
to HC
                MAIN_assignDoWeNeedToUpdateHCWithCypherFlag(1);
                break;
            case LONG_TIMER_EXPIRED:
                MAIN_StopLongTimer();
                BOAT_STATUS_assignAreWeTimingSomething(0);

                SERVO_CONTROLLER_setSOSFlagDOWN();
```

```
                //now we become the master!!!
                BOAT_STATUS_assignXORCypher(MAIN_CreateXORCypher());
                BOAT_STATUS_assignNumberOfCrabsOnboard(XBEE_getNewXORCypher());
                BOAT_STATUS_assignBoatState(BOAT_STATE_TEAM_LEADER);

                if(BOAT_STATUS_getMyFleet() == RED_FLEET) {
                        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_RED_LEADER);
                } else if(BOAT_STATUS_getMyFleet() == GREEN_FLEET) {

        BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_GREEN_LEADER);
                }

                STEPPER_SetCrabCountFlagHeight(300);  //debug
                MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update the crab count to
HC
                //encrypt data
                //BOAT_STATUS_assignXORCypher(MAIN_CreateXORCypher());
                //indicate team leadership
                //change states
                break;
        case NEPTUNE_OPEN_SEASON_BROADCAST_EVENT:
                //do nothing with open season broadcast...(perhaps debugging)
                //go to state playing game
                BOAT_STATUS_assignBoatState(BOAT_STATE_PLAYING_GAME);
                SERVO_CONTROLLER_setSOSFlagDOWN();
                BOAT_STATUS_assignLastCrabPotAccumulationRate(0);
                BOAT_STATUS_assignLastNumberOfCrabsTransfered(0);
                BOAT_STATUS_assignNumberOfCrabsOnboard(0);
                BOAT_STATUS_assignStateOfWaterloggedFlag(0);
                STEPPER_SetCrabCountFlagHeight(BOAT_STATUS_NumberOfCrabsOnboard());
                BOAT_STATUS_assignOpenSeasonStatus(1);
                BOAT_STATUS_assignLongTimerTargetValue(0);
                BOAT_STATUS_assignAreWeTimingSomething(0);
                BOAT_STATUS_assignLastCrabPotOrDockNumber(CRAB_POT_0);
                BOAT_STATUS_assignLastImportantEvent(IMPT_EVNT_OPEN_SEASON);
                MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update the crab count to
HC
                break;
        case NEW_DRIVE_COMMAND_FROM_HC:
                //i'm using this to trigger a response to the HC in the form a status message
                //in this way, if we fail to send a status message then we can still get the info
                //to the HC   ...robust or prepared for to fail?
                //MAIN_setWeNeedToSendAnUpdateToTheHCFlag();
                //actually no, this slows things way way down...
                break;
        default:
                break;
    }
}
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Events Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */

#ifndef _EVENTS_H_
#define _EVENTS_H_

typedef enum {
	NO_EVENT,
	RFID_PIC_SIGNAL_LINE_HIGH_EVENT,
	SC_PIC_SIGNAL_LINE_HIGH_EVENT,
	RFID_PIC_SIGNAL_LINE_LOW_EVENT,
	NEPTUNE_OPEN_SEASON_BROADCAST_EVENT,
	NEPTUNE_DCH_UNLOADED_EVENT,
	NEPTUNE_DCH_REPAIRED_EVENT,
	NEPTUNE_CRAB_POT_DATA_RETURNED_EVENT,
	NEPTUNE_INVALID_KEY_EVENT,
	DCH_WATERLOGGED_EVENT,
	DCH_UN_WATERLOGGED_EVENT,
	WATERLOGGED_LINE_HI_FROM_SC,
	WATERLOGGED_LINE_LO_FROM_SC,
	HC_HAT_SWITCH_HIGH_EVENT,
	HC_HAT_SWITCH_LOW_EVENT,
	XBEE_FULL_MESSAGE_RECEIVED_EVENT,
	NEW_STATUS_MESSAGE_FROM_DCH_EVENT,
	NEW_DRIVE_COMMAND_FROM_HC,
	LONG_TIMER_EXPIRED,
	RED_FLEET_CARD_SWIPED,
	GREEN_FLEET_CARD_SWIPED,
	WHITE_FLEET_CARD_SWIPED,
	TEAMMATE_REQUESTING_CYPHER,
	TEAMMATE_GIVING_CYPHER,
	NEW_RFID_SCAN_DATA,
	NEW_SC_DATA
} Event_t;

Event_t EVENTS_CheckForEvents(void);
void EVENTS_Debug(void);

void EVENTS_RunStateMachine(Event_t event);

/* Helper function to check if any timers have expired since no separate routine for that.
static Event_t CheckForTimerExpired(void);*/
Event_t EVENTS_whatDidNeptuneRespondWith(void);

#endif
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Main Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#include <htc.h>
#include "BITDEFS.H" // BIT0HI, BIT0LO....definitions
#include "defs.h"
#include "events.h"
#include "xbee.h"
#include "ServoController.h"
#include "BoatStatus.h"
#include "rfid.h"
#include "RFID_ReadAndCategorize.h"
#include "stepper.h"

__CONFIG(HS & WDTDIS & BORDIS); //only call once

/* Variables */
//these were statics but i removed that assignement because they were "out of scope"
//in the debugger

unsigned char tempNeptuneResponseType = 0;
unsigned char doWeNeedToSendAnUpdateToTheHCFlag = 0;
unsigned char DoWeNeedToSendDataToNeptuneFlag = 0;
unsigned char DoWeNeedToSendDataToFleetFlag = 0;
unsigned char DoWeNeedToSendCypherToTeamMateFlag = 0;
unsigned char DoWeNeedToSendTeamIdentificationFlag = 0;

unsigned char DoWeNeedToGetRFIDDataFromPicFlag = 0;
unsigned char DoWeNeedToGetDataFromSCFlag = 0;
unsigned char DoWeNeedToSendRFIDDataToSCFlag = 0;
unsigned char DoWeNeedToProccessANewXbeeMessageFlag = 0;
unsigned char DoWeNeedToUpdateHCWithCypherFlag = 0;

unsigned int MAIN_longTimerCount = 0;
unsigned char MAIN_longTimerGoStopFlag = 0;

//boatStatusVar tempBoatStatus;

/* Function Prototypes */
void MAIN_Init(void);
void MAIN_SetupTimer1(void);
//void MAIN_Debug(void);

/* Functions */
void main (void) {
```

```
        MAIN_Init();
    MAIN_SetupTimer1();
        XBEE_Init();  //<-- no interrupts were enabled here
        SERVO_CONTROLLER_initServo();  //set the startup position of the servo
        BOAT_STATUS_initalizeBoatStatus();
        RFID_initRFIDSync();
        STEPPER_Init();

        //after all inits, enable interrupts
        ei(); //enable global interrupts
        PEIE = 1;  //all peripheral interrupts are enabled
        CCP1IE = 1; //enable CCP interrupts (for servo)
        T0IE = 1;
    //Initialize the state machine and loop forever to run it.
        //STATE_InitStateMachine();

        //XBEE_SendFakeStatusMessageToHCfromDCH(0, 0, 0, 0, 0);
        //STEPPER_SetCrabCountFlagHeight(300);
        XBEE_SendStatusMessageToHCfromDCH(12, 44, 32);

        for(;;) {
            EVENTS_RunStateMachine(EVENTS_CheckForEvents());
            if(DoWeNeedToProccessANewXbeeMessageFlag != 0) {
                    if(ASYNC_areWeSendingAUSARTmessage() == 0) {  //if we aren't using the Async
TX, clear to send a message
                            XBEE_InterperateReceivedMessage();          //i moved this
                            DoWeNeedToProccessANewXbeeMessageFlag = 0;  //lower the flag
                    }
            }

            if(DoWeNeedToSendDataToNeptuneFlag != 0) {
                    if(ASYNC_areWeSendingAUSARTmessage() == 0) { //if we aren't using the Async TX,
clear to send a message
                            //Send neptune crab pot data/dock data
                            XBEE_SendNeptuneCrabPotRFID(RFID_getLastS1(), RFID_getLastS2(),
RFID_getLastS3(), RFID_getLastSK1(), RFID_getLastSK2(), RFID_getLastSK3());
                            DoWeNeedToSendDataToNeptuneFlag = 0;
                    }
            }

            if(DoWeNeedToSendDataToFleetFlag != 0) {
                    if(ASYNC_areWeSendingAUSARTmessage() == 0) { //if we aren't using the Async TX,
clear to send a message
                            //SEND DATA FLEET HERE  //,,,,do this,,,,//
                            //here is the function

        XBEE_SendCrabPotInfoToOurFleet(BOAT_STATUS_getLastCrabPotOrDockNumber(),
BOAT_STATUS_getLastCrabPotAccumulationRate());
                            DoWeNeedToSendDataToFleetFlag = 0;
                    }
            }

            //do we have to update the HC about something? Like that we're fooded, or have new crabs, etc
            if(doWeNeedToSendAnUpdateToTheHCFlag != 0) {
                    if(ASYNC_areWeSendingAUSARTmessage() == 0) {  //if we aren't using the Async
TX, clear to send a message
```

```
        //XBEE_SendFakeStatusMessageToHCfromDCH(BOAT_STATUS_NumberOfCrabsOnboard(),
BOAT_STATUS_getWaterloggedFlag(), BOAT_STATUS_getLastCrabPotOrDockNumber()-0xA0,
BOAT_STATUS_getLastCrabPotAccumulationRate(),
BOAT_STATUS_getLastNumberOfCrabsTransfered(), BOAT_STATUS_getLastImportantEvent());

        //XBEE_SendStatusMessageToHCfromDCH(BOAT_STATUS_NumberOfCrabsOnboard(),
BOAT_STATUS_getWaterloggedFlag(), BOAT_STATUS_getLastCrabPotOrDockNumber()-0xA0);
                                //XBEE_SendFakeStatusMessageToHCfromDCH(0, 0, 0, 0, 0);

        XBEE_SendStatusMessageToHCfromDCH(BOAT_STATUS_NumberOfCrabsOnboard(), 0, 0);
                                doWeNeedToSendAnUpdateToTheHCFlag = 0;  //lower the flag
                        }
                }

//-----------------------------
                //do we have to update the HC about Cypher? Like that we're fooded, or have new crabs, etc
                if(DoWeNeedToUpdateHCWithCypherFlag != 0) {
                        if(ASYNC_areWeSendingAUSARTmessage() == 0) { //if we aren't using the Async TX,
clear to send a message

        //XBEE_SendFakeStatusMessageToHCfromDCH(BOAT_STATUS_NumberOfCrabsOnboard(),
BOAT_STATUS_getWaterloggedFlag(), BOAT_STATUS_getLastCrabPotOrDockNumber()-0xA0,
BOAT_STATUS_getLastCrabPotAccumulationRate(),
BOAT_STATUS_getLastNumberOfCrabsTransfered(), BOAT_STATUS_getLastImportantEvent());

        //XBEE_SendStatusMessageToHCfromDCH(BOAT_STATUS_NumberOfCrabsOnboard(),
BOAT_STATUS_getWaterloggedFlag(), BOAT_STATUS_getLastCrabPotOrDockNumber()-0xA0);
                                //XBEE_SendFakeStatusMessageToHCfromDCH(0, 0, 0, 0, 0);

        //XBEE_SendStatusMessageToHCfromDCH(BOAT_STATUS_NumberOfCrabsOnboard(), 0, 0);

        XBEE_SendStatusMessageToHCfromDCH(BOAT_STATUS_getXOR_Cypher(), 0, 0);
                                DoWeNeedToUpdateHCWithCypherFlag = 0;  //lower the flag
                        }
                }

                if(DoWeNeedToSendCypherToTeamMateFlag != 0) {
                        if(ASYNC_areWeSendingAUSARTmessage() == 0) { //if we aren't using the Async TX,
clear to send a message
                                //SEND DATA FLEET HERE  //,,,,do this,,,,//
                                XBEE_SendCypherKeyToAddress(XBEE_getLastTeamMatesAddrHigh(),
XBEE_getLastTeamMatesAddrLow());
                                DoWeNeedToSendCypherToTeamMateFlag = 0;
                        }
                }

                if(DoWeNeedToSendTeamIdentificationFlag != 0) {
                        if(ASYNC_areWeSendingAUSARTmessage() == 0) { //if we aren't using the Async TX,
clear to send a message
                                //SEND DATA FLEET HERE  //,,,,do this,,,,//
                                XBEE_SendImOnTheFleetMessage();
                                DoWeNeedToSendTeamIdentificationFlag = 0;
                        }
                }
```

```
		if(DoWeNeedToGetRFIDDataFromPicFlag != 0) { //if we aren't using the Sync, clear to send a
message
			if(RFID_AreWeSendingSync() == 0) { //is it clear to send/rec sync
				//start getting RFID data
				RFID_setSyncMode(COLLECT_RFID_DATA_MODE);
				DoWeNeedToGetRFIDDataFromPicFlag = 0;
			}
		}

		if(DoWeNeedToGetDataFromSCFlag != 0) {
			if(RFID_AreWeSendingSync() == 0) { //is it clear to send/rec sync
				//start getting sync data
				RFID_setSyncMode(COLLECT_SC_DATA_MODE);
				DoWeNeedToGetDataFromSCFlag = 0;
			}
		}

		if(DoWeNeedToSendRFIDDataToSCFlag != 0) {
			if(RFID_AreWeSendingSync() == 0) { //is it clear to send/rec sync
				//start sending sync data
				RFID_setSyncMode(SEND_RFID_TO_SC_MODE);
				DoWeNeedToSendRFIDDataToSCFlag = 0;
			}
		}
	}//END OF MAIN FOR LOOP
}//END OF MAIN FUNCTION


//interrupt response routine
void interrupt int_res (void) {
	unsigned char x = 0;

	if(RCIF != 0) { //we've had a receive interrupt ASYNC
		XBEE_ReadIncommingAsyncByte(RCREG);  //FIXED this function is too slow!!!, there is an
overrun error.  (bitfield mouseover on RCSTA)
		RCIF = 0; //clear the interrupt flag
	}

	if(TXIF != 0) { //we've had a TX interrupt meaning the output buffer is empty and we can transmit
now
		//check to see if there actually is a message to send.
		if(ASYNC_areWeSendingAUSARTmessage() == 0) { //I had to change these to reduce the
number of function calls
			//we should never be getting here, but if we do, this will help to keep the system running
			TXIE = 0; //disable this interrupt
		} else {
			//XBEE_SendNextMessageByte();
			ASYNC_sendNextUSARTTxMessageByte();
		}
	}

	if (T0IF) {
		T0IF = 0;

		if(MAIN_longTimerGoStopFlag != 0) { //for timing long events
			MAIN_longTimerCount++;  //increment the main timer counter
```

```
        }

        if(STEPPER_getAdjustStepperMotorFlag() != 0) {
                STEPPER_respondToTimerOverflow();
        }
    }

    //this interrupt is for the servo
    if(CCP1IF != 0) { //check to see if there has been a timer software output compare
        if(CCP1IE != 0) {
                //DON'T FORGET TO ENABLE THE INTERRUPT!!!
                SERVO_CONTROLLER_respondToCompareInterrupt();
                CCP1IF = 0;  //clear the flag
        }
    }

    if(SSPIF != 0) {
        if(SSPIE != 0) {
                if(BF!=0) { //if the buffer is full
                        RFID_respondToInterrupt();
                        SSPIF = 0;          //Clear the SSP flag
                }
                if(WCOL != 0) { //has there been a buffer write collision?
                        //indicate to us that this has happened
                        //but for now, lower that flag so that this interrupt doesn't keep firing
                        WCOL = 0;
                }
        }
    }
}

void MAIN_Init(void){
    //set inputs and clear outputs
    PORTA = 0x00; //clear all of port a
    PORTB = 0x00; //clear all of port b
    PORTC = 0x00; //clear all of port c

    /*FOR SYNC COM*/
    //SYNC receiving
    SETBIT(SDI_DDR, SDI_BIT);  //input sdi
    SETBIT(CollectRFIDDataTrigger_DDR, CollectRFIDDataTrigger_BIT); //Alert from RFID
    SETBIT(CollectSCDataTrigger_DDR, CollectSCDataTrigger_BIT);          //Alert form SC

    //SYNC sending
    CLEARBIT(SCK_DDR, SCK_BIT); //output  clock
    CLEARBIT(SDO_DDR, SDO_BIT); //output  sdo

    //SYNC Master Control System
    CLEARBIT(RFID_SS_CONTROL_DDR, RFID_SS_CONTROL_BIT); //output rfid SS line
    CLEARBIT(SC_SS_CONTROL_DDR, SC_SS_CONTROL_BIT); //output sc SS line

    //SERVO OUTPUT
    CLEARBIT(SERVO_OUTPUT_DDR, SERVO_OUTPUT_BIT); //output

    /*FOR STEPPER MOTOR CONTROL*/
    /*FOR WATERLOGGED SENSOR INPUT*/
```

```
        SETBIT(WATERLOGGED_INPUT_FROM_SC_DDR,
WATERLOGGED_INPUT_FROM_SC_BIT);          //waterlogged sensor

        /*for asyncronous communications*/
        CLEARBIT(ASYNC_TX_DDR, ASYNC_TX_BIT); //output
        SETBIT(ASYNC_RX_DDR, ASYNC_RX_BIT); //input

        /*NO ADC PINS*/
        ANSEL = 0;  //NO ADC, all digital pins
        ANSELH = 0;  //NO ADC, all digital pins

        //initialize the RFID_SS and SC_SS to 1
        SETBIT(RFID_SS_CONTROL_PORT, RFID_SS_CONTROL_BIT);
        asm("nop");
        SETBIT(SC_SS_CONTROL_PORT, SC_SS_CONTROL_BIT);

}

void MAIN_SetupTimer1(void) {
        //Using timer 1 for the step control
        //Turn on timer 1
        //use internal instruction clock 20mhz/4 = 5mhz
        //use a /8 prescaler = .625Mhz
        T1CON = 0b10110101;         //setup the inital timer control bits
        //di();

        //now set it up for output compare, internally, software, no external pin
        CCP1CON = 0b00001010; //set the CCP1CON compare reg to output compare, software
        //setup output compare which is needed for the servos!

        CCPR1H  = 0x00; //move the inital step time value into the upper byte of the compare reg
        CCPR1L  = 0x00; //clear the lower byte of the compare time reg

        TMR1H = 0x00; //clear timer high byte
        TMR1L = 0x00; //clear timer low byte
        PIR1 = 0x00; //clear interrupt flags
        //ei();
}

/*This Function Initalizes the Serial Data functionality*/
void initSSPasMaster(void){
        SSPSTAT = 0x00; //Clear the Sync Serial Port Status Register, SMP = 0, CKE = 0
        //Enable Interrupts if we want, do that here
        SSPCON    =          0b00010001;                  //setup SPI port
                                                                  //(5)Don't Enable SSP Yet, (4)
Clock Polarity Select (idle high)
                                                                  //(3:0) Clock = SPI master mode,
clock = Fosc/16
        SETBIT(SSPCON, 5);                          //enable SSP
        //dummy read of Message Buffer
        //SYNC_SDI_FirstReadIn = SSPBUF;
}

//TIMERS
void MAIN_resetLongTimer(void) {
        MAIN_longTimerCount = 0;
```

```
        MAIN_longTimerGoStopFlag = 1;
}

unsigned int MAIN_getLongTimerCount(void) {
        return MAIN_longTimerCount;
}

void MAIN_IncrementLongTimer(void) {
        MAIN_longTimerCount++;
}

void MAIN_StopLongTimer(void) {
        MAIN_longTimerGoStopFlag = 0;
        MAIN_longTimerCount = 0;
}

unsigned char MAIN_CreateXORCypher(void) {
        return TMR1L; //return the low byte of timer 1: this is random
}

unsigned char MAIN_EncodeOrDecodeByteUsingCyper(unsigned char dataByte) {
        return (dataByte ^ BOAT_STATUS_getXOR_Cypher());
}

void MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(unsigned char flagState) {
        doWeNeedToSendAnUpdateToTheHCFlag = flagState;
}

void MAIN_assignDoWeNeedToSendDataToNeptuneFlag(unsigned char flagState) {
        DoWeNeedToSendDataToNeptuneFlag = flagState;
}

void MAIN_assignDoWeNeedToSendDataToFleetFlag(unsigned char flagState) {
        DoWeNeedToSendDataToFleetFlag = flagState;
}

void MAIN_assignDoWeNeedToSendCypherToTeamMateFlag(unsigned char flagState) {
        DoWeNeedToSendCypherToTeamMateFlag = flagState;
}

void MAIN_assignDoWeNeedToSendTeamIdentificationFlag(unsigned char flagState) {
        DoWeNeedToSendTeamIdentificationFlag = flagState;
}

void MAIN_assignDoWeNeedToGetRFIDDataFromPicFlag(unsigned char flagState) {
        DoWeNeedToGetRFIDDataFromPicFlag = flagState;
}

void MAIN_assignDoWeNeedToGetDataFromSCFlag(unsigned char flagState) {
        DoWeNeedToGetDataFromSCFlag = flagState;
}

void MAIN_assignDoWeNeedToSendRFIDDataToSCFlag(unsigned char flagState) {
        DoWeNeedToSendRFIDDataToSCFlag = flagState;
}
```

```
void MAIN_assignDoWeNeedToProccessANewXbeeMessageFlag(unsigned char flagState) {
	DoWeNeedToProccessANewXbeeMessageFlag = flagState;
}

void MAIN_assignDoWeNeedToUpdateHCWithCypherFlag(unsigned char flagState) {
	DoWeNeedToUpdateHCWithCypherFlag= flagState;
}
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Xbee Async Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */

#include "PirateAsync.h"

usartTxMessage TxUSARTmessageBuffer;  //this is the local buffer where a Transmitting message is
stored
unsigned char sendingUSARTmessageFlag = 0;  //this is a local flag that is raised when a message is being
sent

//Higher level USART transmitter
//Take in a data structure
//send that data structure
//indicate when sending is done
//disable TX interrupts when not sending
//enable TX interrupt when we want to send something
//send one byte at a time until we are done, then disable TX interrupts

//this function starts sending a usart message
//it raises the sending message flag
//then takes the message contents and stores them in the local buffer
//then it enables async transmit interrupts so that the message can start sending
void ASYNC_sendUSARTmessage(usartTxMessage message) {
        sendingUSARTmessageFlag = 1; //raise the flag that indicates that we are sending a message
        //transfer USART message to TX message buffer
        TxUSARTmessageBuffer = message;
        TxUSARTmessageBuffer.messageIndex = 0;  //reset the index
        //enable TX interrupts
        TXIE = 1;
}

//this is called when the last byte of the message is sent
//it disables the interrupt and lowers the sending flag
void ASYNC_finishedSendingUSARTmessage(void) { //do the cleanup assocated with finishing sending a
message
        //disable TX interrupts
        TXIE = 0;
        sendingUSARTmessageFlag = 0; //raise the flag that indicates that we are sending a message
}

//this function returns the state of the are-we-sending-flag
unsigned char ASYNC_areWeSendingAUSARTmessage(void) {
        return sendingUSARTmessageFlag;
}

//this function is called from the Interrupt response routine
```

```
void ASYNC_sendNextUSARTTxMessageByte(void) {
	//this will send the data byte in the message buffer that is being pointed to with the index value
	ASYNC_sendUSARTbyte(TxUSARTmessageBuffer.messageByte[TxUSARTmessageBuffer.messa
geIndex]);
	//increase the index count
	if(TxUSARTmessageBuffer.messageIndex >= (TxUSARTmessageBuffer.numberOfBytes-1)) { //we
are at the last byte of the message
		//we have finished sending the message, so cleanup
		ASYNC_finishedSendingUSARTmessage();
	} else {
		TxUSARTmessageBuffer.messageIndex++;
	}
}

//this function simply puts a byte into the TXREG  <-- if we have too many nested functions, this one can
be re-encorperated
//by the functions that call it
void ASYNC_sendUSARTbyte(unsigned char dataByteToSend) {
	//do some sort of MAC or line checking here...if we need to
	TXREG = dataByteToSend;
}

//this does the setup for Async communication
void ASYNC_setupUSART(void) {
	//  di();
		/*Baud rate setup*/
		SCKP = 0; //transmit non-inverted data
		ABDEN = 0; //autobaud functionality turned off
		WUE = 0; //normal receiver operation
	//  BRG16 = 0; //8 bit baud generator used
	//  SPBRG = 63; //Set USART speed  9.6 kbps, .16% error, FOSC 20MHz


	//  BRG16 = 1;
	//  BRGH = 0;
	//  SPBRG = 129; //Set USART speed  9.6 kbps, .16% error, FOSC 20MHz, 5V

		BRG16 = 0;
		BRGH = 1;
		SPBRG = 129; //Set USART speed  9.6 kbps, .16% error, FOSC 20MHz, 5V

		/*TRANSMITTER*/
		//TXSTA REGISTER
			TX9 = 0; //8 bit transmission
			SYNC = 0; //async mode
			//BRGH = 1; //High Speed moded
			TRMT = 1; //Transmit shift reg
			TXEN = 1; //enable transmission
			TXIF = 0; //lower the transmit flag
		/*RECIVER*/
		//RCSTA REGISTER
			RX9 = 0; //8 bit reception
			ADDEN = 0; //disable address detection wakup
			FERR = 0; //framing error flag lowered
			OERR = 0; //overrun error flag lowered
			SPEN = 1;		//enable serial port pins
```

```
                CREN = 1; //Enable continious reception
                RCIF = 0; //clear the flag

            /*INTERRUPTS*/
                RCIE = 1;  //Enable RX interrupt
        //ei();
}
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Xbee Async Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


//This function requres timer 1 to be setup and running...it is used for delays and such
//this can be changed later to timer 2 or 0 or ....etc, but not right now

#ifndef _PIRATE_ASYNC_H_
#define _PIRATE_ASYNC_H_

#include <htc.h>
#include "defs.h"  //this is for the pin defs

/* Port A bits: */

/* Port B bits: */

/* Port C bits */

//this is the outgoing message structure
typedef struct{
  unsigned char messageByte[26];
  unsigned char messageIndex;
  unsigned char numberOfBytes;
}usartTxMessage;

/*functions*/
void ASYNC_setupUSART(void);
void ASYNC_sendUSARTmessage(usartTxMessage message);
void ASYNC_finishedSendingUSARTmessage(void);
unsigned char ASYNC_areWeSendingAUSARTmessage(void);
void ASYNC_sendNextUSARTTxMessageByte(void);
void ASYNC_sendUSARTbyte(unsigned char dataByteToSend);

#endif
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * RFID Categorize Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */

#include <htc.h>
#include "defs.h"
#include "RFID_ReadAndCategorize.h"

//#define

/* Variables */
unsigned char Read = 0;
unsigned char firstCardCheck = 0;
unsigned char firstCheck = 0;
unsigned char Role = 0;
unsigned char Match = FALSE;

unsigned char RFID_WhatWasTheRFIDCard(unsigned char RFIDOne, unsigned char RFIDTwo, unsigned
char RFIDThree) {
        //return checkMatch(RFIDOne, RFIDTwo, RFIDThree);
        //See what the third hex value matches
        switch (RFIDThree) {
            case 0x47:
                        Role = 0;
                        firstCardCheck = DRY_DOCK;
                        break;
            case 0x66:
                        Role = 1;
                        firstCardCheck = DRY_DOCK;
                        break;
            case 0x7F:
                        Role = 2;
                        firstCardCheck = LOADING_DOCK;
                        break;
            case 0x79:
                        Role = 3;
                        firstCardCheck = LOADING_DOCK;
                        break;
            case 0xD4:
                        Role = 4;
                        firstCardCheck = CRAB_POT_1;
                        break;
            case 0x07:
                        Role = 5;
                        firstCardCheck = CRAB_POT_2;
                        break;
            case 0xC8:
```

```
                Role = 6;
                firstCardCheck = CRAB_POT_3;
                break;
        case 0x4F:
                Role = 7;
                firstCardCheck = CRAB_POT_4;
                break;
        case 0x5E:
                Role = 8;
                firstCardCheck = CRAB_POT_5;
                break;
        case 0xFE:
                Role = 9;
                firstCardCheck = CRAB_POT_6;
                break;
        case 0xED:
                Role = 10;
                firstCardCheck = CRAB_POT_7;
                break;
        case 0xF0:
                Role = 11;
                firstCardCheck = CRAB_POT_8;
                break;
        case 0x99:
                Role = 12;
                firstCardCheck = CRAB_POT_9;
                break;
        case 0x96:
                Role = 13;
                firstCardCheck = CRAB_POT_10;
                break;
        case 0x83:
                Role = 14;
                firstCardCheck = CRAB_POT_11;
                break;
        case 0xE0:
                Role = 15;
                firstCardCheck = CRAB_POT_12;
                break;
        case 0xEC:
                Role = 16;
                firstCardCheck = RED_TEAM_CARD;
                break;
        case 0x5A:
                Role = 17;
                firstCardCheck = GREEN_TEAM_CARD;
                break;
        case 0x06:
                Role = 18;
                firstCardCheck = WHITE_TEAM_CARD;
                break;
        default:
                firstCardCheck = NO_MATCH; //No Match
                break;
}
```

```
//First round of filtering
//There was no match
if (firstCardCheck==NO_MATCH) {
    return NO_MATCH;
//Or there was a firstMatch
}

switch (Role) {
    case 0:
            if(RFIDTwo == 0x6F) {
                    Match = TRUE;
            } else {
                    Match = FALSE;
            }
            break;
    case 1:
            if(RFIDTwo == 0x31) {
                    Match = TRUE;
            } else {
                    Match = FALSE;
            }
            break;
    case 2:
            if(RFIDTwo == 0x12) {
                    Match = TRUE;
            } else {
                    Match = FALSE;
            }
            break;
    case 3:
            if(RFIDTwo == 0xB9) {
                    Match = TRUE;
            } else {
                    Match =FALSE;
            }
            break;
    case 4:
            if(RFIDTwo == 0xCF) {
                    Match = TRUE;
            } else {
                    Match = FALSE;
            }
            break;
    case 5:
            if(RFIDTwo == 0xCD) {
                    Match = TRUE;
            } else {
                    Match = FALSE;
            }
            break;
    case 6:
            if(RFIDTwo == 0xCF) {
                    Match = TRUE;
            } else {
                    Match = FALSE;
            }
```

```
                break;
        case 7:
                if(RFIDTwo == 0x07) {
                        Match = TRUE;
                } else {
                        Match = FALSE;
                }
                break;
        case 8:
                if(RFIDTwo == 0x08) {
                        Match = TRUE;
                } else {
                        Match = FALSE;
                }
                break;
        case 9:
                if(RFIDTwo == 0x1E) {
                        Match = TRUE;
                } else {
                        Match = FALSE;
                }
                break;
        case 10:
                if(RFIDTwo == 0xBC) {
                        Match = TRUE;
                } else {
                        Match = FALSE;
                }
                break;
        case 11:
                if(RFIDTwo == 0xED) {
                        Match = TRUE;
                } else {
                        Match = FALSE;
                }
                break;
        case 12:
                if(RFIDTwo == 0xE3) {
                        Match = TRUE;
                } else {
                        Match = FALSE;
                }
                break;
        case 13:
                if(RFIDTwo == 0x10) {
                        Match = TRUE;
                } else {
                        Match = FALSE;
                }
                break;
        case 14:
                if(RFIDTwo == 0xBE) {
                        Match = TRUE;
                } else {
                        Match = FALSE;
                }
```

```
                break;
        case 15:
                if(RFIDTwo == 0xFC) {
                        Match = TRUE;
                } else {
                        Match = FALSE;
                }
                break;
        case 16:
                if(RFIDTwo == 0xD5) {
                        Match = TRUE;
                } else {
                        Match = FALSE;
                }
                break;
        case 17:
                if(RFIDTwo == 0x9C) {
                        Match = TRUE;
                } else {
                        Match = FALSE;
                }
                break;
        case 18:
                if(RFIDTwo == 0x9E) {
                        Match = TRUE;
                } else {
                        Match = FALSE;
                }
                break;
        default:
                Match = FALSE;
                break;
}

if(Match == FALSE) {
    return NO_MATCH;
}

switch (Role) {
    case 0:
                if(RFIDOne == 0x5E) {
                        Match = TRUE;
                }else {
                        Match = FALSE;
                }
    break;
    case 1:
                if(RFIDOne == 0x5F) {
                        Match = TRUE;
                }else {
                        Match = FALSE;
                }
    break;
    case 2:
                if(RFIDOne == 0x84) {
                        Match = TRUE;
```

```
		}else {
			Match = FALSE;
		}
	break;
	case 3:
		if(RFIDOne == 0x83) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 4:
		if(RFIDOne == 0x83) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 5:
		if(RFIDOne == 0x83) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 6:
		if(RFIDOne == 0x83) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 7:
		if(RFIDOne == 0x84) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 8:
		if(RFIDOne == 0x84) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 9:
		if(RFIDOne == 0x84) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 10:
		if(RFIDOne == 0x83) {
			Match = TRUE;
```

```
		}else {
			Match = FALSE;
		}
	break;
	case 11:
		if(RFIDOne == 0x83) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 12:
		if(RFIDOne == 0x83) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 13:
		if(RFIDOne == 0x84) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 14:
		if(RFIDOne == 0x83) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 15:
		if(RFIDOne == 0x83) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 16:
		if(RFIDOne == 0x35) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 17:
		if(RFIDOne == 0x35) {
			Match = TRUE;
		}else {
			Match = FALSE;
		}
	break;
	case 18:
		if(RFIDOne== 0x32) {
			Match = TRUE;
```

```
                }else {
                        Match = FALSE;
                }
            break;
            default:
                Match = FALSE;
            break;
        }

        if(Match == TRUE) {
            return firstCardCheck;
        } else {
            return NO_MATCH;
        }
}

void matchTheThirdHex(unsigned char RFIDThree) {
        switch (RFIDThree) {
            case 0x47:
                Role = 0;
                firstCardCheck = DRY_DOCK;
                break;
            case 0x66:
                Role = 1;
                firstCardCheck = DRY_DOCK;
                break;
            case 0x7F:
                Role = 2;
                firstCardCheck = LOADING_DOCK;
                break;
            case 0x79:
                Role = 3;
                firstCardCheck = LOADING_DOCK;
                break;
            case 0xD4:
                Role = 4;
                firstCardCheck = CRAB_POT_1;
                break;
            case 0x07:
                Role = 5;
                firstCardCheck = CRAB_POT_2;
                break;
            case 0xC8:
                Role = 6;
                firstCardCheck = CRAB_POT_3;
                break;
            case 0x4F:
                Role = 7;
                firstCardCheck = CRAB_POT_4;
                break;
            case 0x5E:
                Role = 8;
                firstCardCheck = CRAB_POT_5;
                break;
            case 0xFE:
                Role = 9;
```

```
                firstCardCheck = CRAB_POT_6;
                break;
        case 0xED:
                Role = 10;
                firstCardCheck = CRAB_POT_7;
                break;
        case 0xF0:
                Role = 11;
                firstCardCheck = CRAB_POT_8;
                break;
        case 0x99:
                Role = 12;
                firstCardCheck = CRAB_POT_9;
                break;
        case 0x96:
                Role = 13;
                firstCardCheck = CRAB_POT_10;
                break;
        case 0x83:
                Role = 14;
                firstCardCheck = CRAB_POT_11;
                break;
        case 0xE0:
                Role = 15;
                firstCardCheck = CRAB_POT_12;
                break;
        case 0xEC:
                Role = 16;
                firstCardCheck = RED_TEAM_CARD;
                break;
        case 0x5A:
                Role = 17;
                firstCardCheck = GREEN_TEAM_CARD;
                break;
        case 0x06:
                Role = 18;
                firstCardCheck = WHITE_TEAM_CARD;
                break;
        default:
                firstCardCheck = NO_MATCH; //No Match
                break;
    }
}

unsigned char doesSecondHexMatch(unsigned char RFIDTwo) {
    switch (Role) {
        case 0:
                if(RFIDTwo == 0x6F) {
                        return TRUE;
                }else {
                        return FALSE;
                }
        case 1:
                if(RFIDTwo == 0x31) {
                        return TRUE;
                }else {
```

```
                return FALSE;
        }
case 2:
        if(RFIDTwo == 0x12) {
                return TRUE;
        }else {
                return FALSE;
        }
case 3:
        if(RFIDTwo == 0xB9) {
                return TRUE;
        }else {
                return FALSE;
        }
case 4:
        if(RFIDTwo == 0xCF) {
                return TRUE;
        }else {
                return FALSE;
        }
case 5:
        if(RFIDTwo == 0xCD) {
                return TRUE;
        }else {
                return FALSE;
        }
case 6:
        if(RFIDTwo == 0xCF) {
                return TRUE;
        }else {
                return FALSE;
        }
case 7:
        if(RFIDTwo == 0x07) {
                return TRUE;
        }else {
                return FALSE;
        }
case 8:
        if(RFIDTwo == 0x08) {
                return TRUE;
        }else {
                return FALSE;
        }
case 9:
        if(RFIDTwo == 0x1E) {
                return TRUE;
        }else {
                return FALSE;
        }
case 10:
        if(RFIDTwo == 0xBC) {
                return TRUE;
        }else {
                return FALSE;
        }
```

```
            case 11:
                        if(RFIDTwo == 0xED) {
                                    return TRUE;
                        }else {
                                    return FALSE;
                        }
            case 12:
                        if(RFIDTwo == 0xE3) {
                                    return TRUE;
                        }else {
                                    return FALSE;
                        }
            case 13:
                        if(RFIDTwo == 0x10) {
                                    return TRUE;
                        }else {
                                    return FALSE;
                        }
            case 14:
                        if(RFIDTwo == 0xBE) {
                                    return TRUE;
                        }else {
                                    return FALSE;
                        }
            case 15:
                        if(RFIDTwo == 0xFC) {
                                    return TRUE;
                        }else {
                                    return FALSE;
                        }
            case 16:
                        if(RFIDTwo == 0xD5) {
                                    return TRUE;
                        }else {
                                    return FALSE;
                        }
            case 17:
                        if(RFIDTwo == 0x9C) {
                                    return TRUE;
                        }else {
                                    return FALSE;
                        }
            case 18:
                        if(RFIDTwo == 0x9E) {
                                    return TRUE;
                        } else {
                                    return FALSE;
                        }
            default:
                        return FALSE;
        }
}

unsigned char doesFirstHexMatch(unsigned char RFIDOne) {
        switch (Role) {
            case 0:
```

```
                if(RFIDOne == 0x5E) {
                        return TRUE;
                }else {
                        return FALSE;
                }
        case 1:
                if(RFIDOne == 0x5F) {
                        return TRUE;
                }else {
                        return FALSE;
                }
        case 2:
                if(RFIDOne == 0x84) {
                        return TRUE;
                }else {
                        return FALSE;
                }
        case 3:
                if(RFIDOne == 0x83) {
                        return TRUE;
                }else {
                        return FALSE;
                }
        case 4:
                if(RFIDOne == 0x83) {
                        return TRUE;
                }else {
                        return FALSE;
                }
        case 5:
                if(RFIDOne == 0x83) {
                        return TRUE;
                }else {
                        return FALSE;
                }
        case 6:
                if(RFIDOne == 0x83) {
                        return TRUE;
                }else {
                        return FALSE;
                }
        case 7:
                if(RFIDOne == 0x84) {
                        return TRUE;
                }else {
                        return FALSE;
                }
        case 8:
                if(RFIDOne == 0x84) {
                        return TRUE;
                }else {
                        return FALSE;
                }
        case 9:
                if(RFIDOne == 0x84) {
                        return TRUE;
```

```
        }else {
                return FALSE;
        }
case 10:
        if(RFIDOne == 0x83) {
                return TRUE;
        }else {
                return FALSE;
        }
case 11:
        if(RFIDOne == 0x83) {
                return TRUE;
        }else {
                return FALSE;
        }
case 12:
        if(RFIDOne == 0x83) {
                return TRUE;
        }else {
                return FALSE;
        }
case 13:
        if(RFIDOne == 0x84) {
                return TRUE;
        }else {
                return FALSE;
        }
case 14:
        if(RFIDOne == 0x83) {
                return TRUE;
        }else {
                return FALSE;
        }
case 15:
        if(RFIDOne == 0x83) {
                return TRUE;
        }else {
                return FALSE;
        }
case 16:
        if(RFIDOne == 0x35) {
                return TRUE;
        }else {
                return FALSE;
        }
case 17:
        if(RFIDOne == 0x35) {
                return TRUE;
        }else {
                return FALSE;
        }
case 18:
        if(RFIDOne== 0x32) {
                return TRUE;
        }else {
                return FALSE;
```

```
                }
        default:
                return FALSE;
    }
}
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * RFID Categorize Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#ifndef _READ_AND_CAT_H_
#define _READ_AND_CAT_H_

//public
unsigned char RFID_WhatWasTheRFIDCard(unsigned char RFIDOne, unsigned char RFIDTwo, unsigned
char RFIDThree);

/* Private */
unsigned char checkMatch(unsigned char RFIDOne, unsigned char RFIDTwo, unsigned char RFIDThree);
unsigned char doesSecondHexMatch(unsigned char RFIDTwo);
void matchTheThirdHex(unsigned char RFIDOne);
unsigned char doesFirstHexMatch(unsigned char RFIDOne);

#endif
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * RFID Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */

#include "rfid.h"

/* SDO Array Lengths */
#define A_ARRAY_LENGTH 8              //OUT: A1-A2-A3-A4-A5-A6-A7-A8        //IN: RFID
#define B_ARRAY_LENGTH 5              //OUT: B1-B2-B3-B4-B5                 //IN: SC data
#define RFID_OUT_ARRAY_LENGTH 5  //OUT: 02-RFID-RFID-RFID-03   //IN: C1-C2-C3-C4-C5
#define LARGEST_SDO_LENGTH 8

#define RFID_IN_ARRAY_LENGTH 8  //IN: RFID                                //OUT: A1-A2-A3-A4-
A5-A6-A7-A8
#define LARGEST_SDI_LENGTH 8

#define SC_ARRAY_LENGTH 5

//variables
////// --- Variables for RFID Receive from RFID Slave PIC --- ////////
/*SDO*/
        //Generic SDO Array
        unsigned char SDO_Array[LARGEST_SDO_LENGTH];
        unsigned char SDO_Array_Length = 0;
        unsigned char SDO_ArrayIndex = 0;
        unsigned char SendingByteFlag = 0;
        //unsigned char BOAT_ReceivedData = 0;
        //unsigned char BOAT_DataIn = 0;
        unsigned char SDI_Array[LARGEST_SDI_LENGTH];
        unsigned char SDI_Array_Length = 0;
        unsigned char SDI_ArrayIndex = 0;
        //Specific SDI Arrays
        //unsigned char SYNC_SDI_RFIDDataArray[RFID_IN_ARRAY_LENGTH];
        //unsigned char BOAT_SS_CONTROL_BIT = 0;

unsigned char LastSK1Byte;
unsigned char LastSK2Byte;
unsigned char LastSK3Byte;

unsigned char haveWeReceivedNewSCDataFlag = 0;
unsigned char haveWeReceivedANewRFIDScanFlag = 0;  //this flag will let the event checker know when
we have received and proccessed
                                                                                        //rfid
data and are ready to give it to neptune/sc  etc
unsigned char LastS1Byte;
unsigned char LastS2Byte;
unsigned char LastS3Byte;
```

```
unsigned char areWeUsingTheSyncSerialFlag = 0;
unsigned char areWeReceivingARFIDMessageFlag = 0;
unsigned char areWeReceivingASCMessageFlag = 0;

unsigned char syncComMode = NOT_BEING_USED_MODE;

unsigned char SyncByteIn = 0x00;

//functions
//this function will  initalize the sync serial subsytem and any
//rfid or sc things that need to be initalized
void RFID_initRFIDSync(void) {
	//This Function Initalizes the Serial Data functionality in MASTER mode
	SSPSTAT = 0x00; //Clear the Sync Serial Port Status Register, SMP = 0, CKE = 0
	//Enable Interrupts if we want, do that here
	SSPCON    =          0b00010001;                //setup SPI port
                                                                //(5)Don't Enable SSP Yet, (4)
Clock Polarity Select (idle high)
                                                                //(3:0) Clock = SPI master mode,
clock = Fosc/16
	SETBIT(SSPCON, 5);                          //enable SSP
	//dummy read of Message Buffer
	//SYNC_SDI_FirstReadIn = SSPBUF;
}

unsigned char RFID_haveWeReceivedANewRFIDScanFlag(void) {
	if(haveWeReceivedANewRFIDScanFlag != 0)  {
	    //lower the flag if the message has been checked
	    haveWeReceivedANewRFIDScanFlag = 0;
	    return 1;
	} else {
	    return 0;
	}
}

unsigned char RFID_haveWeReceivedNewSCDataFlag(void) {
	if(haveWeReceivedNewSCDataFlag != 0) {
	    //lower the flag if the message has been checked
	    haveWeReceivedNewSCDataFlag = 0;
	    return 1;
	} else {
	    return 0;
	}
}

//this returns the RFID cards S1 byte (3rd least significant)
unsigned char RFID_getLastS1(void) {
	return LastS1Byte;
}

//this returns the RFID cards S2 byte (2rd least significant)
unsigned char RFID_getLastS2(void) {
	return LastS2Byte;
}
```

```
//this returns the RFID cards S3 byte (least significant byte)
unsigned char RFID_getLastS3(void) {
	return LastS3Byte;
}

//this returns the RFID cards S1 byte (3rd least significant)
unsigned char RFID_getLastSK1(void) {
	return LastSK1Byte;
}

//this returns the RFID cards S2 byte (2rd least significant)
unsigned char RFID_getLastSK2(void) {
	return LastSK2Byte;
}

//this returns the RFID cards S3 byte (least significant byte)
unsigned char RFID_getLastSK3(void) {
	return LastSK3Byte;
}

//for the event checker
unsigned char RFID_CheckReceivingARFIDMessageFlag(void) {
	return areWeReceivingARFIDMessageFlag;
}

//for the event checker
unsigned char RFID_CheckReceivingASCMessageFlag(void) {
	return areWeReceivingASCMessageFlag;
}

//checks to see if we are currently using the sync system ie: sending or receiving
unsigned char RFID_AreWeSendingSync(void) {
	return areWeUsingTheSyncSerialFlag;
}

void RFID_respondToInterrupt(void){
	if(syncComMode == COLLECT_RFID_DATA_MODE) {
	    SyncByteIn = SSPBUF;                 //Read in message that we got when we transmitted
	    SETBIT(RFID_SS_CONTROL_PORT, RFID_SS_CONTROL_BIT);   //raise the slave select line

	    if(SyncByteIn == RFID_START) {
			//set index of incomming SDI to zero
			SDI_ArrayIndex = 0;
			SDI_Array[SDI_ArrayIndex] = SyncByteIn;
			SDI_ArrayIndex++;

			CLEARBIT(RFID_SS_CONTROL_PORT, RFID_SS_CONTROL_BIT);   //lower the
slave select line
			RFID_SendMessage(SDO_Array[SDO_ArrayIndex]);
			SDO_ArrayIndex++;
	    } else if(SyncByteIn == 0x00) {
	       SDI_ArrayIndex = 2;
			SDI_Array[SDI_ArrayIndex] = SyncByteIn;
			SDI_ArrayIndex++;

			CLEARBIT(RFID_SS_CONTROL_PORT, RFID_SS_CONTROL_BIT);   //lower the
```

```
slave select line
                RFID_SendMessage(SDO_Array[SDO_ArrayIndex]);
                SDO_ArrayIndex++;
        } else if(SyncByteIn == RFID_END || SDI_ArrayIndex == 6) {
                //we are done with receiving the rfid data
                //condence data
                LastS1Byte = SDI_Array[3];
                LastS2Byte = SDI_Array[4];
                LastS3Byte = SDI_Array[5];

                SDI_ArrayIndex = 0;  //just in case
                SDO_ArrayIndex = 0;  //just in case

                //disable interrupts
                SSPIE = 0;

                //set flags for new state

                //let the event checker know we have new important RFID data
                haveWeReceivedANewRFIDScanFlag = 1;

                //lower the flag that says we are using the sync serial
                areWeUsingTheSyncSerialFlag = 0;

                //set the mode of the sync system
                syncComMode = NOT_BEING_USED_MODE;

                //lower flag
                areWeReceivingARFIDMessageFlag = 0;
        } else {
                SDI_Array[SDI_ArrayIndex] = SyncByteIn;
                SDI_ArrayIndex++;

                CLEARBIT(RFID_SS_CONTROL_PORT, RFID_SS_CONTROL_BIT);   //lower the
slave select line
                RFID_SendMessage(SDO_Array[SDO_ArrayIndex]);
                SDO_ArrayIndex++;
        }

        //prevent the array from overflowing
        if(SDI_ArrayIndex >= (SDI_Array_Length-1)) SDI_ArrayIndex = (SDI_Array_Length-1);
    } else if(syncComMode == COLLECT_SC_DATA_MODE) {
        SyncByteIn = SSPBUF;                //Read in message that we got when we transmitted

        SETBIT(SC_SS_CONTROL_PORT, SC_SS_CONTROL_BIT);   //raise the slave select line

        if(SyncByteIn == SC_START) {
                //set index of incomming SDI to zero
                SDI_ArrayIndex = 0;
                SDI_Array[SDI_ArrayIndex] = SyncByteIn;
                SDI_ArrayIndex++;

                CLEARBIT(SC_SS_CONTROL_PORT, SC_SS_CONTROL_BIT);   //lower the slave
select line
                RFID_SendMessage(SDO_Array[SDO_ArrayIndex]);
                SDO_ArrayIndex++;
```

```
        } else if(SyncByteIn == SC_END) {// && SDI_ArrayIndex >= 4)
                //we are done with receiving the rfid data
                //condence data
                LastSK1Byte     = SDI_Array[1];
                LastSK2Byte     = SDI_Array[2];
                LastSK3Byte     = SDI_Array[3];

                SDI_ArrayIndex = 0;  //just in case
                SDO_ArrayIndex = 0;  //just in case

                //disable interrupts
                SSPIE = 0;

                //set flags for new state

                //let the event checker know we have new important SC data
                haveWeReceivedNewSCDataFlag = 1;

                //lower the flag that says we are using the sync serial
                areWeUsingTheSyncSerialFlag = 0;

                //set the mode of the sync system
                syncComMode = NOT_BEING_USED_MODE;

                //lower flag
                areWeReceivingASCMessageFlag = 0;
        } else {
                SDI_Array[SDI_ArrayIndex] = SyncByteIn;
                SDI_ArrayIndex++;

                CLEARBIT(SC_SS_CONTROL_PORT, SC_SS_CONTROL_BIT);   //lower the slave
select line
                RFID_SendMessage(SDO_Array[SDO_ArrayIndex]);
                SDO_ArrayIndex++;
        }
      } else if(syncComMode == SEND_RFID_TO_SC_MODE) {
        SyncByteIn = SSPBUF;                //Read in message that we got when we transmitted
        SETBIT(SC_SS_CONTROL_PORT, SC_SS_CONTROL_BIT);   //raise the slave select line

        if(SyncByteIn == 0xC1) {
                asm("nop");
        }

        if(SDO_ArrayIndex >= 6) { //4
                //we are done here

                //disable interrupts
                SSPIE = 0;

                //set flags for new state
                //we don't need to let the event check know anything

                //lower the flag that says we are using the sync serial
                areWeUsingTheSyncSerialFlag = 0;

                //set the mode of the sync system
```

```
                syncComMode = NOT_BEING_USED_MODE;

                SDO_ArrayIndex = 0;  //just in case
            } else {
                //otherwise, keep sending data
                CLEARBIT(SC_SS_CONTROL_PORT, SC_SS_CONTROL_BIT);   //lower the slave
select line
                RFID_SendMessage(SDO_Array[SDO_ArrayIndex]);
                SDO_ArrayIndex++;
            }
        }
}

void RFID_setSyncMode(unsigned char mode) {
        SETBIT(SC_SS_CONTROL_PORT, SC_SS_CONTROL_BIT);   //lower the slave select line
        SETBIT(RFID_SS_CONTROL_PORT, RFID_SS_CONTROL_BIT);   //lower the slave select line

        if(mode == COLLECT_RFID_DATA_MODE) {
            //set flag
            areWeReceivingARFIDMessageFlag = 1;

            //set the flag that lets ppl know we are using the sync system
            areWeUsingTheSyncSerialFlag = 1;

            //set which mode we are in
            syncComMode = COLLECT_RFID_DATA_MODE;

            //populate the outgoing sync array
            SDO_Array_Length = A_ARRAY_LENGTH;   //this inits the junk array that we will send to the
RFID pic
            SDO_Array[0] = 0xA1;
            SDO_Array[1] = 0xA2;
            SDO_Array[2] = 0xA3;
            SDO_Array[3] = 0xA4;
            SDO_Array[4] = 0xA5;
            SDO_Array[5] = 0xA6;
            SDO_Array[6] = 0xA7;
            SDO_Array[7] = 0xA8;

            SDI_Array[0] = 0x00;
            SDI_Array[1] = 0x00;
            SDI_Array[2] = 0x00;
            SDI_Array[3] = 0x00;
            SDI_Array[4] = 0x00;
            SDI_Array[5] = 0x00;
            SDI_Array[6] = 0x00;
            SDI_Array[7] = 0x00;

            SDO_ArrayIndex = 0;

            //set the expected size of the incomming RFID array
            SDI_Array_Length = RFID_IN_ARRAY_LENGTH;

            SDI_ArrayIndex = 0;

            CLEARBIT(RFID_SS_CONTROL_PORT, RFID_SS_CONTROL_BIT);   //slave select to low =
```

```
selected

            RFID_SendMessage(SDO_Array[SDO_ArrayIndex]);
            SDO_ArrayIndex++;

            //enable interrupts?
            SSPIE = 1; // enable interrupts from the SPI, which will trigger an interrupt and we will be able to
send the first byte of data
        } else if(mode == COLLECT_SC_DATA_MODE) {
            //set flag
            areWeReceivingASCMessageFlag = 1;

            //set the flag that lets ppl know we are using the sync system
            areWeUsingTheSyncSerialFlag = 1;

            //set which mode we are in
            syncComMode = COLLECT_SC_DATA_MODE;

            //populate the outgoing sync array
            SDO_Array_Length = B_ARRAY_LENGTH;   //this inits the junk array that we will send to the
RFID pic
            SDO_Array[0] = 0xB1;
            SDO_Array[1] = 0xB2;
            SDO_Array[2] = 0xB3;
            SDO_Array[3] = 0xB4;
            SDO_Array[4] = 0xB5;
            SDO_ArrayIndex = 0;

            SDI_Array[0] = 0x00;
            SDI_Array[1] = 0x00;
            SDI_Array[2] = 0x00;
            SDI_Array[3] = 0x00;
            SDI_Array[4] = 0x00;
            SDI_Array[5] = 0x00;
            SDI_Array[6] = 0x00;
            SDI_Array[7] = 0x00;

            //set the expected size of the incomming RFID array
            SDI_Array_Length = SC_ARRAY_LENGTH;

            CLEARBIT(SC_SS_CONTROL_PORT, SC_SS_CONTROL_BIT);   //slave select to low =
selected

            RFID_SendMessage(SDO_Array[SDO_ArrayIndex]);
            SDO_ArrayIndex++;
            //SDI_ArrayIndex++;
            //enable interrupts?
            SSPIE = 1; // enable interrupts from the SPI, which will trigger an interrupt and we will be able to
send the first byte of data
        }else if(mode == SEND_RFID_TO_SC_MODE) {
            //set the flag that lets ppl know we are using the sync system
            areWeUsingTheSyncSerialFlag = 1;

            //set which mode we are in
            syncComMode = SEND_RFID_TO_SC_MODE;
```

```
            //populate the outgoing sync array
            SDO_Array_Length = SC_ARRAY_LENGTH;   //this inits the junk array that we will send to the
RFID pic
            SDO_Array[0] = RFID_START;
            SDO_Array[1] = LastS1Byte;
            SDO_Array[2] = LastS2Byte;
            SDO_Array[3] = LastS3Byte;
            SDO_Array[4] = RFID_END;
            SDO_ArrayIndex = 0;

            SDI_Array[0] = 0x00;
            SDI_Array[1] = 0x00;
            SDI_Array[2] = 0x00;
            SDI_Array[3] = 0x00;
            SDI_Array[4] = 0x00;
            SDI_Array[5] = 0x00;
            SDI_Array[6] = 0x00;
            SDI_Array[7] = 0x00;

            //set the expected size of the incomming RFID array
            SDI_Array_Length = SC_ARRAY_LENGTH;
            SDI_ArrayIndex = 0;

            CLEARBIT(SC_SS_CONTROL_PORT, SC_SS_CONTROL_BIT);   //slave select to low =
selected

            RFID_SendMessage(SDO_Array[SDO_ArrayIndex]);
            SDO_ArrayIndex++;
            //SDI_ArrayIndex++;
            //enable interrupts?
            SSPIE = 1; // enable interrupts from the SPI, which will trigger an interrupt and we will be able to
send the first byte of data
        }
}

void RFID_SendMessage(unsigned char messageByte) {
        SSPBUF     = messageByte;  //this should start the transmission
}
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * RFID Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */

  ///This function requres timer 1 to be setup and running...it is used for delays and such
  //this can be changed later to timer 2 or 0 or ....etc, but not right now

#ifndef _RFID_H_
#define _RFID_H_

#include <htc.h>
#include "defs.h"  //this is for the pin defs

/* Start and End Bytes */
#define SC_START 0x04
#define  SC_END 0x08
#define  RFID_START 0x02
#define RFID_END 0x03

#define NOT_BEING_USED_MODE 1
#define COLLECT_RFID_DATA_MODE 1
#define SEND_RFID_TO_SC_MODE 2
#define COLLECT_SC_DATA_MODE 3

void RFID_initRFIDSync(void);
unsigned char RFID_haveWeReceivedANewRFIDScanFlag(void);
unsigned char RFID_haveWeReceivedNewSCDataFlag(void);

unsigned char RFID_getLastS1(void);
unsigned char RFID_getLastS2(void);
unsigned char RFID_getLastS3(void);

unsigned char RFID_CheckReceivingARFIDMessageFlag(void);
unsigned char RFID_CheckReceivingASCMessageFlag(void);
void RFID_SendMessage(unsigned char messageByte);

void RFID_AssignIsANewRFIDMessageWaitingFlag(unsigned char flagState);
void RFID_AssignIsANewSCMessageWaitingFlag(unsigned char flagState);
void RFID_setSyncMode(unsigned char mode);
void RFID_respondToInterrupt(void);
unsigned char RFID_AreWeSendingSync(void);
unsigned char RFID_haveWeReceivedNewSCDataFlag(void);
//this returns the RFID cards S1 byte (3rd least significant)

unsigned char RFID_getLastSK1(void);
unsigned char RFID_getLastSK2(void);
unsigned char RFID_getLastSK3(void);
```

```
#endif
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Servo Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#include <htc.h>
#include "ServoController.h"
#include "defs.h"

/* Variables */
unsigned char servoPostionHighByte = 2;
unsigned char servoPostionLowByte = 1;

/* Functions */
void SERVO_CONTROLLER_respondToCompareInterrupt(void) {
	unsigned char TimeHighByte;
	unsigned char TimeLowByte;
	static unsigned char servoLineState = 0;

	TimeHighByte = TMR1H;
	TimeLowByte = TMR1L;

	if(servoLineState == 1) { //if the servo line is high, clear the line
	    //power the servo line
	    CLEARBIT(SERVO_OUTPUT_PORT, SERVO_OUTPUT_BIT);
	    //set the next output compare (18+ms);
	    if((0xFF - TimeLowByte) < SERVO_PERIOD_COUNT_LOW) {
			TimeHighByte += (SERVO_PERIOD_COUNT_HIGH + 1);
			TimeLowByte += SERVO_PERIOD_COUNT_LOW;
	    } else {  //otherwise, just set the values
			TimeHighByte += SERVO_PERIOD_COUNT_HIGH;
			TimeLowByte += SERVO_PERIOD_COUNT_LOW;
	    }

	    CCPR1H = TimeHighByte;
	    CCPR1L = TimeLowByte;  //because of stack overflows
	    //set the next timer compare
	    //SERVO_CONTROLLER_setNextCompareTimeValue(TimeHighByte, TimeLowByte);
	    //lower the servo line state flag
	    servoLineState = 0;
	} else {	//if it's low, raise the line
	    //raise the servo line
	    SETBIT(SERVO_OUTPUT_PORT, SERVO_OUTPUT_BIT);
	    //set the next output compare (1-2ms)

	    //this checks for overflow from the low byte (so that the high byte will get incremented by 1)
	    if((0xFF - TimeLowByte) < servoPostionLowByte) {
```

```
                TimeHighByte += (servoPostionHighByte + 1);
                TimeLowByte += servoPostionLowByte;
        } else {  //otherwise, just set the values
                TimeHighByte += servoPostionHighByte;
                TimeLowByte += servoPostionLowByte;
        }

        CCPR1H = TimeHighByte;
        CCPR1L = TimeLowByte;
        //SERVO_CONTROLLER_setNextCompareTimeValue(TimeHighByte, TimeLowByte);
        //raise the servo line flag
        servoLineState = 1;
    }
}

void SERVO_CONTROLLER_setSOSFlagUP(void) {
    servoPostionHighByte = SOS_FLAG_SERVO_UP_HIGH_VALUE;
    servoPostionLowByte = SOS_FLAG_SERVO_UP_LOW_VALUE;
}

void SERVO_CONTROLLER_setSOSFlagDOWN(void) {
    servoPostionHighByte = SOS_FLAG_SERVO_DOWN_HIGH_VALUE;
    servoPostionLowByte = SOS_FLAG_SERVO_DOWN_LOW_VALUE;
}

void SERVO_CONTROLLER_setServoPosition(unsigned char position) {
    unsigned int servoPosition;
    //this is just a rough servo controller based on the specific timer that is being used
    //SERVO_PULSE_WIDTH_MIN is about 625, and the position var can be 0-255,
    //so i multiply it by two and add it to the minimum witdth to get a range from
    // 625 to 625 + 255*4 which is about SERVO_PULSE_WIDTH_MAX (1250) (but a little over)
    servoPosition = SERVO_PULSE_WIDTH_MIN + position * 4;

    //this checks to make sure that the value isn't too high for the servo
    if(servoPosition > SERVO_PULSE_WIDTH_MAX)  servoPosition =
SERVO_PULSE_WIDTH_MAX;

    servoPostionHighByte = servoPosition/0xFF;
  servoPostionLowByte = servoPosition - (servoPostionHighByte * 0xFF);
}

void SERVO_CONTROLLER_initServo(void) {
    servoPostionHighByte = SOS_FLAG_SERVO_DOWN_HIGH_VALUE;
    servoPostionLowByte = SOS_FLAG_SERVO_DOWN_LOW_VALUE;
}
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Servo Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


//this uses the CCP output compare (software) functionality of timer 1.  Timer one must be setup
//at FOSC = 20mhz, prescaler = 1/8, (this is how it is now)

#ifndef _SERVO_CONTROLLER_H_
#define _SERVO_CONTROLLER_H_

#define SERVO_PERIOD_COUNT_HIGH 0x30
#define SERVO_PERIOD_COUNT_LOW 0xD4

#define SERVO_PULSE_WIDTH_MIN 500 //625  //these will change based on the servo used  (this is set
to a reasonable lower limit)
#define SERVO_PULSE_WIDTH_MAX 1400 //1250  //these will changed based on the servo used  (this
is set to a reasonable upper limit)

//Public function prototypes
void SERVO_CONTROLLER_setServoPosition(unsigned char position);
void SERVO_CONTROLLER_initServo(void);
void SERVO_CONTROLLER_respondToCompareInterrupt(void);
void SERVO_CONTROLLER_setSOSFlagUP(void);
void SERVO_CONTROLLER_setSOSFlagDOWN(void);

#endif
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Stepper Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#include <htc.h>
#include "bitdefs.h" // BIT0HI, BIT0LO....definitions
#include "defs.h"
#include "stepper.h"

//Initialize row indexing variable
int row = 0;
static char FullStepArray[4];

/* Prototypes */
static void STEPPER_IncrementTheRow(void);
static void STEPPER_ScaleValues(void);

signed int targetStepValue = 0;
signed int currentStepValue = 0;
unsigned char portOutput = 0x00;
unsigned char adjustMotorFlag = 0;

/* Functions */
void STEPPER_Init(void) {
        STEPPER_InitPorts();
        STEPPER_InitTimer0();
        //numberOfCrabsAboard = 0;
        targetStepValue = 0;
        currentStepValue = 0;
        //crabFlagValue = 0;
        //adjustMotorFlag = 0;
}

void STEPPER_SetCrabCountFlagHeight(unsigned int crabs) {  //0-300 input
        if(crabs > MAX_NUMBER_OF_CRABS_ON_DCH) {
            crabs = MAX_NUMBER_OF_CRABS_ON_DCH;
        }

        targetStepValue = 2*crabs;  //gives us 0-600 scaled (stepper target value)
        adjustMotorFlag = 1;  //flag
        //STEPPER_SetFlagHeight(numberOfCrabsAboard);
}

void STEPPER_respondToTimerOverflow(void) {
        if((currentStepValue < targetStepValue +2) && (currentStepValue > targetStepValue - 2)) {  //are we
reasonably close to the target value
```

```
            //we are close enough
            adjustMotorFlag = 0;  //lower the software flag

            CLEARBIT(STEPPER_INPUT_1_PORT, STEPPER_INPUT_1_BIT);
            CLEARBIT(STEPPER_INPUT_2_PORT, STEPPER_INPUT_2_BIT);
            CLEARBIT(STEPPER_INPUT_3_PORT, STEPPER_INPUT_3_BIT);
            CLEARBIT(STEPPER_INPUT_4_PORT, STEPPER_INPUT_4_BIT);
            //STEPPER_StopMotor();
        } else if (targetStepValue > currentStepValue) {  //target value is bigger...
            //Direction = CLOCKWISE;
            //take one step
            currentStepValue++;
            row = currentStepValue & 0x03;
            portOutput = PORTC & 0b10000111;
            PORTC = portOutput | FullStepArray[row];
        } else if (targetStepValue < currentStepValue) {
            //Direction = COUNTERCLOCKWISE;
            //take one step
            currentStepValue--;
            row = currentStepValue & 0x03;
            portOutput = PORTC & 0b10000111;
            PORTC = portOutput | FullStepArray[row];
        }
}

unsigned char STEPPER_getAdjustStepperMotorFlag(void) {
        return adjustMotorFlag;
}

void STEPPER_InitPorts(void) {
        //Four input lines to the stepper motor
        CLEARBIT(STEPPER_INPUT_1_DDR, STEPPER_INPUT_1_BIT);
        CLEARBIT(STEPPER_INPUT_2_DDR, STEPPER_INPUT_2_BIT);
        CLEARBIT(STEPPER_INPUT_3_DDR, STEPPER_INPUT_3_BIT);
        CLEARBIT(STEPPER_INPUT_4_DDR, STEPPER_INPUT_4_BIT);

        CLEARBIT(PORTC, 3);
        CLEARBIT(PORTC, 4);
        CLEARBIT(PORTC, 5);
        CLEARBIT(PORTC, 6);

        FullStepArray[0] = 0b01010000;  //1010
        FullStepArray[1] = 0b01001000;  //1001
        FullStepArray[2] = 0b00101000;          //0101
        FullStepArray[3] = 0b00110000;          //0110
}

void STEPPER_InitTimer0(void) {
        TMR0 = 0;
        //INTCON = 0;
        OPTION = 0b00000111; //256 prescaler
        //T0IE = 1;
        //GIE = 0;
}
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Stepper Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#ifndef _STEPPER_H_
#define _STEPPER_H_

/* Functions */
//void STEPPER_SetFlagHeight(unsigned short);
void STEPPER_StopMotor(void);
void STEPPER_Init(void);
void STEPPER_InitPorts(void);
void STEPPER_InitTimer0(void);
void STEPPER_SetCrabCountFlagHeight(unsigned int crabs);
unsigned char STEPPER_getAdjustStepperMotorFlag(void);
void STEPPER_respondToTimerOverflow(void);

#endif
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Xbee Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#include "xbee.h"
#include "PirateAsync.h"
#include "BoatStatus.h"

/* XBEE module transmits and receives communications via the XBEE radio transmitters.
 * Takes in a data structure, sends the structure one byte at a time. */

/* Variables */
//these are the RX data buffers, they should be structures, but they aren't yet
unsigned char rxMessageBuffer[RX_BUFFER_SIZE];
unsigned char rxMessageBufferIndex = 0;
unsigned char rxCompleteFlag = 0;
unsigned char rxLengthOfIncommingMessage = 0;
unsigned char rxLengthOfIncommingMessageFrame = 0;

//thse are flags that are used to indicate that a type of message was received
unsigned char MessageBeingRecivedFlag = NO_MESSAGE;
unsigned char FullMessageRecivedFlag = NO_MESSAGE;
unsigned char NeptuneOpenSeasonFlag = 0;
unsigned char NeptuneResponseTypeFlag = NEP_RESP_WAITING_FOR_RESPONSE;
unsigned char HasNeptuneRespondedFlag = 0;
unsigned char LastCrabPotAccumulationRate = 0;
unsigned char LastCrabPotCrabsTranfered = 0;
unsigned char NewDriveCommandFromHCFlag = 0; //only used if we want to respond to the HC

unsigned char LastXORCypherGivenToUs = 0x00;
unsigned char HaveWeGottenANewXORCypher = 0;
unsigned char LastTeamMatesAddressHighByte = 0x00;
unsigned char LastTeamMatesAddressLowByte = 0x00;
unsigned char HaveWeHadATeamMateRequestFlag = 0;

unsigned char tempEvent = 0;

/* Prototypes */

/* Functions */
//Initialize XBEE communications
void XBEE_Init(void) {
        ASYNC_setupUSART();
        TXIE = 0;  //disable TX interrupt
}
```

```
unsigned char XBEE_getLastCrabPotAccumulationRate(void) {
	return LastCrabPotAccumulationRate;
}

unsigned char XBEE_getLastAmountOfCrabTransfered(void) {
	return LastCrabPotCrabsTranfered;
}

unsigned char XBEE_getNeptuneResponseType(void) {
	return NeptuneResponseTypeFlag;
}

unsigned char XBEE_hasNeptuneResponded(void) {
	if(HasNeptuneRespondedFlag != 0) {
	    //lower the flag if the message has been checked
	    HasNeptuneRespondedFlag = 0;
	    return 1;
	} else {
	    return 0;
	}
}

//a check to see if neptune sent open season
unsigned char XBEE_CheckNeptuneOpenSeasonFlag(void) {
	if(NeptuneOpenSeasonFlag != 0) {
	    //lower the flag if the message has been checked
	    NeptuneOpenSeasonFlag = 0;
	    return 1;
	} else {
	    return 0;
	}
}

unsigned char XBEE_CheckHaveWeGottenANewXORCypher(void){
	if(HaveWeGottenANewXORCypher != 0) {
	    //lower the flag if the message has been checked
	    HaveWeGottenANewXORCypher = 0;
	    return 1;
	} else {
	    return 0;
	}
}

unsigned char XBEE_getNewXORCypher(void) {
	return LastXORCypherGivenToUs;
}

unsigned char XBEE_CheckHaveWeHadATeamMateRequestFlag(void){
	if(HaveWeHadATeamMateRequestFlag != 0) {
	    //lower the flag if the message has been checked
	    HaveWeHadATeamMateRequestFlag = 0;
	    return 1;
	} else {
	    return 0;
	}
}
```

```
unsigned char XBEE_getLastTeamMatesAddrHigh(void) {
	return LastTeamMatesAddressHighByte;
}

unsigned char XBEE_getLastTeamMatesAddrLow(void) {
	return LastTeamMatesAddressLowByte;
}

//a check to see if we've received a drive cmd from the HC
unsigned char XBEE_CheckNewDriveCommandFromHCFlag(void) {
	if(NewDriveCommandFromHCFlag != 0) {
	    //lower the flag if the message has been checked
	    NewDriveCommandFromHCFlag = 0;
	    return 1;
	} else {
	    return 0;
	}
}

//this function sends out the "I'm on the XXX team!"  Hopefully someone will respond with the cypher
//if not, we become the team leader...
void XBEE_SendImOnTheFleetMessage(void) {
	unsigned char dataToSendToFleet[3];
	//populate a local array
	dataToSendToFleet[0] = BOAT_STATUS_getMyFleet();
	dataToSendToFleet[1] = FLEET_MSG_TYPE_IM_ON_TEAM;
	dataToSendToFleet[2] = BOAT_STATUS_getMyFleet();

	//start the sending of the message
	ASYNC_sendUSARTmessage(XBEE_populateRadioMessage(BROADCAST_RADIO_ADDRESS,
BROADCAST_RADIO_ADDRESS, dataToSendToFleet, 3));
}

//this function sends out encrypted crab pot data to our fleet and HC's
void XBEE_SendCrabPotInfoToOurFleet(unsigned char crabPotNumber, unsigned char crabPotRate) {
	unsigned char dataToSendToFleet[4];
	//populate a local array
	dataToSendToFleet[0] = BOAT_STATUS_getMyFleet();
	dataToSendToFleet[1] = FLEET_MSG_CRAB_DATA;
	dataToSendToFleet[2] = MAIN_EncodeOrDecodeByteUsingCyper(crabPotNumber);
	dataToSendToFleet[3] = MAIN_EncodeOrDecodeByteUsingCyper(crabPotRate);

	//start the sending of the message
	ASYNC_sendUSARTmessage(XBEE_populateRadioMessage(BROADCAST_RADIO_ADDRESS,
BROADCAST_RADIO_ADDRESS, dataToSendToFleet, 4));
}

//this function sends the cypher key to a specific address
void XBEE_SendCypherKeyToAddress(unsigned char targetAddressHighByte, unsigned char
targetAddressLowByte) {
	unsigned char dataToSendToFleet[3];
	//populate a local array
	dataToSendToFleet[0] = BOAT_STATUS_getMyFleet();
	dataToSendToFleet[1] = FLEET_MSG_TYPE_THE_CYPHER_IS;
	dataToSendToFleet[2] = BOAT_STATUS_getXOR_Cypher();
```

```
        //start the sending of the message
        ASYNC_sendUSARTmessage(XBEE_populateRadioMessage(targetAddressHighByte,
targetAddressLowByte, dataToSendToFleet, 3));
}

//let the HC know what our current crab count is.  YOU NEED TO CHECK TO SEE IF YOU CAN SEND
FIRST!!!
//it used to be the commented out function below without crab pot number, but for debugging purposes it
has it now
//void XBEE_SendStatusMessageToHCfromDCH(unsigned int currentCrabCount, unsigned char
waterloggedFlag) <-- this is how it was, and probably should be after testing

//DataByte0 = DCHHC_STATUS_MESSAGE
//DataByte1+DataByte2 = amount of crab on the DCH
//DataByte3 = are we waterlogged?  0 = no, 1 = yes
//the rest of the data bytes are for debugging
//DataByte4 = the last crab pot number we've visited
//DataByte5 = the last crab pot accumulation rate we've seen
//DataByte6 = the last amout of crabs that were transfered to us

//it would be nice to have one byte that represents an event so that the DCH will know if the boat
//is unloaded/repaired/got a crab pot,  we can send "the important last event" or something like that

void XBEE_SendFakeStatusMessageToHCfromDCH(unsigned int currentCrabCount, unsigned char
waterloggedFlag, unsigned char crabPotNumberToSend, unsigned char rateOfCrabs, unsigned char
amountOfCrabs) {
        unsigned char dataToSend[9];

        //populate a local array
        dataToSend[0] = DCHHC_STATUS_MESSAGE;

        //the next two bytes are the current amount of crab on the boat, add them together on the other
        //end to get the full amount
        if(currentCrabCount>0xFF) {
            dataToSend[1] = (unsigned char) (currentCrabCount-0xFF);
            dataToSend[2] = 0xFF;
        } else {
            dataToSend[1] = 0;
            dataToSend[2] = (unsigned char) currentCrabCount;
        }

        //the fourth byte indicates if the boat is waterlogged
        dataToSend[3] = waterloggedFlag;

        dataToSend[4] = crabPotNumberToSend;
        dataToSend[5] = rateOfCrabs;
        dataToSend[6] = amountOfCrabs;

        tempEvent = BOAT_STATUS_getLastImportantEvent();
        dataToSend[7] = BOAT_STATUS_getLastImportantEvent();
        dataToSend[8] = 34;

        ASYNC_sendUSARTmessage(XBEE_populateRadioMessage(HC_RADIO_ADDRESS_MSB,
HC_RADIO_ADDRESS_LSB, dataToSend, 9));
}
```

```
void XBEE_SendStatusMessageToHCfromDCH(unsigned int currentCrabCount, unsigned char
waterloggedFlag, unsigned char crabPotNumberToSend) {
	unsigned char dataToSend[10];
	unsigned int numberOfCrabsOnBoat;

	//numberOfCrabsOnBoat = BOAT_STATUS_NumberOfCrabsOnboard();

	//populate a local array
	dataToSend[0] = DCHHC_STATUS_MESSAGE;

	//the next two bytes are the current amount of crab on the boat, add them together on the other
	//end to get the full amount
	if(currentCrabCount>0xFF) {
	    dataToSend[1] = (unsigned char) (currentCrabCount-0xFF);
	    dataToSend[2] = 0xFF;
	} else {
	    dataToSend[1] = 0;
	    dataToSend[2] = (unsigned char) currentCrabCount;
	}
	//the fourth byte indicates if the boat is waterlogged
	dataToSend[3] = BOAT_STATUS_getWaterloggedFlag();
	//the fifth byte is the team's XOR_Cypher

	dataToSend[4] = BOAT_STATUS_getLastCrabPotOrDockNumber(); //these are temp  (our data)
	dataToSend[5] = BOAT_STATUS_getLastCrabPotAccumulationRate();  //these are temp  (our data)
	dataToSend[6] = BOAT_STATUS_getLastNumberOfCrabsTransfered(); //these are temp  (our data)
	dataToSend[7] = BOAT_STATUS_getLastImportantEvent();
	dataToSend[8] = BOAT_STATUS_getXOR_Cypher();
	dataToSend[9] = BOAT_STATUS_getMyFleet();

	//i also want to send something that will tell the HC when we've gotten a RFID card scanned, that
way it can beep and
	//let us know via audio that the boat has correctly hit a pot (this will be very helpful when driving)

	//ASYNC_sendUSARTmessage(XBEE_populateRadioMessage(HC_RADIO_ADDRESS_MSB,
HC_RADIO_ADDRESS_LSB, dataToSend, 9));
	ASYNC_sendUSARTmessage(XBEE_populateRadioMessage(HC_RADIO_ADDRESS_MSB,
HC_RADIO_ADDRESS_LSB, dataToSend, 10));
}

//YOU NEED TO CHECK TO SEE IF YOU CAN SEND FIRST!!!
//this function sends a RFID data and security key stuff to neptune, it works
void XBEE_SendNeptuneCrabPotRFID(unsigned char s1, unsigned char s2, unsigned char s3, unsigned
char sk1, unsigned char sk2, unsigned char sk3) {
	unsigned char dataToSendToNeptune[7];

	//populate a local array
	dataToSendToNeptune[0] = NEPTUNE_REQUEST_BYTE;
	dataToSendToNeptune[1] = s1;
	dataToSendToNeptune[2] = s2;
	dataToSendToNeptune[3] = s3;
	dataToSendToNeptune[4] = sk1;
	dataToSendToNeptune[5] = sk2;
	dataToSendToNeptune[6] = sk3;
```

```
        //start the sending of the message
        ASYNC_sendUSARTmessage(XBEE_populateRadioMessage(NEPTUNE_RADIO_ADDR_MSB,
NEPTUNE_RADIO_ADDR_LSB, dataToSendToNeptune, 7));

        //set the neptune responce flag to waiting for response
        //this isn't activly used, but it could be, it is only used after we get a message back, when
        //it is assigned a type
        NeptuneResponseTypeFlag = NEP_RESP_WAITING_FOR_RESPONSE;
        //lower the has neptune responded flag so that we can have an event when neptune responds
        HasNeptuneRespondedFlag = 0;
}

//this function populates a data structure of type usartTxMessage with a message of arbitraty <20 bytes
currently, but this can be changed
//in ASYNC.h
usartTxMessage XBEE_populateRadioMessage(unsigned char MsgAddrHigh, unsigned char
MsgAddrLow, unsigned char MsgData[], unsigned char MsgDataLength) {
        usartTxMessage myMessage;
        unsigned char MsgIndex = 0;
        unsigned char MsgLengthHighByte = 0;
        unsigned char MsgLengthLowByte = 0;
        unsigned char frameID = 0;
        unsigned char checksumValue = 0;

        //frameID = TMR1H;
        frameID = 0x26;
        //MESSAGE LENGTH = FRAME DATA ONLY (NO CHECKSUM)
        MsgLengthLowByte = 5 + MsgDataLength;  //I'm going to say that this will be less than 255

        myMessage.numberOfBytes = MsgDataLength + 9;

        myMessage.messageByte[0] = RADIO_MESSAGE_START_DELIMETER;  //first byte is the start
byte
        myMessage.messageByte[1] = MsgLengthHighByte; //length
        myMessage.messageByte[2] = MsgLengthLowByte;  //length
        myMessage.messageByte[3] = RADIO_MESSAGE_SEND_API_IDENTIFIER;
        myMessage.messageByte[4] = frameID;
        myMessage.messageByte[5] = MsgAddrHigh;
        myMessage.messageByte[6] = MsgAddrLow;
        myMessage.messageByte[7] = 0x00;  //OPTIONS BYTE  (default)

        //messageData
        for(MsgIndex = 0; MsgIndex<MsgDataLength; MsgIndex++) {
            myMessage.messageByte[8+MsgIndex] = MsgData[MsgIndex];
        }

        //calculate the checksum value
        for(MsgIndex = 3; MsgIndex<3+MsgLengthLowByte; MsgIndex++) {
            checksumValue += myMessage.messageByte[MsgIndex];
        }
        checksumValue = 0xFF - checksumValue;
        myMessage.messageByte[8+MsgDataLength] = checksumValue;  //checksum

        return myMessage;
}
```

```
//if we get a private message from neptune, we have to figure out what it means
void XBEE_ProccessPrivateMessageFromNeptune(unsigned char data1, unsigned char complement1,
unsigned char data2, unsigned char complement2) {
	//first we will check to see if the complements are valid
	NeptuneResponseTypeFlag = NEP_RESP_COMPLEMENT_MIS_MATCH;
	if(data1 == NEPTUNE_RESP_INVALID_KEY) {
	   NeptuneResponseTypeFlag = NEP_RESP_INVALID_KEY;
	} else if(data1 == NEPTUNE_RESP_SHIP_UNLOADED) {
	   NeptuneResponseTypeFlag = NEP_RESP_SHIP_UNLOADED;
	} else if(data1 == NEPTUNE_RESP_SHIP_REPAIRED) {
	   NeptuneResponseTypeFlag = NEP_RESP_SHIP_REPAIRED;
	} else {
	   LastCrabPotAccumulationRate = data1;
	   LastCrabPotCrabsTranfered = data2;
	   //set the flag to let other functions know
	   NeptuneResponseTypeFlag = NEP_RESP_CRAB_POT_DATA;
	}
	HasNeptuneRespondedFlag = 1;
}

/* Checks to see if a message is in the process of being transferred. */
//phased out, but left in for backwards compatibility
unsigned char XBEE_IsOutgoingRadioMessageInProgress(void) {
	return ASYNC_areWeSendingAUSARTmessage();
}

/* Checks to see if a message is in the process of being transferred. */
//not used anymore
unsigned char XBEE_HasAFullMessageBeenReceived(void) {
	if(FullMessageRecivedFlag == MESSAGE_RECIVED) {
	   //lower the flag if the message has been checked
	   FullMessageRecivedFlag = NO_MESSAGE;
	   return MESSAGE_RECIVED;
	} else {
	   return NO_MESSAGE;
	}
}

/* Transfers the next byte in the message. When end of message has been reached, will stop sending. */
//not used.
void XBEE_SendNextMessageByte(void) {
	//this will send the data byte in the message buffer that is being pointed to with the index value
	ASYNC_sendNextUSARTTxMessageByte();
}

/* RX */
/* Reads in message byte by byte on the receive register and stores them in the rxMessageBuffer array. */
void XBEE_ReadIncommingAsyncByte(unsigned char rxByte) {
	//rxByte = RCREG;  //read in the data
	if(MessageBeingRecivedFlag == NO_MESSAGE) {
	   //this is the start of a new message
	   //check to see if the start delimiter is appoprate
	   if(rxByte == RADIO_MESSAGE_START_DELIMETER) {
	      //this is a radio async message, and is also the first byte.  what luck!
	      //store the data
	      rxMessageBuffer[0] = RADIO_MESSAGE_START_DELIMETER;
```

```
                //reset the index (we are setting it to 1 because we just stored something in 0)
                rxMessageBufferIndex = 1;

                //raise the message being recived flag
                MessageBeingRecivedFlag = MESSAGE_BEING_RECIVED;

                rxLengthOfIncommingMessage = 12;  //this will get us past the index of the message
length LSB which will update the message length (hopefully)
                rxLengthOfIncommingMessageFrame = 12;
        } else {
                //either we've messed something up, we got bad data, or something else non-radio is
talking on the line
                //error flag
                //don't reset index, store data, or raise flags
        }
    } else {
        //this is the continuation byte of a message (because the message being recived flag is raised)
        //which byte is this?  Length?  Type?  Broadcast/Not?

        //store the message byte
        rxMessageBuffer[rxMessageBufferIndex] = rxByte;  //store the next byte into the incomming
message buffer

        if(rxMessageBufferIndex == INDEX_INCOMMING_MSG_LENGTH_LSB) {
                //I think that just the LSB will be enough to read in the message length
                rxLengthOfIncommingMessageFrame = rxMessageBuffer[rxMessageBufferIndex];
                rxLengthOfIncommingMessage = rxLengthOfIncommingMessageFrame + 4;
        }

        if(rxMessageBufferIndex == rxLengthOfIncommingMessage - 1) { //we have reached the end of
the message (including the checksum)
                //reset things

                //raise and lower flags
                MessageBeingRecivedFlag = NO_MESSAGE;
                FullMessageRecivedFlag = MESSAGE_RECIVED;

                //call something here to parse the data
                //XBEE_InterperateRecivedMessage();
        }else{
                //we are still in the incoming message
                //increment the message index so that the next byte is stored correctly
                 rxMessageBufferIndex++;

                if(rxMessageBufferIndex >= (RX_BUFFER_SIZE)) { //if the index has reached the end
of the array size (the buffer)
                        rxMessageBufferIndex = RX_BUFFER_SIZE - 1;  //prevent us from exceeding
the size of the array
                        //there has been some sort of error if we got here,
                        MessageBeingRecivedFlag = NO_MESSAGE;  //reset things
                }
        }
    }
}
```

```
void XBEE_InterperateReceivedMessage(void) {
	//this function will decide what type of message was recived, and from whom, and extract the imporant data from it

	unsigned char msgLength;
	unsigned char msgFrameLength;
	unsigned char bufferContents[RX_BUFFER_SIZE];
	unsigned char j;
	unsigned char checkSum = 0;

	//we have to store the important data quickly so it doesn't get over-written by a new incomming message, this would be a lot easier
	//if there were structures implemented

	//this won't work because of libs n'such
  //memcpy(bufferContents, rxMessageBuffer, RX_BUFFER_SIZE);
	//for(j=0; j<=msgLength
	msgLength = rxLengthOfIncommingMessage;
	msgFrameLength = rxLengthOfIncommingMessageFrame;

	for(j=0; j<msgLength; j++) {
	    //transfer data over
	    bufferContents[j] = rxMessageBuffer[j];
	    if(j>=INDEX_INCOMMING_API_IDENT) {
			//this adds up the contents of the message frame so that we can check the checksum
			checkSum += bufferContents[j];
	    }
	}

	if(bufferContents[INDEX_INCOMMING_API_IDENT] == RADIO_MESSAGE_RECEIVE_API_IDENTIFIER) { //is the message API identifier correct for an incomming radio message?
	    if(checkSum != 0xFF) {
			//the checksum was wrong...error here, don't pass on the data
			//let someone know that the checksum was wrong
	    } else {
			//the checksum was right, finish the calculations and pass on the data to the approprate people

			//first, what type of message is it?
			//look at source address
			if(bufferContents[INDEX_INCOMMING_SOURCE_ADDR_MSB] == HC_RADIO_ADDRESS_MSB && bufferContents[INDEX_INCOMMING_SOURCE_ADDR_LSB] == HC_RADIO_ADDRESS_LSB) {
				//if the source address of the radio message matches our HC's address

				//is the message private?
				//or broadcast?
				//it doesn't really matter, if it is from our HC, it probably is private and it probably
				//is a control command
				//which, the motor controller pics should be responding to anyways....
				//the motor pics are listening in on the RX line from the XBEE
				//so we don't do anything
				if(bufferContents[INDEX_INCOMMING_START_OF_DATA] == HCDCH_DRIVE_CMD_MESSAGE) {
```

```
                                        NewDriveCommandFromHCFlag = 1;
                                }
                        } else if(bufferContents[INDEX_INCOMMING_SOURCE_ADDR_MSB] ==
NEPTUNE_RADIO_ADDR_MSB && bufferContents[INDEX_INCOMMING_SOURCE_ADDR_LSB]
== NEPTUNE_RADIO_ADDR_LSB) {
                                //if the source address of the radio message matches neptunes address
                                //is the message private?
                                if((bufferContents[INDEX_INCOMMING_OPTIONS_BYTE] & 0x02) == 0) {
                                        //it is a private message, therefor it is either a crab pot respoce, a repair
dock responce, or a drydock responce

                                        //causing stack overflow

        XBEE_ProccessPrivateMessageFromNeptune(bufferContents[INDEX_INCOMMING_START_OF_
DATA], bufferContents[INDEX_INCOMMING_START_OF_DATA+1],
bufferContents[INDEX_INCOMMING_START_OF_DATA+2],
bufferContents[INDEX_INCOMMING_START_OF_DATA+3]);

                                        //first we will check to see if the complements are valid
                                } else {
                                //or broadcast?
                                        //it is most likely open-season, is it?
                                        if(msgFrameLength-5 == 3) {
                                                //if it has three bytes in the data packet, it's even more likely to
be open-season

        if(bufferContents[INDEX_INCOMMING_START_OF_DATA] ==
NEPTUNE_OPEN_SEASON_BYTE_ONE &&

bufferContents[INDEX_INCOMMING_START_OF_DATA+1] ==
NEPTUNE_OPEN_SEASON_BYTE_TWO &&

bufferContents[INDEX_INCOMMING_START_OF_DATA+2] ==
NEPTUNE_OPEN_SEASON_BYTE_THREE) {
                                                        //THERE WAS AN OPEN SEASON COMMAND
FROM NEPTUNE!!
                                                        //set the open season flag
                                                        NeptuneOpenSeasonFlag = 1;
                                                }
                                        }
                                }
                        } else {
                                //the message came from someone else this will mostly be ignored, but will be
important during
                                //team setup

                                //is the message private?
                                if((bufferContents[INDEX_INCOMMING_OPTIONS_BYTE] & 0x02) == 0) {
                                        if(bufferContents[INDEX_INCOMMING_START_OF_DATA] ==
BOAT_STATUS_getMyFleet()) { // && BOAT_STATUS_getBoatStatus().myFleet != WHITE)
                                                //if the incomming private message matches our fleet and our
fleet isn't white
                                                //then it is probably a fleetmate giving us the XOR cypher key

        if(bufferContents[INDEX_INCOMMING_START_OF_DATA+1] ==
FLEET_MSG_TYPE_THE_CYPHER_IS) {
```

```
                                                //yes, it is a teammate giving us the cypher
                                                //store the cypher locally
                                                LastXORCypherGivenToUs =
bufferContents[INDEX_INCOMMING_START_OF_DATA+2];
                                                //set a flag to let the event checker know that we've
gotten a new XOR cypher
                                                HaveWeGottenANewXORCypher = 1;
                                        }
                                }
                        } else { //or broadcast?
                                if(bufferContents[INDEX_INCOMMING_START_OF_DATA] ==
BOAT_STATUS_getMyFleet()) { // && BOAT_STATUS_getBoatStatus().myFleet != WHITE)
                                        //if the incomming broadcast message matches our fleet and
our fleet isn't white
                                        //then it is probably a fleetmate asking for the xor cypher or, it
is someone broadcasting pot information
                                        //if it is a fleetmate asking for the XOR cypher, we should give
it to him, or at least let the event
                                        //checker know so it can give it it him
                                        //otherwise, it is probably a fleet crabpod data broadcast, in
which case we need can ignore it.  Our
                                        //HC will listen to it and update the LCD screen appropratly

        if(bufferContents[INDEX_INCOMMING_START_OF_DATA+1] ==
FLEET_MSG_TYPE_IM_ON_TEAM) {
                                                //yes, it is a teammate telling us they are on our team,
and would like the cypher

                                                //store the teammate's address locally
                                                LastTeamMatesAddressHighByte =
bufferContents[INDEX_INCOMMING_SOURCE_ADDR_MSB];
                                                LastTeamMatesAddressLowByte =
bufferContents[INDEX_INCOMMING_SOURCE_ADDR_LSB];

                                                //set a flag to let the event checker know that this has
happened
                                                HaveWeHadATeamMateRequestFlag = 1;
                                        }
                                }
                        }
                }
        } //end checksum check else
    } else if (bufferContents[INDEX_INCOMMING_API_IDENT] ==
RADIO_MESSAGE_SEND_STATUS_API_IDENTIFIER) {  //end is the message API identifier correct
for an incomming radio message?
        //this checks to see if we got a status message back from the XBEE radio telling us that the
message
        //sending was successful or not successful

        //if it isn't successful, re-send whatever message you last sent
        if(bufferContents[INDEX_INCOMMING_API_IDENT+2] != 0) { //if it doesn't = 0 which means
successful transmission
                //send the last message again, or in this case, just re-send a status message, because that's
probably what we last sent (and is most important to us)
                MAIN_assignDoWeNeedToSendAnUpdateToTheHCFlag(1);  //update the crab count to
HC
```

```
            }
        } //end radio API if statement
}
```

```
/*
 * Team Seconds from Disaster
 * 5.25.10
 * ME218c
 * Central PIC
 * Xbee Module
 * Language: C
 *
 * Blake English
 * Nina Joshi
 * Andi Kleissner
 */


#ifndef _XBEE_H_
#define _XBEE_H_

#include <htc.h>
#include "defs.h"
#include "PirateAsync.h"

//Messages between the HC and DCH or vicaversa
#define HCDCH_DRIVE_CMD_MESSAGE 0xF1
#define DCHHC_STATUS_MESSAGE 0xF2
#define DCHHC_SECURITY_KEY_MESSAGE 0xF4
#define DCHHC_PRIVATE_CRAB_MESSAGE 0xF3

#define RX_BUFFER_SIZE 22

//flags for something
#define NO_MESSAGE 0
#define MESSAGE_RECIVED 1
#define MESSAGE_BEING_RECIVED 2

#define RADIO_MESSAGE_START_DELIMETER 0x7E

//Radio message API identifiers
#define RADIO_MESSAGE_SEND_API_IDENTIFIER 0x01
#define RADIO_MESSAGE_SEND_STATUS_API_IDENTIFIER 0x89
#define RADIO_MESSAGE_RECEIVE_API_IDENTIFIER 0x81

//incoming radio message defines
#define INDEX_INCOMMING_START_DELIMETER 0
#define INDEX_INCOMMING_MSG_LENGTH_MSB 1
#define INDEX_INCOMMING_MSG_LENGTH_LSB 2
#define INDEX_INCOMMING_API_IDENT 3
#define INDEX_INCOMMING_SOURCE_ADDR_MSB 4
#define INDEX_INCOMMING_SOURCE_ADDR_LSB 5
#define INDEX_INCOMMING_SIGNAL_STRENGTH 6
#define INDEX_INCOMMING_OPTIONS_BYTE 7
#define INDEX_INCOMMING_START_OF_DATA 8

//important neptune data defines
#define NEPTUNE_OPEN_SEASON_BYTE_ONE 0x47
#define NEPTUNE_OPEN_SEASON_BYTE_TWO 0x6F
#define NEPTUNE_OPEN_SEASON_BYTE_THREE 0x21
```

```
#define NEPTUNE_REQUEST_BYTE 0x66

#define NEPTUNE_RESP_INVALID_KEY 253
#define NEPTUNE_RESP_SHIP_UNLOADED 254
#define NEPTUNE_RESP_SHIP_REPAIRED 255

//types of neptune responce that the main program can see
#define NEP_RESP_WAITING_FOR_RESPONSE 0
#define NEP_RESP_CRAB_POT_DATA 1
#define NEP_RESP_INVALID_KEY 2
#define NEP_RESP_SHIP_UNLOADED 3
#define NEP_RESP_SHIP_REPAIRED 4
#define NEP_RESP_COMPLEMENT_MIS_MATCH 5

#define FLEET_MSG_TYPE_IM_ON_TEAM 0x01
#define FLEET_MSG_TYPE_THE_CYPHER_IS 0x02
#define FLEET_MSG_CRAB_DATA 0x03

/*RADIO ADDRESSES*/
#define BROADCAST_RADIO_ADDRESS 0xFF

#define HC_RADIO_ADDRESS_MSB 0x20
#define HC_RADIO_ADDRESS_LSB 0x86

#define DCH_RADIO_ADDRESS_MSB 0x21
#define DCH_RADIO_ADDRESS_LSB 0x86

#define NEPTUNE_RADIO_ADDR_MSB 0x77
#define NEPTUNE_RADIO_ADDR_LSB 0x77

/* Variables */

/* Functions */
void XBEE_Init(void);
unsigned char XBEE_getLastCrabPotAccumulationRate(void);
unsigned char XBEE_getLastAmountOfCrabTransfered(void);
unsigned char XBEE_getNeptuneResponseType(void);
unsigned char XBEE_hasNeptuneResponded(void);
void XBEE_SendNeptuneCrabPotRFID(unsigned char s1, unsigned char s2, unsigned char s3, unsigned
char sk1, unsigned char sk2, unsigned char sk3);
usartTxMessage XBEE_populateRadioMessage(unsigned char MsgAddrHigh, unsigned char
MsgAddrLow, unsigned char MsgData[], unsigned char MsgDataLength);
void XBEE_ProccessPrivateMessageFromNeptune(unsigned char data1, unsigned char complement1,
unsigned char data2, unsigned char complement2);
unsigned char XBEE_IsOutgoingRadioMessageInProgress(void);
unsigned char XBEE_HasAFullMessageBeenReceived(void);
unsigned char XBEE_CheckNeptuneOpenSeasonFlag(void);
void XBEE_SendNextMessageByte(void);
void XBEE_ReadIncommingAsyncByte(unsigned char rxByte);
void XBEE_InterperateReceivedMessage(void);
void XBEE_SendStatusMessageToHCfromDCH(unsigned int currentCrabCount, unsigned char
waterloggedFlag, unsigned char crabPotNumberToSend);
unsigned char XBEE_CheckNewDriveCommandFromHCFlag(void);

void XBEE_SendFakeStatusMessageToHCfromDCH(unsigned int currentCrabCount, unsigned char
```

```
waterloggedFlag, unsigned char crabPotNumberToSend, unsigned char rateOfCrabs, unsigned char
amountOfCrabs);
void XBEE_SendCrabPotInfoToOurFleet(unsigned char crabPotNumber, unsigned char crabPotRate);
void XBEE_SendCypherKeyToAddress(unsigned char targetAddressHighByte, unsigned char
targetAddressLowByte);
void XBEE_SendImOnTheFleetMessage(void);
unsigned char XBEE_CheckHaveWeGottenANewXORCypher(void);
unsigned char XBEE_getNewXORCypher(void);
unsigned char XBEE_CheckHaveWeHadATeamMateRequestFlag(void);
unsigned char XBEE_getLastTeamMatesAddrHigh(void);
unsigned char XBEE_getLastTeamMatesAddrLow(void);

#endif
```