Weiner Meister Final Report
ME218c – June 6th, 2008
Design Overview

Game Strategy:
Our strategy for the game was to have a fast and maneuverable boat with immense long-range water dispensing capabilities. For water dispensing we chose to use a long distance squirting apparatus versus a short-range dumping apparatus. This combination of long range water dispensing with our agile boat allows us to be a strong offensive and defensive force during game play.
Our two-independent-propeller drive train worked in tandem with our “bun-shaped” hull allowing us to achieve this goal of combined speed and maneuverability. For water dispensing, the boat featured two bilge pumps which directed the water through two nozzles mounted to the front of our craft. The boat also featured a sealed midsection to house all of the electronics.
The hull is constructed of closed-cell foam which was carved into its delicious bun shape. The underside of the hull is shaped so that there are two pontoons on either side to help stabilize the boat. The hull also features cutouts on the underside to house the drive train components as well as the bilge pumps for water dispensing.
Our drive consists of two independent dc motors each driving their own propellers. The motors are housed inside sealed PVC pipe and have drive shafts exiting through grease-sealed bushings. These drive shafts are connected to RC boat propellers. The use of the two independent motors allows the boat to go backwards, forwards and rotate in place.
Water dispensing

The boat features two 500 gallon per hour bilge pumps that provide some serious water dispensing capabilities. The pumped water runs up to the top of our boat through vinyl tubing and is then forced through a brass nozzle. The nozzle was sized such that it gives us a fairly long range (~12 feet) without compromising our sizable volumetric flow.

Board mounting/Water proofing

All of our circuitry is mounted on a masonite board which easily slips in and out of a piece of PVC pipe. The masonite board allows the electronics to sit in the middle of the
pipe so that if water were to enter the pipe it would sit in a pool below the electronics. The PVC pipe has a sealed cap on one end and a removable one on the other. The PVC pipe features a small slit on the side which allows us to route the wiring into the pipe. The slit is located such that when the cap is pressed on, the wiring is compressed to eliminate any air gaps for water to get in.
Helm

The helm is designed to simulate the classic barbeque experience. There is an actual barbeque incorporated into the helm as a well barbeque utensil and condiments. The user manipulates the barbeque utensil and condiments to maneuver the boat and to squirt water. The helm also communicates information to the captain through the use of dynamic graphical indicators which are controlled by servos. There is also a siren to indicate when the boat is “stood down”.
Indicators

The helm uses servo controlled indicators to communicate the current active base as well as the number of the boat the helm is currently controlling. The servos are mounted underneath the main table surface and have arrows mounted to them. The servos rotate the arrows to various positions to point at the necessary graphic to convey the appropriate information.

Board mounting

The boards are mounted to a laser-cut piece of masonite which fits inside of the barbeque. The masonite board features openings to allow wiring to be easily mounted down and out of the barbeque.
Sensors/Inputs

The speed and direction of the boat are controlled using a barbeque utensil derived joystick. A barbeque spatula was cut and then brazed to a threaded piece of brass. This allows the spatula handle to be threaded onto the joystick. The joystick is a store bought item which essentially consists of two potentiometers mounted to a threaded piece of rod. Mounting the spatula handle to the joystick allows the captain to use the spatula handle to maneuver the boat in an intuitive way (pushing the spatula forward makes the boat go forward, etc.).

The water dispensing is activated by shaking a container of mustard. The mustard bottle contains a weighted bare wire inside which makes contact with an aluminum tube when the mustard container is shaken. When contact is made the microprocessor sends a signal to the boat telling it to squirt water.
The helm also features two “special” buttons. They are simply two push buttons which can be used to control “special” features the boats may contain. For example, our boat had a siren which could be activated using these special buttons.
**ELECTRICAL DESIGN**

**Boat JP5 Board - iButton and XBee**

This board has connections for the E128 to communicate with the Xbee board provided to us, and has a PIC circuit for iButton reading. Note that we had to jumper two connections on the Xbee board to allow the header input to communicate with the Xbee module without a PIC on the Xbee board.

We opted to use a PIC for reading the iButton since it is easier to control the timing when the device is dedicated to just this function. The iButton uses a 1-wire communication scheme that requires controlling and then reading the same line. The E128 indicated to the PIC that it wanted an iButton number by lowering a single enable line. The PIC indicates to the user that it wants to read an iButton by flashing the LED in the center of the reader. When it successfully reads the serial number, it then uses SPI to transfer the required bytes to the E128. Once the E128 has verified that there is a matched serial number, it raises the enable line to end this stage. This satisfies the project requirement that either the boat or helm must have two actively communicating processors.

**Boat JP5 Board - iButton and XBee**

<table>
<thead>
<tr>
<th>JP2</th>
<th>1</th>
<th>GND</th>
<th>20</th>
<th>GND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>PS2</td>
<td>19</td>
<td>PP5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>PS3</td>
<td>18</td>
<td>PP4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>PP2</td>
<td>17</td>
<td>PP3</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>PP0</td>
<td>16</td>
<td>PP1</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>P0D</td>
<td>15</td>
<td>P04</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>P0D</td>
<td>14</td>
<td>P05</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>P0D</td>
<td>13</td>
<td>P06</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>P0D</td>
<td>12</td>
<td>P07</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>GND</td>
<td>11</td>
<td>GND</td>
</tr>
</tbody>
</table>

5V from E128

<table>
<thead>
<tr>
<th>JP3</th>
<th>1</th>
<th>GND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>XBBE DOUT</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>XBBE DIN</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>iButton Select</td>
</tr>
</tbody>
</table>

XBBE

---

**Boat JP5 Board - iButton and XBee**

<table>
<thead>
<tr>
<th>JP4</th>
<th>1</th>
<th>GND</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>TX/RX</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>DATA</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>LED+</td>
</tr>
</tbody>
</table>

**iButton Reader PIC1**

Vdd 20

1 | RA5/OSC1 | RA0/AN0 |
2 | RA4/AN3/OSC2 | RA1/AN1 |
3 | RA3/MCLR | RA2/AN2 |
4 | RC5/CP1 | RC0/AN4 |
5 | RC4/C2OUT | RC1/AN5 |
6 | RC3/AN7 | RC2/AN6 |
7 | RC6/SS | RB4/SDI |
8 | RC7/SDO | RB5/RX |
9 | SS | MOSI |
10| PS7/SS | SCK |

**iButton JP5**

GND 1

1 | PS5/SCK |
2 | SCK |
3 | SS |
4 | PS6/SCK |
5 | PS4/MISO |
6 | PS4/MISO |
7 | PS5/MISO |
8 | PS7/SS |
9 | SS |
10| PS6/SCK |

PIC16F876

---

**Boat JP5 Board - iButton and XBee**

1 | Vdd |
2 | RA0/AN0 |
3 | RA1/AN1 |
4 | RA2/AN2 |
5 | RC0/AN4 |
6 | RC1/AN5 |
7 | RC2/AN6 |
8 | RB4/SDI |
9 | RB5/RX |
10| SS |
11| MOSI |
12| SCK |
13| MISO |
14| MISO |
15| MISO |
16| SCK |
17| TX/RX |
18| DATA |
19| LED+ |
20| LED+ |
Boat JP6 Board - Outputs:
This board has connections for the PWM signal to the motor drivers and the digital outputs for water and special functions. Very simple, indeed.

Boat JP6 Board - Outputs

<table>
<thead>
<tr>
<th>JP10</th>
<th>JP11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>GND</td>
</tr>
<tr>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

Power to E128 JP13

Signals JP16

Rmotor PWM
Rmotor Dir
TeamBlue
Water
GND

Rmotor PWM
Rmotor Dir
TeamBlue
Water
GND

TeamRed
Lmotor Dir
Lmotor PWM
Lmotor
**Boat Power Board:**
This board has a block of screw terminals for connecting the batteries, power switch, and power to the motor drivers. It also has a block of power MOSFETs that each control a bilge pump or light, according to the signals received from the JP6 board.
**Vessel - E128**

<table>
<thead>
<tr>
<th>JP6 24 Pin</th>
<th>RIBBON CABLE --&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>NC</td>
<td>UC1</td>
</tr>
<tr>
<td>PU7</td>
<td>UC2</td>
</tr>
<tr>
<td>PU6</td>
<td>UC3</td>
</tr>
<tr>
<td>PT0</td>
<td>UC4</td>
</tr>
<tr>
<td>PT1</td>
<td>UC5</td>
</tr>
<tr>
<td>PT2</td>
<td>UC6</td>
</tr>
<tr>
<td>PT3</td>
<td>UC7</td>
</tr>
<tr>
<td>PT4</td>
<td>UC8</td>
</tr>
<tr>
<td>PT5</td>
<td>UC9</td>
</tr>
<tr>
<td>PT6</td>
<td>UC10</td>
</tr>
<tr>
<td>PE0</td>
<td>UC11</td>
</tr>
<tr>
<td>NC</td>
<td>UC12</td>
</tr>
</tbody>
</table>

**--- RIBBON CABLE**

<table>
<thead>
<tr>
<th>JP5 20 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
</tr>
<tr>
<td>GND</td>
</tr>
<tr>
<td>PS2</td>
</tr>
<tr>
<td>PS3</td>
</tr>
<tr>
<td>PP2</td>
</tr>
<tr>
<td>PP0</td>
</tr>
<tr>
<td>PAD0</td>
</tr>
<tr>
<td>PAD1</td>
</tr>
<tr>
<td>PAD2</td>
</tr>
<tr>
<td>PAD3</td>
</tr>
<tr>
<td>GND</td>
</tr>
</tbody>
</table>

**--- KEY**

<table>
<thead>
<tr>
<th>JP2 5 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Use</strong></td>
</tr>
<tr>
<td>PS7/SS</td>
</tr>
<tr>
<td>PS6/SCK</td>
</tr>
<tr>
<td>GND</td>
</tr>
<tr>
<td>PS5/MOSI</td>
</tr>
<tr>
<td>PS4/MISO</td>
</tr>
</tbody>
</table>
Helm JP5 Board – iButton, Xbee, Joystick:
This board is virtually a copy of the boat iButton and Xbee board. Since the JP5 connector on the E128 also has the analog input port, this includes a connection for the joystick.
**Helm JP6 Board – Inputs and Servos:**
This board has inputs for the various buttons on the helm, and outputs to control the servos that indicate team number and active base. A MOSFET is used to power the siren.
**Helm Power Management:**
The helm is powered by two 7.2 V NiCad batteries which are wired in parallel. The output of the batteries runs through a 10 Amp fuse to protect our microprocessor and circuits in case something goes awry. There is also a switch which is used to power on and off the entire electrical system. Voltage regulation is handled by the voltage regulator on the E128 protection board.

Our calculations show that our helm is capable of running for **15 hours** off of these two batteries. Each of the batteries provides 1500 mAh and the average current draw of our entire electrical system is 200 mA. Thus our runtime is 15 hours (3000 mAh / 200 mA).
## Helm E128 Pin Table

### Helm - E128

<table>
<thead>
<tr>
<th>Use</th>
<th>Name</th>
<th>Pin Number</th>
<th>Pin Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP6 24 Pin</td>
<td>RIBBON CABLE --&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NC</td>
<td>Name</td>
<td>Pin Number</td>
<td>Pin Number</td>
</tr>
<tr>
<td></td>
<td>Pin Number</td>
<td>Pin Number</td>
<td>Name</td>
</tr>
<tr>
<td>NC</td>
<td>1</td>
<td>24</td>
<td>NC</td>
</tr>
<tr>
<td>PU7</td>
<td>2</td>
<td>23</td>
<td>GND</td>
</tr>
<tr>
<td>PU6</td>
<td>3</td>
<td>22</td>
<td>PU5</td>
</tr>
<tr>
<td>Special Button 0 (IN)</td>
<td>PT0</td>
<td>4</td>
<td>21</td>
</tr>
<tr>
<td>Special Button 1 (IN)</td>
<td>PT1</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>WATER BUTTON (IN)</td>
<td>PT2</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>RESET BUTTON (IN)</td>
<td>PT3</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>STAND DOWN (OUT)</td>
<td>PT4</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>PT5</td>
<td>9</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>PT6</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>PE0</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>NC</td>
<td>12</td>
<td>13</td>
</tr>
</tbody>
</table>

### &lt;-- RIBBON CABLE

<table>
<thead>
<tr>
<th>Use</th>
<th>Name</th>
<th>Pin Number</th>
<th>Pin Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP5 20 Pin</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Name</td>
<td>Pin Number</td>
<td>Pin Number</td>
</tr>
<tr>
<td>GND</td>
<td>1</td>
<td>20</td>
<td>GND</td>
</tr>
<tr>
<td>XBEE DOUT (IN)</td>
<td>PS2</td>
<td>2</td>
<td>19</td>
</tr>
<tr>
<td>XBEE DIN (OUT)</td>
<td>PS3</td>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>Ibutton Select</td>
<td>PP2</td>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>PP0</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>STEERING (IN)</td>
<td>PAD0</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>SPEED (IN)</td>
<td>PAD1</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>PAD2</td>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>PAD3</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>GND</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

### &lt;-- KEY

<table>
<thead>
<tr>
<th>Use</th>
<th>Name</th>
<th>Pin Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP2 5 Pin</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Name</td>
<td>Pin Number</td>
</tr>
<tr>
<td>PS7/SS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>PS6/SCK</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PS5/MOSI</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PS4/MISO</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Selected component values and calculations:

Power MOSFETs (IRLZ34N)
Max voltage: 55V
Max current: 30A
Bilge pump draw: 2.5A
Indicator light draw: 100mA
Siren draw: 100mA
⇒ All devices are well under the max capacity, even without heat sinks.

Motor Drivers (TLE5206-2):
Rated continuous current: 5 A
Rated peak current: 6 A
Max Stall Current:
Motor coil resistance: 1.8 ohms
Max supply voltage: ~7 V
Max stall current (V/R): ~4 A
⇒ Under limit. In practice, current draw was about 2A.

Regulator on E128 Board:
Max current: 1 A
Average current draw for Helm: 200 mA
E128-powered devices on boat: 250 mA
⇒ All is good.
## Bill of Materials*

### Boat:

<table>
<thead>
<tr>
<th>Description</th>
<th>Quantity</th>
<th>Unit Cost</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/2 sheet Pink Foam</td>
<td>1</td>
<td>$10.00</td>
<td>$10.00</td>
</tr>
<tr>
<td>2 ft section 4&quot; ABS pipe</td>
<td>1</td>
<td>$4.95</td>
<td>$4.95</td>
</tr>
<tr>
<td>4&quot; ABS end cap</td>
<td>2</td>
<td>$6.58</td>
<td>$13.16</td>
</tr>
<tr>
<td>4mm Motor shaft</td>
<td>2</td>
<td>$10.55</td>
<td>$21.10</td>
</tr>
<tr>
<td>4mm Universal Joint</td>
<td>2</td>
<td>$7.65</td>
<td>$15.30</td>
</tr>
<tr>
<td>35mm, 2 blade propeller</td>
<td>2</td>
<td>$3.60</td>
<td>$7.20</td>
</tr>
<tr>
<td>Shaft grease (waterproofing)</td>
<td>1</td>
<td>$3.29</td>
<td>$3.29</td>
</tr>
<tr>
<td>500 GPH Bilge Pump</td>
<td>2</td>
<td>$15.00</td>
<td>$30.00</td>
</tr>
<tr>
<td>1-1/4&quot; PVC pipe</td>
<td>1</td>
<td>$5.00</td>
<td>$5.00</td>
</tr>
<tr>
<td>Maxon A-max 6V motor</td>
<td>2</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Perf Board</td>
<td>3</td>
<td>$2.95</td>
<td>$8.85</td>
</tr>
</tbody>
</table>

### Helm:

<table>
<thead>
<tr>
<th>Description</th>
<th>Quantity</th>
<th>Unit Cost</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weber Grill</td>
<td>1</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Spatula</td>
<td>1</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Mustard Bottle</td>
<td>1</td>
<td>$2.49</td>
<td>$2.49</td>
</tr>
<tr>
<td>Servos</td>
<td>3</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Perf Board</td>
<td>2</td>
<td>$2.95</td>
<td>$5.90</td>
</tr>
</tbody>
</table>

### Odds and Ends:

<table>
<thead>
<tr>
<th>Description</th>
<th>Quantity</th>
<th>Unit Cost</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wire of various gages</td>
<td>1</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Paint and stuff</td>
<td>1</td>
<td>$0.00</td>
<td>$0.00</td>
</tr>
<tr>
<td>Lots of molex</td>
<td>1</td>
<td>$5.00</td>
<td>$5.00</td>
</tr>
<tr>
<td>Switches and Lights</td>
<td>1</td>
<td>$10.00</td>
<td>$10.00</td>
</tr>
</tbody>
</table>

| Grand Total             |          |           | $142.24 |

* cost = $0 denotes part was donated, found, stolen, or otherwise acquired
SOFTWARE DESIGN

Software Overview

We thanked ourselves everyday for our decision to use an E128 as the primary processor on both the boat and the helm. This allowed us to do the vast majority of our programming in C (versus the assembly language used on the PICs). Programming in C made it much easier to create and debug robust state machines for both the helm and boat. We were also able to share a lot of the code between the boat and the helm which made it much easier to make system wide changes.

Our boat and helm implemented the state machines that were created by the communication committee. The committee essentially laid out exactly how our software needed to behave so all we had to do was implement it. Below is a description of how the helm and boat behave during game play. This is lifted from the document the communications committee created and is what we used as a reference when coding.

Pairing

When powered up, the helms and craft enter a ‘waiting for iButton’ state. Once a helm detects an iButton, the serial number is read and the helm enters the ‘waiting for sync’ state. Once in this state, the helm attempts to sync with a craft by repeatedly broadcasting an IBUTTON message containing the identity of the read iButton. Similarly, crafts also power up into a ‘waiting for iButton’ state. Once a craft reads an iButton, it transitions to a ‘waiting for sync’ state in which it monitors all broadcast messages sent by the helms.

When a craft receives an IBUTTON broadcast message, it checks the iButton identification data. If this data matches the iButton read by the craft, it transitions into the ‘game’ state. During this transition, the craft stores the address of the helm that sent the message and sends a MATCHED message in response to the helm. This message informs the helm which craft it is controlling for the remainder of the round. In the future, the helm may only send commands to that particular craft and the craft must only respond to commands from that helm and the admirals.

Pre-Game Operation

After successfully pairing, both the helm and craft should check the serial number from the iButton and display their team affiliation (odd serial numbers are for red team and even serial numbers are for blue teams). While this display could be performed immediately upon receiving an iButton serial number, it should be activated only when the craft has successfully paired with its helm. This display will indicate not only affiliation, but also, a successful pairing. Furthermore, after pairing, the helm must indicate to the helmsperson the number of the craft it is controlling.
The helm should at this point transition to a ‘wait for game’ state. While in this state the helm should send NO_ACTION commands at a rate of 5Hz in order to maintain the RF link with the craft and prevent it from moving. Meanwhile, the craft will transition to a ‘game’ state.

Helms remain in the ‘waiting for game’ state until the admirals broadcast the START_OF_GAME command. Once this command is issued, the game begins and helms may commence sending COMMAND signals to craft. Each COMMAND signal indicates the desired speed, direction, water delivery status (on/off), and all other special commands (see table of commands for more details). Admirals must then transmit a RED/BLUE_GOAL command to specify which goal is active.

**Admiral Commands**

During the game, admirals may send STAND_DOWN or HARD / SOFT_RESET commands directly to any craft. When a STAND_DOWN command is received, the craft must cease all activity and pass this command along to the helm. It is encouraged, but not required, that the craft also visibly communicate that it is disabled to observers of the match. The craft must take no action other than retransmitting STAND_DOWN_RECEIVED to the helm until the helm acknowledges the transmission. Once the helm receives and acknowledges the STAND_DOWN_RECEIVED command, it should send NO_ACTION commands until the stand down period expires (10 seconds) and display that the craft has been stood down to the helmsperson.

When a craft receives a HARD_RESET command from the Admiral, it must stop what it is doing, reset its team association, disassociate from its helm, and return to the initial ‘waiting for iButton’ state. A SOFT_RESET command is similar to a HARD_RESET command except that the craft remains in the ‘game’ state and does not disassociate from its paired helm. In other words, after receiving a SOFT_RESET command, the craft shut off all actuators, water, and special abilities then listens for new commands from its previously paired helm.

Admirals may also broadcast a RED/BLUE_GOAL broadcast command to all helms/craft. Upon, receiving this command, all helms must indicate the active base and players must move their craft to the proper side of the playfield, or be subject to a STAND_DOWN command.

Admirals may also PING craft or helms in order to determine the state and pairing of each craft/helm. Upon receiving a PING command, crafts and helms must respond with their state and pairing.

At the end of the game, the admirals will issue an END_GAME command. Upon receiving this command, helms will cease all water delivery and may only send motion commands directing craft back to the starting area. After all craft have returned, the admiralty will broadcast a HARD_RESET command.
to all craft and helms in order to break all helm/craft pairings. Helms and craft
should both respond to this command by disassociating from their paired
craft/helm, resetting all visible team/craft affiliations, and returning to the
‘waiting for iButton’ state.

**Communications Failure**

If, at any time, the helm and craft fall out of communication (no packets
are received by the craft) for a period of greater than three seconds, the craft must
turn off all actuators, water, and special functions. This is to prevent damage from
and to the uncontrolled craft. Unless a **HARD_RESET** command is issued, the
craft should continue to listen for commands from the helm. Control of the craft
should resume as normal once communications are reestablished (possibly by
bringing the helm into the receiving range of the craft).
State Diagrams

Craft

Water Craft Communication State Diagram

Wait for iButton
During: None
Entry: Reset Program & team indicator
Exit: None

Look for Controller
During: Listen for xBee
Entry: None
Exit: Broadcast back to helm to tell it serial numbers have been matched & turn on team indicator

Hard Reset command from Admirals

No command received from helm after 3 seconds

Turn off all actuators, keep listening

Serial numbers matched

Soft Reset from Admiral

Turn off all actuators, keep listening

Playing Game
During: If normal game play command received from helm, act accordingly.
Entry: None
Exit:

Helm acknowledges Stand Down received

Stand Down from Admiral

Turn off all actuators, tell helm stand down received

Stand Down State
During: Keep listening, keep actuators off
Entry:
Exit:
E128 Code Listing

boat.h

#ifndef BOAT
#define BOAT

//FUNCTION PROTOTYPES
// Public Function Prototypes
int RunBoatSM(int CurrentEvent);
void StartBoatSM (void);
int QueryBoatSM (void);
int CheckBoatEvents (void);

//Private function prototypes
static int During_BST_WAITING_FOR_IBUTTON(int Event);
static int During_BST_LOOKING_FOR_HELM(int Event);
static int During_BST_PLAYING_GAME(int Event);
static int During_BST_STANDING_DOWN(int Event);
static void ParseNavByte(unsigned char NAV);
static void ParseSpecialByte(unsigned char SPEC);
#endif

boat.c

(reinterpret boat.c -------------------/
//--- code courtesy of WeinerMeister---/
//----------------------------------/

//boat.c contains any code that is specific to the boat, including propeller control
#include "headers.h"
//global variables
extern unsigned char GMyTeam;

//---------------------------- Module Variables ---------------------------*/
// everybody needs a state variable, you may need others as well
static int CurrentState = 0;

//----------------------------- Boat Event Checkers ---------------------*/

//--main event checker for the boat state machine
//most events occur as a result of a new xbee communication packet, which we parse here
int CheckBoatEvents(void)
{
    int CurrentEvent = EV_NO_EVENT;
    int KeyStroke;

    //Check for events
    //These events should be arranged in order of priority, since
    //if two events are encountered at once, only process the first one so the second
    is processed the next time around
    if(CheckXbeeRX()){
        CurrentEvent = EV_NEW_XBEE;
    }
    else if(CheckSendTimer()){
        CurrentEvent = EV_TMR_SEND;
    }

    #ifdef SIMULATE_EVENTS //this allows us to simulate our state machine using
    keyboard presses
    if (kbhit() != 0){ //there was a key pressed
        KeyStroke = getchar();
        switch(toupper(KeyStroke)){
        
    }
}
case 'N' : CurrentEvent = EV_NEXT; break;

    //check for signals that we want to send an admiral command
SimulateAdmiral(KeyStroke);

#endif

return(CurrentEvent);
}

luck Runnable State Machine --------------------------*/
int RunBoatSM(int CurrentEvent)
{
    unsigned char MakeTransition = FALSE;/* are we making a state transition? */
    int NextState = CurrentState;

    //print out our current state machine status
if(CurrentEvent != EV_NO_EVENT)
    PrintState(CurrentState, CurrentEvent);

    switch (CurrentState)
    {
        case BST_WAITING_FOR_IBUTTON :
            // Execute During function for state one. EV_ENTRY & EV_EXIT are processed here
CurrentEvent = During_BST_WAITING_FOR_IBUTTON(CurrentEvent);
            //process any events
if (CurrentEvent != EV_NO_EVENT)
    {switch (CurrentEvent)
        {
            case EV_NEXT: //if a next command is pressed, skip ahead
                NextState = BST_LOOKING_FOR_HELM;
                MakeTransition = TRUE;
                break;
            case EV_IBUTTON: //If an ibutton tapped us, move on
                NextState = BST_LOOKING_FOR_HELM;
                MakeTransition = TRUE;
                break;
        }
    }
    break;

        case BST_LOOKING_FOR_HELM :
            // Execute During function for state one. EV_ENTRY & EV_EXIT are processed here
CurrentEvent = During_BST_LOOKING_FOR_HELM(CurrentEvent);
            //process any events
if (CurrentEvent != EV_NO_EVENT)
    {switch (CurrentEvent)
        {
            case EV_NEXT: //if a next command is pressed, skip ahead
                NextState = BST_PLAYING_GAME;
                MakeTransition = TRUE;
                break;
            case EV_MATCHED: //We are matched with the helm, so move on
                NextState = BST_PLAYING_GAME;
                MakeTransition = TRUE;
                break;
            case EV_HARD_RESET: //We are being reset to read another ibutton
                NextState = BST_WAITING_FOR_IBUTTON;
                MakeTransition = TRUE;
                break;
        }
    }
    break;

        default:
            break;
    }
case BST_PLAYING_GAME :
  // Execute During function for state one. EV_ENTRY & EV_EXIT are processed here
  CurrentEvent = During_BST_PLAYING_GAME(CurrentEvent);
  // process any events
  if ( CurrentEvent != EV_NO.Event )
  {
    switch (CurrentEvent)
    {
      case EV_STAND_DOWN: // We have been asked to stand down, so stop all
        actuation
          NextState = BST_STANDING_DOWN;
          MakeTransition = TRUE;
          break;
      case EV_NEXT: // If a next command is pressed, skip ahead
        NextState = BST_WAITING_FOR_IBUTTON;
        MakeTransition = TRUE;
        break;
      case EV_HARD_RESET: // We are being broken up from our helm
        NextState = BST_WAITING_FOR_IBUTTON;
        MakeTransition = TRUE;
        break;
    }
  }
  break;

case BST_STANDING_DOWN :
  // Execute During function for state one. EV_ENTRY & EV_EXIT are processed here
  CurrentEvent = During_BST_STANDING_DOWN(CurrentEvent);
  // process any events
  if ( CurrentEvent != EV_NO_EVENT )
  {
    switch (CurrentEvent)
    {
      case EV_NEXT: // If a next command is pressed, skip ahead
        NextState = BST_PLAYING_GAME;
        MakeTransition = TRUE;
        break;
      case EV_PLAY_ON: // Continue playing the game, now that stand down is acknowledged by helm
        NextState = BST_PLAYING_GAME;
        MakeTransition = TRUE;
        break;
    }
  }
  break;
}

// Check for error events, and printout
if(CurrentEvent == EV_ERROR)
  printf("EV_ERROR FOUND!\r\n");

// If we are making a state transition
if (MakeTransition == TRUE)
{
  // Execute exit function for current state
  RunBoatSM(EV_EXIT);
  // Modify state variable
  CurrentState = NextState;
  // Execute entry function for new state
  RunBoatSM(EV_ENTRY);
}

return(CurrentEvent);
}
CurrentState = BST_WAITING_FOR_IBUTTON;
// call the entry function (if any) for the ENTRY_STATE
RunBoatSM(EV_ENTRY);
}

int QueryBoatSM ( void )
{
    return(CurrentState);
}

WalletBoatSM(int Event)
{
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
        //reset all module variables and stop all actuators
        GMyTeam = NOTEAM;
        Stop(); //stop both propellers
        EraseStoredSerial(); //erase previously stored ibutton serial
        PTT & BIT6LO; //reset team lights
        PTT & BIT7LO;
    } else if ( Event == EV_EXIT)
    {
    } else
    // do the 'during' function for this state
    {
        //check for ibutton touch and update team affiliation accordingly
        if(RequestIbutton()) //if there is an ibutton present with a valid serial number,
            move on
        return EV_IBUTTON;
    } return Event;
}

WalletBoatSM(int Event)
{
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
    } else if ( Event == EV_EXIT)
    {
    } else
    // do the 'during' function for this state
    {
        //listen for xbee communications
        if(Event == EV_NEW_XBEE)
        {
            if(GetXbeeByte0() == IBUTTON){ //is the communication telling us about an
                ibutton?
                if(CheckSerialMatch()){
                    printf("We have a matched serial number! AKA we got that bitch\r\n");
                    //we're a match, so do what we need to
                    //get to know each others' zigbee addresses
                    ImprintPartner();
                    //set our team affiliation based on ibutton serial number
                    // (moved light illumination into here)
                    SetTeam(GetStoredSerialLSB());
                    //send message to the helm telling it that we have a match
                    Send218Data(TO_PARTNER, WATERCRAFT, MATCHED_1, MATCHED_2);
                    //and move to the game
                    return EV_MATCHED;
                }
            }
        }
    } return Event;
}
static int During_BST_PLAYING_GAME(int Event){
   // process EV_ENTRY & EV_EXIT events
   if ( Event == EV_ENTRY )
   {
      //initialze and reset the timer the check for lost communication
      SetTimer(TMR_LOST_COMM, LOST_COMM_TIME);
   }
   else if ( Event == EV_EXIT){
      //turn off all actuators
      Stop(); //stop both propellers
   }
   else
   // do the 'during' function for this state
   {
      //check xbee for admiral commands and nav commands
      if(Event == EV_NEW_XBEE){
         if(GetXbeeByte0() == ADMIRAL){ //admiral commands
            switch(GetXbeeByte2()){
            case STAND_DOWN: //0x01
               Send218Data(TO_ADMIRAL, ACK, 0, STAND_DOWN); //acknowledge
               return EV_STAND_DOWN;
            break;
            case SOFT_RESET: //0x20
               Send218Data(TO_ADMIRAL, ACK, 0, SOFT_RESET); //acknowledge soft reset
               Stop(); //turn off all actuators
               ParseSpecialByte(0x00);//turn off water and all special functions
               break;
            case HARD_RESET: //0x40
               return EV_HARD_RESET;
            break;
            }
         }
      }
      else if(GetXbeeByte0() == NAVIGATION){ //nav commands
         //kick the lost com timer
         SetTimer(TMR_LOST_COMM, LOST_COMM_TIME);
         ParseNavByte(GetXbeeByte1());
         ParseSpecialByte(GetXbeeByte2());
      }
   }
   //if there is no communication for three seconds, turn off all actuators and listen
   else if(CheckTimerExpired(TMR_LOST_COMM) == TRUE) {
      printf("We lost communication with helm. Turning off all actuators. \r\n");
      Stop(); //turn off all actuators
      ParseSpecialByte(0x00);//turn off water and all special functions
   }
   return Event;
}

static int During_BST_STANDING_DOWN(int Event){
   // process EV_ENTRY & EV_EXIT events
   if ( Event == EV_ENTRY )
   {
      Stop(); //turn off all actuators
      ParseSpecialByte(0x00);//turn off water and all special functions
   }
   else if ( Event == EV_EXIT)
   {
   }
   else
   // do the 'during' function for this state
   {
      //check to see if the helm has acknowledged that we are standing down
      if(Event == EV_NEW_XBEE){
         if(GetXbeeByte0() == ACK){ //helm acknowledgement that craft is standing down
            printf("Helm acknowledged our stand down\r\n");
         }
      }
   }
}

static int During_BST_STANDING_DOWN(int Event){
   // process EV_ENTRY & EV_EXIT events
   if ( Event == EV_ENTRY )
   {
      Stop(); //turn off all actuators
      ParseSpecialByte(0x00);//turn off water and all special functions
   }
   else if ( Event == EV_EXIT)
   {
   }
   else
   // do the 'during' function for this state
   {
      //check to see if the helm has acknowledged that we are standing down
      if(Event == EV_NEW_XBEE){
         if(GetXbeeByte0() == ACK){ //helm acknowledgement that craft is standing down
            printf("Helm acknowledged our stand down\r\n");
         }
      }
   }
}
return EV_PLAY_ON;
}
else { //if we have not been acknowledged, try again
    //tell the helm that we are standing down
    Send218Data(TO_PARTNER, WATERCRAFT, STAND_DOWN_RECEIVED_1,
                STAND_DOWN_RECEIVED_2);
}
return Event;

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

*/

//Parses an 8-bit nav byte containing speed and direction information
//translates speed and direction into L and R prop power
//R = speed + direction
//L = speed - direction
static void ParseNavByte(unsigned char NAV) {
    unsigned char spdNIB; //speed (from lower nibble; 0=reverse, F=forward, 8=stopped)
    unsigned char dirNIB; //direction (from upper nibble; 0=fullL, F=fullR, 8=straight)
    char Lpower, Rpower;
    char Ldirection = FORWARD; //actual calculated L motor direction
    char Rdirection = FORWARD; //actual calculated R motor direction
    char dutyValues[8] = {0,28,40,52,64,76,88,100};

    printf("NAV BYTE = \x\r\n", NAV);
    //separate out speed and direction nibbles
    spdNIB=NAV&(0x0F); //mask out the upper nibble
    dirNIB=(NAV&(0xF0))>>4; //mask out the lower nibble, then shift data into the lower
    //nibble

    //calculate relative left and right power based on speed and direction
    //FIX THIS HACKY CONVERSION CODE
    Lpower=(spdNIB-8)-(dirNIB-8); //now centered around 0 (positive=FWD, negative=BACK)
    Rpower=(spdNIB-8)+(dirNIB-8); //now centered around 0

    //convert to duty and direction
    if(Lpower < 0){
        Ldirection = BACKWARD;
        Lpower *= -1;
    }
    if(Rpower < 0){
        Rdirection = BACKWARD;
        Rpower *= -1;
    }

    //make sure power values are in range
    if(Lpower > 7)
        Lpower = 7;
    if(Rpower > 7)
        Rpower = 7;

    //look up duty cycles in a table, and set the motors
    SetMotor(L_MOTOR, Ldirection, dutyValues[Lpower]);
    SetMotor(R_MOTOR, Rdirection, dutyValues[Rpower]);

    //print debugging functions
    printf("SpeedNIB = \%d | DirectionNIB = \%d \r\n", spdNIB, dirNIB);
    printf("Lpower = \%d | Rpower = \%d \r\n", Lpower, Rpower);
    printf("Lmotor = \%d (dir=%d) | Rmotor = \%d (dir=%d) \r\n", dutyValues[Lpower], Ldirection, dutyValues[Rpower], Rdirection);
}

//Parses an 8-bit special byte, following a navigation header
static void ParseSpecialByte(unsigned char SPEC){
    printf("SPEC BYTE = \x\r\n", SPEC);
    //special button 0 or special button 1 is active
    if((SPEC & (0x30)) != 0){
        printf("Special active. Siren on!\r\n");
        PTT |= BIT2HI; //turn on siren output
} else{
    printf("Specials off.\r\n");
    PTT &= BIT2LO; //turn off siren output
}

//parse water shooting
if((SPEC & (0x0F)) != 0) { //if water is shooting
    printf("Water shooter on!\r\n");
    PTT |= BIT3HI; //turn on water output
} else{
    printf("Water off.\r\n");
    PTT &= BIT3LO; //turn off water output
}

} 

//------------- TEST FUNCTION -----------------//

#ifdef BOAT_TEST
//send a string of commands to the boat and see how L and R motors respond

void main(void){
    InitAll();

    while(TRUE){
        PrintDecAsBin(0x8F);
        printf(" - Full straight Forward! \r\n");
        ParseNavByte(0x8F);
        Wait(1500);

        PrintDecAsBin(0x83);
        printf(" - Partial straight backward \r\n");
        ParseNavByte(0x83);
        Wait(1500);

        PrintDecAsBin(0xFF);
        printf(" - Full right forward \r\n");
        ParseNavByte(0xFF);
        Wait(1500);

        PrintDecAsBin(0x25);
        printf(" - Partial left backward \r\n");
        ParseNavByte(0x25);
        Wait(1500);

        PrintDecAsBin(0x88);
        printf(" - Stopped \r\n");
        ParseNavByte(0x88);
        Wait(1500);
    }
}
#endif

defines.h

#ifndef DEFINES
#define DEFINES
//Test defines
#define HELM_MAIN
#ifndef BOAT_MAIN
#define BOAT_MAIN
#endif
#define PWM_TEST
#ifndef BOAT_TEST
#define BOAT_TEST
#endif
#define IBUTTON_SPI_TEST
#ifndef XBEE_TEST
#define XBEE_TEST
#endif
#define SERVO_TEST
#ifndef HELM_TEST
#define HELM_TEST
#endif
#define HELM_SERVO_TEST
#define SIMULATE_EVENTS //don't really need it but for simulating events

//who am I (depends on program target)
```c
#define IAMBOAT 0
#define IAMHELM 1

// team affiliation
#define NOTEAM 0
#define RED 1
#define BLUE 2
#define BASEA 3
#define BASEB 4

// send-to definitions
#define TO_BROADCAST 0
#define TO_PARTNER 1
#define TO_ADMIRAL 2

// Convenience
#define TRUE 1
#define FALSE 0
#define SUCCESS 0
#define FAILURE 1

// assign timer numbers
#define TMR_WAIT         0
#define TMR_SEND            1 // keeps track of period between sends during game at rate of 5Hz
#define TMR_LOST_COMM       2 // if comm with partner is lost for three seconds
#define TMR_STAND_DOWN      3 // for letting us know when we can start playing again
#define TMR_MUSTARD_SHAKE   4 // send new data every 200ms (CHANGE BACK!!)
#define STAND_DOWN_TIME 10000 // stand down lasts for 10 seconds
#define LOST_COMM_TIME  3000 // how much time we will tolerate no comm from helm
#define MUSTARD_SHAKE_TIME 1000 // number of milliseconds that the mustard should come out after you shake

// STATES
// boat state machine
#define BST_WAITING_FOR_IBUTTON       1
#define BST_LOOKING_FOR_HELM          2
#define BST_PLAYING_GAME              3
#define BST_STANDING_DOWN             4
// helm state machine
#define HST_WAITING_FOR_IBUTTON       1
#define HST_LOOKING_FOR_BOAT          2
#define HST_WAITING_FOR_GAME_START    3
#define HST_PLAYING_GAME              4
#define HST_CRUISING_POST_GAME        5
#define HST_STANDING_DOWN             6

// EVENTS
// general
#define EV_NO_EVENT        1
#define EV_ENTRY        2
#define EV_EXIT         3
#define EV_ERROR        4
// helm commands to craft
#define EV_NO_ACTION    5 // signal
#define EV_IBUTTON     6 // event if a valid ibutton is received
// admiral commands to craft
#define EV_STAND_DOWN    7
#define EV_GAME_START    8
#define EV_GAME_STOP     9
#define EV_HARD_RESET   10
// timer events
#define EV_TMR_SEND     11
#define EV_TMR_LOST_COMM 12 // if no communication for 3 seconds
// other
#define EV_MATCHED     13
#define EV_PLAY_ON      14
#define EV_NEXT        15
#define EV_NEW_XBEE     16
```
//boating
#define R_MOTOR  1 //use to ID the right motor
#define L_MOTOR  0 //use to ID the left motor
#define BOTH_MOTORS 2 //makes both motors do their thing
#define FORWARD  1 //motor pushes the robot forward
#define BACKWARD  0 //motor pushes the robot backward
#define RIGHT   1
#define LEFT  0

//propellor motor PWM
#define PRESCALER 2   //24Mhz clock / 2 = 12 MHz
#define POSTSCALER 3  //12 MHz / (3*2) = 2000 kHz
#define MS (24000/(PRESCALER*POSTSCALER*2)) // =1000 defines the number of ticks in a microsecond
#define MOTOR_PWM_PERIOD 100 //(MS/10) //MS/10 = 20kHz
#define DEFAULT_MOTOR_DUTY (MOTOR_PWM_PERIOD) //default duty cycle = 100%

//Ibutton
#define IBUTTON_RESET_BYTE 0xFF //arbitrary reset pattern

// SCI
#define BAUD_BITS 156  // (24000000/(16*156) = 9615 Baud) (very wrong?)
#define XBEE_MESSAGE_SIZE 12
#define SET_TO_MASTER 1
#define SET_TO_SLAVE 0

//SERVO PWM HELPERS
#define PRESCALER_A 16  //24Mhz clock / 16 = 1500 Khz
#define POSTSCALER_A 36  //24Mhz / 16 / (2*36) = 20.83 kilohertz
#define MS_B (24000/(PRESCALER_A*POSTSCALER_A*2)) // = 20.83 defines the number of ticks in a ms
#define SERVO_PWM_PERIOD 209 //(MS_B/200) = 50ish Hz
#define SERVO_MAX_DUTY 26
#define SERVO_MIN_DUTY 6 //used to be 2
#define SERVO_INIT_DUTY 6
#define ACTIVE_BASE_SERVO 4
#define RED_BOAT_NUM_SERVO 1
#define BLUE_BOAT_NUM_SERVO 0

//BOAT HELPERS
#define BOAT_PTT_INIT (BIT0LO) //0 is an input and the rest are outputs
#define BOAT_PTU_INIT (BIT0HI | BIT1HI); //Port U pins 0 and 1 are outputs
#define BOAT_PTAD_INIT ("AAAAAAAA") //0,1 are analog inputs the rest are inputs (but not currently used)

//HELM HELPERS
#define HELM_PTT_INIT ((BIT0LO & BIT1LO & BIT2LO & BIT3LO)|(BIT4HI)) //0,1,2,3 are inputs, 4 is an output
#define HELM_PTU_INIT (BIT0HI | BIT1HI | BIT4HI) //Port U pins 0, 1, and 4 are outputs
#define HELM_PTAD_INIT ("AAAAAAAA") //0,1 are analog inputs the rest are inputs (but not currently used)

#define SPEED_PIN 1
#define DIRECTION_PIN 0
#define SPEED_CONVERSION 64
#define DIRECTION_CONVERSION 64

//------------- 218C Comm Definitions ---------------------/
//Framing
#define START_BYTE 0x7E
#define LENGTH_MSB 0x00
#define LENGTH_LSB 0x08

//API identifier
#define API_RX 0x81
#define API_TX 0x01

//Frame ID: change this to non-zero if you wish
//your xBee to respond with a Tx Status message
#define FRAME_ID 0x00

// Addresses
#define ADMIRAL_ADDRESS_MSB 0xBC    // This was AF before, but the
#define ADMIRAL_ADDRESS_LSB 0xFF    // comm spec had a typo in it
#define HELM_MSB 0xBC
#define CRAFT_MSB 0xAF

// for ME218C Data Byte 0 Header
#define IBUTTON 0x01
#define NAVIGATION 0x02
#define ADMIRAL 0x04
#define WATERCRAFT 0x08
#define PING_RESPONSE 0x10
#define ACK 0x80

// Admiral Messages
#define STAND_DOWN 0x01
#define START_GAME 0x02
#define END_GAME 0x04
#define BLUE_GOAL 0x08
#define RED_GOAL 0x10
#define SOFT_RESET 0x20
#define HARD_RESET 0x40
#define ADMIRAL_PING 0x80

// Commands from Helm to Watercraft
#define NO_ACTION_1 0x88
#define NO_ACTION_2 0x00

// Commands from Watercraft to Helm
#define STAND_DOWN_RECEIVED_1 0x00
#define STAND_DOWN_RECEIVED_2 0x02
#define MATCHED_1 0x00
#define MATCHED_2 0x01

// Ping Responses
#define WAITING_IBUTTON 0x01
#define WAITING_PAIR 0x02
#define PAIRED 0x04
#define FAIRED 0x04

#endif

headers.h

#ifndef HEADERS
#define HEADERS

// Standard Libraries
#include "ME218_E128.h"
#include <hidef.h>
#include <mc9s12e128.h>
#include <bitdefs.h>
#include "S12eVec.h"  /* vector addresses for interrupts */
#include <S12e128bits.h>    /* bit definitions */
#include <timerS12.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "ADS12e.h"

// Our libraries
#include "boat.h"
#include "helm.h"
#include "defines.h"
#include "helpers.h"
#include "ibutton.h"
#include "main.h"
#include "motor.h"
#include "servo.h"
#include "xbee.h"
#endif

helm.h

#ifndef HELM
#define HELM

// FUNCTION PROTOTYPES
// Public Function Prototypes
int RunHelmSM(int CurrentEvent);
void StartHelmSM(void);
int QueryHelmSM(void);
int CheckHelmEvents(void);

// Private function prototypes
static int During_HST_CRUISING_POST_GAME(int Event);
static int During_HST_STANDING_DOWN(int Event);
static int During_HST_PLAYING_GAME(int Event);
static int During_HST_WAITING_FOR_GAME_START(int Event);
static int During_HST_LOOKING_FOR_BOAT(int Event);
static int During_HST_WAITING_FOR_IBUTTON(int Event);
static unsigned char GetSpeedLevel(void);
static unsigned char GetDirectionLevel(void);
static unsigned char CreateNavByte(void);
static unsigned char CreateSpecialByte(void);
static void SetBaseIndicator(unsigned char goal);
static void SetTeamIndicator(unsigned char team);

// switch checker functions
static unsigned char CheckResetState(void);
static unsigned char CheckSpec0State(void);
static unsigned char CheckSpec1State(void);
static unsigned char CheckWaterState(void);

#endif

helm.c

//-- code courtesy of WeinerMeister--/

helm.c contains any code that is specific to the helm, including reading all of the inputs
and sending zigbee packets to the boat

#include "headers.h"

extern unsigned char GMyTeam;

static unsigned char resetState, waterState;

int CurrentState = 0;

static void SetBaseIndicator(unsigned char goal);
static void SetTeamIndicator(unsigned char team);

// switch checker functions
static unsigned char CheckResetState(void);
static unsigned char CheckSpec0State(void);
static unsigned char CheckSpec1State(void);
static unsigned char CheckWaterState(void);

int RunHelmSM(int CurrentEvent);
void StartHelmSM(void);
int QueryHelmSM(void);
int CheckHelmEvents(void);
//most events occur as a result of a new xbee communication packet, which we parse here
int CheckHelmEvents(void)
{
    int CurrentEvent = EV_NO_EVENT;
    int KeyStroke;

    //Check for events
    //These events should be arranged in order of priority, since
    //if two events are encountered at once, only process the first one so the second
    //is processed the next time around
    if(resetState != CheckResetState()){
        if(resetState == 0){
            resetState = 1; //toggle the state variable
            CurrentEvent = EV_HARD_RESET;
        } else
            resetState = 0;
    }

    if(CheckXbeeRX()){
        CurrentEvent = EV_NEW_XBEE;
    } else if(CheckSendTimer()){
        CurrentEvent = EV_TMR_SEND;
    }

    #ifdef SIMULATE_EVENTS //this allows us to simulate our state machine using
    keyboard presses:
    else if (kbhit() != 0){ //there was a key pressed
        KeyStroke = getchar();
        switch(toupper(KeyStroke)){
            case 'N' : CurrentEvent = EV_NEXT; break;
        }
    } //check for signals that we want to send an admiral command
    SimulateAdmiral(KeyStroke);
    #endif

    return(CurrentEvent);
}

/****************************************************************************
Helm State Machine -----------------------------------------------*/
int RunHelmSM( int CurrentEvent )
{
    unsigned char MakeTransition = FALSE;/* are we making a state transition? */
    int NextState = CurrentState;

    //print out our current state machine status
    if((CurrentEvent != EV_NO_EVENT) && (CurrentEvent != EV_NEW_XBEE))
        PrintState(CurrentState, CurrentEvent);

    switch ( CurrentState )
    {
    case HST_WAITING_FOR_IBUTTON :
        // Execute During function for state one. EV_ENTRY & EV_EXIT are processed
        // here
        CurrentEvent = During_HST_WAITING_FOR_IBUTTON(CurrentEvent);
        //process any events
        if ( CurrentEvent != EV_NO_EVENT )
            switch (CurrentEvent)
            {
            case EV_NEXT: //if a next command is pressed, skip ahead
                NextState = HST_LOOKING_FOR_BOAT;
                MakeTransition = TRUE;
                break;
            case EV_IBUTTON: //If an ibutton tapped us, move on
                NextState = HST_LOOKING_FOR_BOAT;
                MakeTransition = TRUE;
                break;
            }
case HST_LOOKING_FOR_BOAT :
  // Execute During function for state one. EV_ENTRY & EV_EXIT are processed here
  CurrentEvent = During_HST_LOOKING_FOR_BOAT(CurrentEvent);
  //process any events
  if ( CurrentEvent != EV_NO_EVENT )
  {
    switch (CurrentEvent)
    {
      case EV_NEXT: //if a next command is pressed, skip ahead
        SimulateIbutton(IAMHELM); //hard code the other zigbee address into our communications
        NextState = BST_PLAYING_GAME;
        MakeTransition = TRUE;
        break;
      case EV_MATCHED: //We are matched with the boat, so move on
        NextState = BST_PLAYING_GAME;
        MakeTransition = TRUE;
        break;
      case EV_HARD_RESET: //Go to initial state because we have been reset
        NextState = HST_WAITING_FOR_IBUTTON;
        MakeTransition = TRUE;
        break;
    }
  }
  break;

case HST_WAITING_FOR_GAME_START :
  // Execute During function for state one. EV_ENTRY & EV_EXIT are processed here
  CurrentEvent = During_HST_WAITING_FOR_GAME_START(CurrentEvent);
  // HARD CODED TO START GAME RIGHT AWAY!!
  // REMOVE BEFORE FINAL CHECKOFF!
  //NextState = HST_PLAYING_GAME;
  //MakeTransition = TRUE;
  //process any events
  if ( CurrentEvent != EV_NO_EVENT )
  {
    switch (CurrentEvent)
    {
      case EV_NEXT: //if a next command is pressed, skip ahead
        NextState = HST_PLAYING_GAME;
        MakeTransition = TRUE;
        break;
      case EV_GAME_START: //we got an admiral command saying to start to the game
        NextState = HST_PLAYING_GAME;
        MakeTransition = TRUE;
        break;
      case EV_HARD_RESET: //Go to initial state because we have been reset
        NextState = HST_WAITING_FOR_IBUTTON;
        MakeTransition = TRUE;
        break;
    }
  }
  break;

case HST_PLAYING_GAME :
  // Execute During function for state one. EV_ENTRY & EV_EXIT are processed here
  CurrentEvent = During_HST_PLAYING_GAME(CurrentEvent);
  //process any events
  if ( CurrentEvent != EV_NO_EVENT )
  {
    switch (CurrentEvent)
    {
      case EV_STAND_DOWN: //if we get a stand down command, go into the stand down state
        break;
    }
  }
  break;
NextState = HST_STANDING_DOWN;
MakeTransition = TRUE;
break;
case EV_NEXT: //if a next command is pressed, skip ahead
    NextState = HST_CRUISING_POST_GAME;
    MakeTransition = TRUE;
break;
case EV_GAME_STOP: //If we get a game over command from the admiral then
go to game over state
    NextState = HST_CRUISING_POST_GAME;
    MakeTransition = TRUE;
break;
case EV_HARD_RESET: //Go to initial state because we have been reset
    NextState = HST_WAITING_FOR_IBUTTON;
    MakeTransition = TRUE;
break;
}

NextState = HST_STANDING_DOWN;
MakeTransition = TRUE;
break;
case HST_STANDING_DOWN :
    // Execute During function for state one. EV_ENTRY & EV_EXIT are processed here
    CurrentEvent = During_HST_STANDING_DOWN(CurrentEvent);
    //process any events
    if ( CurrentEvent != EV_NO_EVENT )
    {
        switch (CurrentEvent)
        {
            case EV_NEXT: //if a next command is pressed, skip ahead
                NextState = HST_PLAYING_GAME;
                MakeTransition = TRUE;
                break;
            case EV_PLAY_ON: //Continue playing the game, now that stand down is
                acknowledged by helm
                NextState = HST_PLAYING_GAME;
                MakeTransition = TRUE;
                break;
            case EV_HARD_RESET: //Go to initial state because we have been reset
                NextState = HST_WAITING_FOR_IBUTTON;
                MakeTransition = TRUE;
                break;
        }
    }
    break;

NextState = HST_CRUISING_POST_GAME;
MakeTransition = TRUE;
break;
case HST_CRUISING_POST_GAME :
    // Execute During function for state one. EV_ENTRY & EV_EXIT are processed here
    CurrentEvent = During_HST_CRUISING_POST_GAME(CurrentEvent);
    //process any events
    if ( CurrentEvent != EV_NO_EVENT )
    {
        switch (CurrentEvent)
        {
            case EV_NEXT: //if a next command is pressed, skip ahead
                NextState = HST_WAITING_FOR_IBUTTON;
                MakeTransition = TRUE;
                break;
            case EV_HARD_RESET: //Go to initial state because we have been reset
                NextState = HST_WAITING_FOR_IBUTTON;
                MakeTransition = TRUE;
                break;
        }
    }
    break;
}

// Check for error events, and printout
if (CurrentEvent == EV_ERROR)
    printf("EV_ERROR FOUND!\r\n");
// If we are making a state transition
if (MakeTransition == TRUE)
{
    // Execute exit function for current state
    RunHelmSM(EV_EXIT);
    CurrentState = NextState; //Modify state variable
    // Execute entry function for new state
    RunHelmSM(EV_ENTRY);
}

   return(CurrentEvent);
}

// Function StartGameSM
****************************************************************************
void StartHelmSM ( void )
{
    //do initialization of helm module variables here
    //Initialize initial state of actuators
    resetState = CheckResetState();
    waterState = CheckWaterState();
    CurrentState = HST_WAITING_FOR_IBUTTON;
    // call the entry function (if any) for the ENTRY_STATE
    RunHelmSM(EV_ENTRY);
}

int QueryHelmSM ( void )
{
   return(CurrentState);
}

private functions
****************************************************************************

static int During_HST_WAITING_FOR_IBUTTON(int Event)
{
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
        //reset all module variables and stop all actuators
        GMyTeam = NOTEAM;
        EraseStoredSerial(); //erase previously stored ibutton serial

        //reset all servos to initial position
        printf("Initializing servos to home position\r\n\n");
        SetBaseIndicator(NOTEAM);
        SetTeamIndicator(NOTEAM);

        //turn off siren to indicate we are no longer being stood down
        printf("Initializing siren to OFF\r\n\n");
        PTT &= BIT4LO;
    }else if ( Event == EV_EXIT)
    {
    }
    else
        // do the 'during' function for this state
        {  //check for ibutton touch and update team affiliation accordingly
            if(RequestIbutton()) //if there is an ibutton present with a valid serial number,
            move on
            return EV_IBUTTON;
        }

    return Event;
}

static int During_HST_LOOKING_FOR_BOAT(int Event)
{
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
    }
    else
        // do the 'during' function for this state
        {
            printf("Initializing servos to home position\r\n\n");
            SetBaseIndicator(NOTEAM);
            SetTeamIndicator(NOTEAM);

            //turn off siren to indicate we are no longer being stood down
            printf("Initializing siren to OFF\r\n\n");
            PTT &= BIT4LO;
        }

    return Event;
}
{  
  else if ( Event == EV_EXIT) 
  {
  
  }else 
  // do the 'during' function for this state 
  {
      //do we need to send zigbee communications 
      if(Event == EV_TMR_SEND){
          //send serial number 
          Send218Data(TO_BROADCAST, IBUTTON, GetStoredSerialMSB(), GetStoredSerialLSB());
      }

      //listen for xbee communications 
      if(Event == EV_NEW_XBEE){
          if(GetXbeeByte0() == WATERCRAFT){ //is the communication telling us about an 
              ibutton?
                  if((GetXbeeByte1() == MATCHED_1) && (GetXbeeByte2() == MATCHED_2)){
                      //we're a match, so do what we need to 
                      //get to know each others' zigbee addresses 
                      ImprintPartner();

                      //set our team affiliation (this feels weird for Mr. Helm) 
                      SetTeam(GetStoredSerialLSB());
                      //turn on our team servo 
                      SetTeamIndicator(GetTeamNumber()); //use the team # to set our team 
                      affiliation 
                      //and move to the waiting for game start stage 
                      return EV_MATCHED;
                  }
          }
      }

      return Event;
  }

static int During_HST_WAITING_FOR_GAME_START(int Event){

  // process EV_ENTRY & EV_EXIT events
  if ( Event == EV_ENTRY)
  {
      
  }else if ( Event == EV_EXIT)
  {
      //do nothing
  }

  else 
  // do the 'during' function for this state 
  {

      //do we need to send zigbee communications 
      if(Event == EV_TMR_SEND){
          //send no action 
          Send218Data(TO_PARTNER, NAVIGATION, NO_ACTION_1, NO_ACTION_2);
      }

      //check xbee for admiral command to start game 
      if(Event == EV_NEW_XBEE){
          if(GetXbeeByte0() == ADMIRAL){ //admiral commands 
              if(GetXbeeByte2() == START_GAME){
                  Send218Data(TO_ADMIRAL, ACK, 0, START_GAME); //acknowledge game 
                  has started 
                  return EV_GAME_START;
              }
          }
      }

      return Event;
  }

static int During_HST_PLAYING_GAME(int Event){

  // process EV_ENTRY & EV_EXIT events 
  if ( Event == EV_ENTRY) 
  {
  
  }else if ( Event == EV_EXIT) 
  {

}
}  
else  
  // do the 'during' function for this state  
  
  //check xbee for admiral commands and nav commands  
  if(Event == EV_NEW_XBEE){  
    if(GetXbeeByte0() == WATERCRAFT){  //watercraft commands  
      if(GetXbeeByte2() == STAND_DOWN_RECEIVED_2)  
        //Acknowledge the boat is standing down  
        Send218Data(TO_PARTNER, ACK, STAND_DOWN_RECEIVED_1,  
        STAND_DOWN_RECEIVED_2);  
        return EV_STAND_DOWN;  
    }  
    // Check for admiral commands  
    else if(GetXbeeByte0() == ADMIRAL){  //admiral commands  
      switch (GetXbeeByte2())  
      {  
        case END_GAME :  
          Send218Data(TO_ADMIRAL, ACK, 0, END_GAME);  
          //acknowledge game has ended  
          return EV_GAME_STOP;  
        case HARD_RESET :  
          return EV_HARD_RESET;  
        break;  
        case BLUE_GOAL :  
          SetBaseIndicator(BASEA); //Turn on blue goal servo  
          break;  
        case RED_GOAL :  
          SetBaseIndicator(BASEB); //Turn on blue goal servo  
          break;  
      }  
      }  
    }  
    // if it is time to send, then we send nav and special data to our boat  
    if(Event == EV_TMR_SEND)  
    {  
      Send218Data(TO_PARTNER, NAVIGATION, CreateNavByte(), CreateSpecialByte());  
    }  
    return Event;  
  }  
}  

static int During_HST_STANDING_DOWN(int Event){  
  // process EV_ENTRY & EV_EXIT events  
  if (Event == EV_ENTRY)  
  {  
    SetTimer(TMR_STAND_DOWN, STAND_DOWN_TIME);  
    //turn on siren to indicate we are being stood down  
    printf("Turning on siren as we stand down\r\n");  
    PTT |= BIT4HI;  
  }  
  else if (Event == EV_EXIT)  
  {  
    //turn off siren to indicate we are no longer being stood down  
    printf("Turning off siren as we exit stand down\r\n");  
    PTT &= BIT4LO;  
  }  
  else  
  // do the 'during' function for this state  
  {  
    //check xbee for admiral commands and nav commands  
    if(Event == EV_NEW_XBEE){  
      // Check for admiral commands  
      if(GetXbeeByte0() == ADMIRAL){  //admiral commands  
        switch (GetXbeeByte2())  
        {  
          case END_GAME :  
            return EV_GAME_STOP;  
        }  
      }  
    }  
}
case HARD_RESET:
    return EV_HARD_RESET;
    break;

case BLUE_GOAL:
    SetBaseIndicator(BLUE); //Turn on blue goal servo
    break;

case RED_GOAL:
    SetBaseIndicator(RED); //Turn on blue goal servo
    break;

//Check if stand down timer is expired
if(CheckTimerExpired(TMR_STAND_DOWN))
    return EV_PLAY_ON;

//do we need to send zigbee communications
if(Event == EV_TMR_SEND)
    //Send218Data(TO_PARTNER, NAVIGATION, NO_ACTION_1,NO_ACTION_2);
    Send218Data(TO_PARTNER, ACK, STAND_DOWN_RECEIVED_1, STAND_DOWN_RECEIVED_2); //keep acknowledging to make sure our boat knows that we know that it's standing down
    return Event;
}

static int During_HST_CRUISING_POST_GAME(int Event){
    // process EV_ENTRY & EV_EXIT events
    if ( Event == EV_ENTRY)
    {
        SetBaseIndicator(NOTEAM);
        SetTeamIndicator(NOTEAM);
    }
    else if ( Event == EV_EXIT)
    {
    }
    else
    // do the 'during' function for this state
    {
        //check xbee for admiral commands and nav commands
        if(Event == EV_NEW_XBEE){
            // Check for admiral commands
            if(GetXbeeByte0() == ADMIRAL){ //admiral commands
                if (GetXbeeByte2() == HARD_RESET)
                    return EV_HARD_RESET;
            }
        }
        // if it is time to send, then we send nav and special data to our boat
        if(Event == EV_TMR_SEND)
        {
            Send218Data(TO_PARTNER, NAVIGATION, CreateNavByte(), 0x00);
        }
    }
    return Event;
}

// Creates the special byte, which will be sent to the boat
static unsigned char CreateSpecialByte (void)
{
    unsigned char special_byte = 0;
    //special button 0
    if(CheckSpec0State()) special_byte |= BIT5HI;
    //special button 1
    if(CheckSpec1State()) special_byte |= BIT4HI;
    //water dispensor
    if(CheckWaterState()) special_byte |= 0x0F;

    return special_byte;
}
// Gets the readings from the speed and direction inputs and assembles
// byte1 to send the appropriate data to the boat
static unsigned char CreateNavByte(void)
{
    unsigned char speed_level;
    unsigned char direction_level;
    unsigned char byte1;

    // Read the sensors and store the values
    speed_level = GetSpeedLevel();
    direction_level = GetDirectionLevel();

    // Shift a nibble and put it in the byte
    byte1 = (direction_level << 4);
    // Add the lower nibble
    byte1 += speed_level;

    printf("    byte1: %h \r\n", byte1);
    return byte1;
}

// Returns a number between 0 and 15 (one nibble)
static unsigned char GetSpeedLevel(void)
{
    int temp;
    unsigned char level;

    temp = ADS12_ReadADPin(SPEED_PIN);
    // This should get us a number between 0 and 15
    level = temp/SPEED_CONVERSION + 1; // add one to correct for voltage levels
    // Some test to make sure we are in the range
    if(level > 15)
        level = 15;
    if(level < 0)
        level = 0;

    printf("    speed level: %d \r\n", level);
    return level;
}

// Returns a number between 0 and 15 (one nibble)
static unsigned char GetDirectionLevel(void)
{
    int temp;
    unsigned char level;

    temp = ADS12_ReadADPin(DIRECTION_PIN);
    // This should get us a number between 0 and 15
    level = temp/DIRECTION_CONVERSION + 1; // add one to correct for voltage levels
    // Some test to make sure we are in the range
    if(level > 15)
        level = 15;
    if(level < 0)
        level = 0;

    printf("    direction level: %d \r\n", level);
    return level;
}

// Sets the goal servo to red, blue, or no team
static void SetBaseIndicator(unsigned char goal){
    if(goal == BASEA){
        SetServoPosition(8, ACTIVE_BASE_SERVO);
        printf("Setting base indicator servo to BASEA\r\n");
    }
    else if (goal == BASEB) {
        SetServoPosition(26, ACTIVE_BASE_SERVO);


```c
    printf("Setting base indicator servo to BASEB\n");
    }
else{ //no team
    SetServoPosition(17, ACTIVE_BASE_SERVO);
    printf("Setting base indicator servo to NONE\n");
    }
}

//sets the team indicator servo to the correct position, and zeros the other team's servo
static void SetTeamIndicator(unsigned char team){
    printf("Setting team indicator servo to team number: %d\n", team);
    if((team % 2) == 0){ //we are on the red team
        SetServoPosition(team, RED_BOAT_NUM_SERVO);
        SetServoPosition(0, BLUE_BOAT_NUM_SERVO);
    }
    else { //we are on the blue team
        SetServoPosition(team, BLUE_BOAT_NUM_SERVO);
        SetServoPosition(0, RED_BOAT_NUM_SERVO);
    }
}

static unsigned char CheckResetState(void){
    if(PTT & BIT3HI) return TRUE;
    else return FALSE;
}

static unsigned char CheckSpec0State(void){
    if(PTT & BIT0HI) return TRUE;
    else return FALSE;
}

static unsigned char CheckSpec1State(void){
    if(PTT & BIT1HI) return TRUE;
    else return FALSE;
}

static unsigned char CheckWaterState(void){
    //check for a change in tilt switch position from disengaged to engaged
    unsigned char switchState;
    if((PTT & BIT2HI) == 0)  //the current state of the mustard switch
        switchState = 0;
    else
        switchState = 1;
    if(waterState != switchState){
        //reset the timer for continuing spray
        TMRS12_InitTimer(TMR_MUSTARD_SHAKE, MUSTARD_SHAKE_TIME);
        waterState = switchState;
    }
    if(TMRS12_IsTimerExpired(TMR_MUSTARD_SHAKE) == TMRS12_EXPIRED)
        return FALSE;returnValue //do not spray
    return TRUE; //spray away
}

//------------TEST FUNCTIONS------------------/

//Tests the basic helm xbee transmitting functionality
#ifdef HELM_TEST
void main(void)
{
    unsigned char byte0;
    unsigned char byte1;
    unsigned char byte2;

    printf("Being Helm Test! \r\n");
    while(TRUE)
    {
        //This is blocking code so we only transmit at 5Hz
        Wait(SEND_RATE);
```
printf("Time to transmit! \r\n");
//Get all the data we need to send a packet
byte0 = NAVIGATION; //for testing we are sending navigation commands
byte1 = CreateNavByte();
byte2 = CreateSpecialByte(); //for testing we don't need the special
actions byte

//Send218Data(TO_PARTNER, byte0, byte1, byte2);
Send218Data(TO_BROADCAST, byte0, byte1, byte2);
printf("Transmission complete! \r\n");
}
}
#endif

//tests to make sure the servos on the helm are pointing to the correct places
#ifdef HELM_SERVO_TEST
void main(void)
{
    char i;
    InitAll();
    //Cycle through variable pulse lengths
    while(TRUE)
    {
        printf("Testing base indicator servo\r\n");
        SetBaseIndicator(NOTEAM);
        Wait(1500);
        SetBaseIndicator(BASEA);
        Wait(1500);
        SetBaseIndicator(BASEB);
        Wait(1500);

        for(i=0; i<=12; i++)
        {
            SetTeamIndicator(i);
            Wait(2000);
        }
    }
}
#endif

helpers.h

#ifndef HELPERS
#define HELPERS

//Function Prototypes
//timer functions
void Wait(int ticks);
void SetTimer(unsigned char timer, int ticks);
unsigned char CheckTimerExpired(unsigned char timer);
unsigned char CheckSendTimer(void);

//other helper functions
void PrintDecAsBin(unsigned char decimal);
void TestDecToBin(void);
void dec2bin(unsigned char decimal, unsigned char *binary);
#endif

helpers.c

//------------ helpers.c -----------//
//-- code courtesey of BurgerStache --//
//----------------------------------//
#include "headers.h"

// Waits for a number of milliseconds given by ticks (blocking)
void Wait(int ticks) {
    // uses timer 0 for blocking WAIT, which is one of 8 possible timers
    TMRS12_InitTimer(TMR_WAIT, ticks);
    while(TMRS12_IsTimerExpired(0) != TMRS12_EXPIRED);
}

// sets a timer to count down
// input the length of the timer in MS and the ID of the timer
void SetTimer(unsigned char timer, int ticks) {
    printf(" Timer %d set with ticks = %d\n", timer, ticks);
    TMRS12_InitTimer(timer, ticks);
}

// returns true if 200ms have passed, so it is time to send
unsigned char CheckSendTimer(void) {
    // Initialize if this is the first time calling this function
    if((TMRS12_IsTimerActive(TMR_SEND) == FALSE) || (TMRS12_IsTimerExpired(TMR_SEND) == TMRS12_EXPIRED)) {
        TMRS12_InitTimer(TMR_SEND, SEND_RATE);  // reset the timer if we are returning true
        return TRUE;  // return true if 200ms has elapsed since last call
    }
    return FALSE;
}

// returns true if the given timer is expired
unsigned char CheckTimerExpired(unsigned char timer) {
    unsigned char timex = (TMRS12_IsTimerExpired(timer) == TMRS12_EXPIRED);
    if(timex == TRUE) {
        printf("  Timer %d expired\n", timer);
        TMRS12_ClearTimerExpired(timer);  // clear the timer so we don't keep creating events
    }
    return timex;
}

// prints a decimal number as a binary string
void PrintDecAsBin(unsigned char decimal) {
    char binary[80];
    dec2bin(decimal, binary);
    printf("%s", binary);
}

// Test function for our decimal to binary printing function
void TestDecToBin(void) {
    long decimal;
    char binary[80];
    printf(" Enter an integer value : ");
    scanf("%ld", &decimal);
    dec2bin(decimal, binary);
    printf(" The binary value of %ld is %s \n", decimal, binary);
    getchar();  // trap enter
    getchar();  // wait
}

// accepts a positive decimal integer and returns a binary coded string
void dec2bin(unsigned char decimal, char *binary) {
    int k = 0, n = 0;
    int neg_flag = 0;
    int remain;
    char temp[80];

    do // parse the number, starting with the LSB
    {
        remain = decimal % 2;
        //...
// whittle down the decimal number
decimal = decimal / 2;
// converts digit 0 or 1 to character '0' or '1'
temp[k++] = remain + '0';
} while (decimal > 0);

//fill the remaining bits with zeros
while (k < 8)
{
    temp[k++] = '0';
}

// reverse the spelling
while (k > 0)
{
    binary[n++] = temp[--k];
}
    binary[n] = 0; // end with NULL
}

ibutton.h

.isPresent ibutton.c is-------
//-- code courtesy of WeinerMeister--/
//-------------------------------/

//ibutton.c asks for byte #2 from the ibutton to get the serial number
#include "headers.h"

//Function prototypes

//public
void InitSPI(int isMaster);
unsigned char ReadIbutton(void);
unsigned char GetStoredSerialLSB(void);
unsigned char GetStoredSerialMSB(void);
void EraseStoredSerial(void);

//private
unsigned char RequestIbutton(void);
static unsigned char SPITx(unsigned char Tx);
static unsigned char SPIRx(void);
static unsigned int ReceiveIbuttonByte(void);

ibutton.c

.isPresent ibutton.c is-------
//-- code courtesy of WeinerMeister--/
//-------------------------------/

//ibutton.c is responsible for interfacing with the ibutton and managing the team
affiliation data structure
//asks for byte #2 from the ibutton to get the serial number
//looks up a table of serial numbers and corresponding team colors
#include "headers.h"

//global variables
extern unsigned char GMyTeam;

//module variables
unsigned char MySerialLSB = 0;
unsigned char MySerialMSB = 0;
unsigned char SerialDataLow = 0;
unsigned char SerialDataHigh = 0;
unsigned char NewSerialFlag = 0;

/////////////////////////////////////////////////////////// 
// SPI FUNCTIONS

void InitSPI(int isMaster)
{
    //Initialize the SPI system
    SPICR1 |= _S12_SPE; //Enable SPI
    SPICR1 |= _S12_SPIE; //Enable SPI Interrupt

    switch (isMaster)
    {
        case SET_TO_MASTER:
            SPICR1 |= _S12_MSTR; //Make master
            printf("SPI mode changed to MASTER \n"r); //Sample the second byte from the ibutton, which is the second byte
            //if the second byte does exist in our table, set the team!
            unsigned char ReadIbutton(void)
            {
                //Ignore first byte from the ibutton
                //Receive the second byte and verify that it is a proper serial number
                //return the serial number if it is valid
                //also save this serial number as a module variable
                //MySerial = serial #
                //do a table lookup and set our team number accordingly
                GMyTeam = BLUE; //ex
                //return 0 if a bad serial number or no ibutton
                return 0;
            }

        case SET_TO_SLAVE:
            SPICR1 &= ~_S12_MSTR; //Make slave
            printf("SPI mode changed to SLAVE \n"r); //Sample the second byte from the ibutton, which is the second byte
            //if the second byte does exist in our table, set the team!
            unsigned char ReadIbutton(void)
            {
                //Ignore first byte from the ibutton
                //Receive the second byte and verify that it is a proper serial number
                //return the serial number if it is valid
                //also save this serial number as a module variable
                //MySerial = serial #
                //do a table lookup and set our team number accordingly
                GMyTeam = BLUE; //ex
                //return 0 if a bad serial number or no ibutton
                return 0;
            }

        default:
            printf("Error setting SPI mode! \n"r); //Sample the second byte from the ibutton, which is the second byte
            //if the second byte does exist in our table, set the team!
            unsigned char ReadIbutton(void)
            {
                //Ignore first byte from the ibutton
                //Receive the second byte and verify that it is a proper serial number
                //return the serial number if it is valid
                //also save this serial number as a module variable
                //MySerial = serial #
                //do a table lookup and set our team number accordingly
                GMyTeam = BLUE; //ex
                //return 0 if a bad serial number or no ibutton
                return 0;
            }
    }

    //Set the baud rate to 11kHz
    /******** Baud rate = 24MHz / ((SPPR+1)*2^(SPR+1))******/
    SPIBR |= _S12_SPPR2 | _S12_SPPR1 | _S12_SPPR0; //7
    SPIBR |= _S12_SPR2 | _S12_SPR1 | _S12_SPR0; //7
    EnableInterrupts;
}

//simply returns our stored serial number, returning zero if not affiliated with an
//ibutton
unsigned char GetStoredSerialLSB(void)
{
    return MySerialLSB;
}

unsigned char GetStoredSerialMSB(void)
{
    return MySerialMSB;
}

//simply erases our stored serial number, replacing it with zero
void EraseStoredSerial(void)
{
    MySerialLSB = 0;
    MySerialMSB = 0;
}
static void ResetIbutton(void)
{
    //send character IBUTTON_RESET_BYTE to the ibutton, which signals the PIC to clear its
    //memory of which ibutton it met
    SPITx(IBUTTON_RESET_BYTE);
}

//Transmits a character to the SPI data register, then proceeds to transmit it
//automatically
//Returns SUCCESS if transmitted successfully
//Returns FAILURE if another transfer is in progress
static unsigned char SPITx(unsigned char Tx)
{
    unsigned char dummy = 0; //dummy variable for reading SPISR. Simply by reading a
    //variable in, it is cleared.

    //printf("Ready, writing data...
\r");
    //Transmit
    if((SPISR & _S12_SPTEF) == 0) //This line will fail if the slave is not
        return FAILURE;
    //clear the SPIF flag, which is the received data flag. May not be necessary
    dummy = SPISR;
    dummy = SPIDR;
    //clear the SPTEF flag and writes data to SPIDR
    dummy = SPISR;
    SPIDR = Tx;
    //printf("Done transmitting...
\r");
    return SUCCESS;
}

//Returns a character that is received from the E128 SPI data register
static unsigned char SPIRx(void)
{
    unsigned char dummy = 0;
    unsigned char Rx; //data that is to be received

    //Receive
    //printf("Want to read from slave...
\r");
    while( (SPISR & _S12_SPTEF) == 0); //BLOCKING CODE: will wait to receive data
    //while( (SPISR & _S12_SPIF) == 0); slave uses SPIF usually, master uses SPTEF
    //usually. But both are reset.

    //Clears the SPIF flag
    dummy = SPISR;
    dummy = SPIDR;

    //Clears SPTEF flag and transmits dummy data
    dummy = SPISR;
    SPIDR = 0; //transmits a zero
    while(!((SPISR & _S12_SPTEF))); //also blocking code. Necessary, but may split into
two functions for project.
    while((SPISR & _S12_SPIF) == 0); //slave uses SPIF usually, master uses SPTEF

    //Read what is received and return it in Rx
    dummy = SPISR;
    Rx = SPIDR;

    //printf("SPIRx sees: %d\n\r", Rx);
    return Rx;
}

unsigned char RequestIbutton(void)
{
    // This sets PTP2 high to signal the PIC to read an iButton.

    unsigned char dummy = 0; //dummy variable for reading SPISR. Simply by reading a
    //variable in, it is cleared.
    //printf("Ready, writing data...
\r");
    //Transmit
    if((SPISR & _S12_SPTEF) == 0) //This line will fail if the slave is not
        return FAILURE;
    //clear the SPIF flag, which is the received data flag. May not be necessary
    dummy = SPISR;
    dummy = SPIDR;
    //clear the SPTEF flag and writes data to SPIDR
    dummy = SPISR;
    SPIDR = Tx;
    //printf("Done transmitting...
\r");
    return SUCCESS;
}

//Returns SUCCESS if transmitted successfully
//Returns FAILURE if another transfer is in progress
static unsigned char SPITx(unsigned char Tx)
{
DDRP |= BIT2HI;
PTP |= BIT2HI;

if(NewSerialFlag) {
PTP &= BIT2LO;
MySerialLSB = SerialDataLow;
MySerialMSB = SerialDataHigh;
NewSerialFlag = 0;           //clear flag
printf("Ibutton requested = %x %x \\n", SerialDataHigh, SerialDataLow);
    return TRUE;
}  
return FALSE;

}  
/*// OLD, BLOCKING WAY TO receive IButton byte  
static unsigned int ReceiveIbuttonByte(void)
{  
    char dummy;
    PTP |= BIT2HI;

    while(!(SPISR & _S12_SPIF));    // BLOCKING CODE!!! ************
dummy = SPISR;
SerialDataLow = SPIDR;
while(!(SPISR & _S12_SPIF));    // BLOCKING CODE!!! ************
dummy = SPISR;
SerialDataHigh = SPIDR;
SerialDataHigh = ( SerialDataHigh << 8 ) + SerialDataLow;
PTP &= BIT2LO;
return SerialDataHigh;
}*/

void interrupt _Vec_spi ReadSPI (void)
{
    static unsigned char byte_number = 1;
    unsigned char status;
    unsigned char new_data;
    //THOU SHALT NOT USE PRINTF IN AN SPI INTERRUPT ROUTINE!!
    //printf("\n\r");  
    status = SPISR;
    new_data = SPIDR;  //Read data (this also clears the flag)

    if((byte_number == 1) && (NewSerialFlag != 1))  //
    {
        //printf("1
\r"); //"D" - we have data
        SerialDataLow = new_data;
        byte_number = 2;
    }
    else if((byte_number == 2) && (NewSerialFlag != 1))  //
    {
        //printf("2\n\r"); //"D" - we have data
        SerialDataHigh = new_data;
        byte_number = 1;
        NewSerialFlag = 1;
    }
}  // End of SPI ISR

/**************************** TESTING FUNCTIONS *****************************/
//the main function is used for testing only
#ifdef IBUTTON_SPI_TEST
void main(void) {
    unsigned int iButton = 0;
    unsigned char dummy;
    printf("Beginning SPI test for E128!\n\r");
    //Initialize various functionalities
    InitSPI(SET_TO_SLAVE);
    //Init PP2 to output (to tell pic to send the serial number)
    DDRP &= BIT2LO;
    while(TRUE) {
        if(kbhit() != 0) {
            dummy = getchar(); //this makes it not go into a weird loop
            printf("Beginning to get iButton");
            while(! RequestIbutton() );
            printf("Byte 1 is: %X \n\r", MySerialLSB);
            printf("Byte 2 is: %X \n\r", MySerialMSB);
        }
    }
    return;
} #endif

main.h
#ifndef MAIN
#define MAIN
//Function prototypes
void InitAll(void);
void InitBoat(void);
void InitHelm(void);
unsigned char CheckSerialMatch(void);
void SimulateAdmiral(unsigned char KeyStroke);
void PrintState(int state, int event);
void SetTeam(unsigned char teamNum);
#endif

main.c
#include "headers.h"
//global variables
unsigned char GMyTeam;
#ifdef HELM_MAIN
    unsigned char GWhoAmI = IAMHELM; //IAMHELM or IAMBOAT
#else
    unsigned char GWhoAmI = IAMBOAT; //IAMHELM or IAMBOAT
#endif
//module variables

#ifdef BOAT_MAIN
void main(void){
    //Initialize all variables
    InitAll();
    //start the master state machine initialization
    printf("Starting boat state machine\r\n");
    StartBoatSM();
    //check for and handle events
    while(TRUE){
        RunBoatSM(CheckBoatEvents());
    }
}
#endif

#ifdef HELM_MAIN
void main(void){
    //Initialize all variables
    InitAll();
    //start the master state machine initialization
    printf("Starting helm state machine\r\n");
    StartHelmSM();
    //check for and handle events
    while(TRUE){
        RunHelmSM(CheckHelmEvents());
    }
}
#endif

//InitAll does any initialization that is identical for the boat and the helm
//then it calls the specific boat and helm init procedures
void InitAll(void) {
    printf("\r\nWelcome to me.\r\n");
    printf("Initializing all.\r\n");
    //Initialize timer
    TMRS12_Init(TMRS12_RATE_1MS);
    //call inferior initialization functions
    InitSPI(SET_TO_SLAVE);
    InitSCI(); //for xbee
    //Init PP2 to output (to tell pic to send the serial number)
    DDRP |= BIT2HI;
    PTP &= BIT2LO;
    //team affiliation
    GMyTeam = NOTEAM;
    //Check and print battery voltages
    //CheckBattVoltages();
    //Do specialized init procedures for boat and helm
    if(GWhoAmI == IAMBOAT){ //check the boat SM if we're a boat
        printf("I am the boat.\r\n");
        InitBoat();
    } else{ //I am a helm
        printf("I am the helm.\r\n");
        InitHelm();
    }
}

//Init boat ports, etc.
void InitBoat(void){
    printf("Initializing boat.\r\n");
    InitPWM(); //Initialize PWM for boat propellors
// Set port directions
DDRT = BOAT_PTT_INIT;
DDRU = BOAT_PTU_INIT;

// Set initial pin values
PTT = 0;
PTU = 0;

// Set motors to begin at a stop
Stop();

// Init helm ports, etc.
void InitHelm(void){
  printf("Initializing helm.\r\n");
  InitServoPWM(); // for silly helm dials

  // Set port directions
  DDRT = HELM_PTT_INIT;
  DDRU = HELM_PTU_INIT;

  // Set initial pin values
  PTT = 0;
  PTU = 0;

  // AD
  ADS12_Init(HELM_PTAD_INIT);

  // Initializes AD ports
  if(ADS12_Init(BOAT_PTAD_INIT) != ADS12_OK)
    printf("ERR: AD Initialization unsuccessful\r\n");
}

// compares the serial number incoming from the xbee and the ibutton serial
// if they are the same (and non-zero), then return TRUE
// call this only when you know that an ibutton has been read with RequestIbutton
 unsigned char CheckSerialMatch(void){
  // return true if both bytes are matched and non-zero
  return ((GetXbeeByte1() == GetStoredSerialMSB())   // MSB
    && (GetXbeeByte2() == GetStoredSerialLSB())     // LSB
    && ((GetXbeeByte1() != 0) || (GetXbeeByte2() != 0)));// at least one byte is non-zero
}

// sets our team affiliation and appropriate lights
void SetTeam(unsigned char teamNum){
  printf("Team number = %d\r\n",teamNum);
  if((teamNum %2) == 0){// even teams are BLUE
    printf("We're on the BLUE team\r\n");
    PTT |= BIT7HI;  // blue team on and red team off
    PTT &= BIT6LO;
    GMyTeam = BLUE;
  } else // odd teams are RED
  {
    printf("We're on the RED team\r\n");
    PTT |= BIT6HI;  // red team on and blue team off
    PTT &= BIT7LO;
    GMyTeam = RED;
  }
}

// takes a keystroke (numbers 1 through 8) and sends the corresponding admiral command to our partner
void SimulateAdmiral(unsigned char KeyStroke){

unsigned char sendByte = 0;
switch(toupper(KeyStroke)){
    case '1':
        sendByte = STAND_DOWN;
        printf("Admiral says to STAND_DOWN\r\n");
        break;
    case '2':
        sendByte = START_GAME;
        printf("Admiral says to START_GAME\r\n");
        break;
    case '3':
        sendByte = END_GAME;
        printf("Admiral says to END_GAME\r\n");
        break;
    case '4':
        sendByte = BLUE_GOAL;
        printf("Admiral says to BLUE_GOAL\r\n");
        break;
    case '5':
        sendByte = RED_GOAL;
        printf("Admiral says to RED_GOAL\r\n");
        break;
    case '6':
        sendByte = SOFT_RESET;
        printf("Admiral says to SOFT_RESET\r\n");
        break;
    case '7':
        sendByte = HARD_RESET;
        printf("Admiral says to HARD_RESET\r\n");
        break;
    case '8':
        sendByte = ADMIRAL_PING;
        printf("Admiral says to ADMIRAL_PING\r\n");
        break;
}
    // send an admiral command to our partner
    if(sendByte != 0)
    Send218Data(TO_PARTNER, ADMIRAL, 0x00, sendByte);
}

// ------------ DEBUGGING FUNCTIONS -------------------

void PrintState(int state, int event){
    printf("\r\n-----State Machine-----\r\n");
    printf("CurrentState = ");
    if(GWhoAmI == IAMBOAT){
    // boat
        switch(state) {
            case BST_WAITING_FOR_IBUTTON : printf("BST_WAITING_FOR_IBUTTON"); break;
            case BST_LOOKING_FOR_HELM   : printf("BST_LOOKING_FOR_HELM"); break;
            case BST_PLAYING_GAME       : printf("BST_PLAYING_GAME"); break;
            case BST_STANDING_DOWN      : printf("BST_STANDING_DOWN"); break;
        }
    } else{
    // helm
        switch(state) {
            case HST_WAITING_FOR_IBUTTON : printf("HST_WAITING_FOR_IBUTTON"); break;
            case HST_LOOKING_FOR_BOAT    : printf("HST_LOOKING_FOR_BOAT"); break;
            case HST_WAITING_FOR_GAME_START : printf("HST_WAITING_FOR_GAME_START");
            break;
            case HST_PLAYING_GAME       : printf("HST_PLAYING_GAME"); break;
            case HST_CRUISING_POST_GAME : printf("HST_CRUISING_POST_GAME"); break;
            case HST_STANDING_DOWN      : printf("HST_STANDING_DOWN"); break;
        }
    }
    printf("\r\nCurrentEvent = ");
    switch(event) {
        case EV_NO_EVENT : printf("EV_NO_EVENT"); break;
        case EV_ENTRY    : printf("EV_ENTRY"); break;
        case EV_ENTRY
case EV_EXIT : printf("EV_EXIT"); break;
case EV_ERROR     : printf("EV_ERROR"); break;
case EV_NO_ACTION : printf("EV_NO_ACTION"); break;
case EV_IBUTTON   : printf("EV_IBUTTON"); break;
case EV_STAND_DOWN: printf("EV_STAND_DOWN"); break;
case EV_GAME_START: printf("EV_GAME_START"); break;
case EV_GAME_STOP : printf("EV_GAME_STOP"); break;
case EV_HARD_RESET: printf("EV_HARD_RESET"); break;
case EV_TMR_SEND  : printf("EV_TMR_SEND"); break;
case EV_TMR_LOST_COMM: printf("EV_TMR_LOST_COMM"); break;
case EV_MATCHED   : printf("EV_MATCHED"); break;
case EV_PLAY_ON   : printf("EV_PLAY_ON"); break;
case EV_NEXT      : printf("EV_NEXT"); break;
case EV_NEW_XBEE : printf("EV_NEW_XBEE"); break;
}

motor.h

#ifndef MOTOR
#define MOTOR

//FUNCTION PROTOTYPES
// Public Function Prototypes
void InitPWM(void);
void SetMotor(char motorID, char direction, char duty);
void Stop(void); //stops both motors

#endif

motor.c

//-- code courtesy of WeinerMeister--//
//----------------------------------//
//motor.c contains any code that is specific to the boat, including propeller control

#include "headers.h"

void InitPWM(void){
    //Initialize the clock
    PWMSCLA = POSTSCALER; //scale the A clock by / (3*2)
    PWMPRCLK |= 1; //use clock A with M/4 scalar (write to bit 1)

    //Initialize PWM for motor 1 (T0)
    PWME |= BIT0HI; //enable PWM on bit 0
    MODRR |= BIT0HI; //map T0 to PWM
    PWMCLK |= BIT0HI; //use SA (scaled clock)
    PWMPOL |= BIT0HI; //select the PWM polarity. 1 = output initially high
    PWMPER0 = MOTOR_PWM_PERIOD; //contains the count of the total number of cycles on
clock A or SA that will constitute the total period for PWM channel 0
    PWMDTY0 = DEFAULT_MOTOR_DUTY; //contains the count of the total number of cycles on
either clock A or SA that will constitute the active period for PWM channel 0

    //Initialize PWM for motor 2 (T1)
    PWME |= BIT1HI; //enable PWM on bit 1
    MODRR |= BIT1HI; //map T1 to PWM
    PWMCLK |= BIT1HI; //use SA (scaled clock)
    PWMPOL |= BIT1HI; //select the PWM polarity. 1 = output initially high
    PWMPER1 = MOTOR_PWM_PERIOD; //contains the count of the total number of cycles on
clock A or SA that will constitute the total period for PWM channel 0
    PWMDTY1 = DEFAULT_MOTOR_DUTY; //contains the count of the total number of cycles on
either clock A or SA that will constitute the active period for PWM channel 0
}
//Sets the duty cycle of the given motor and sets the direction output
//motorID = LEFT or RIGHT
//direction = FORWARD or BACKWARD
//duty = 0 to 100
void SetMotor(char motorID, char direction, char duty){
  //calculate the number of clock ticks to give powers to the motor
  unsigned int dutyTicks;
  dutyTicks = (MOTOR_PWM_PERIOD * duty)/100;
  //check to make sure the parameters are in bounds
  if(duty < 0 || duty > 100){
    printf("ERR: duty out of bounds in SetMotor \r\n");
    return; //failure
  }
  if(!((direction == FORWARD) || (direction == BACKWARD))){
    printf("ERR: direction must be forward or backward \r\n");
    return; //failure
  }
  if(!((motorID == R_MOTOR) || (motorID == L_MOTOR) || (motorID == BOTH_MOTORS))){
    printf("ERR: unknown motorID given \r\n");
    return; //failure
  }
  //Set the direction and PWM based on which motor and which direction are selected
  if((motorID == L_MOTOR)||(motorID == BOTH_MOTORS)){
    if(direction == FORWARD){
      PTT |= BIT5HI; //set direction pin output
      PWMDTY1 = (char)(MOTOR_PWM_PERIOD-dutyTicks); //set motor PWM
      //registers as prescribed by the PWM subsystem. Invert duty when direction pin is high.
      //printf("I'm setting left motor duty to: %d (INVERSE) \n\r",dutyTicks);
    }else{
      PTT &= BIT5LO;
      PWMDTY1 = (char)dutyTicks;
      //printf("I'm setting left motor duty to: %d \n\r",dutyTicks);
    }
  }
  if((motorID == R_MOTOR) || (motorID == BOTH_MOTORS)){
    if(direction == FORWARD){
      PTT &= BIT4HI;
      PWMDTY0 = (char)(MOTOR_PWM_PERIOD-dutyTicks); //set motor PWM
      //registers as prescribed by the PWM subsystem. Invert duty when direction pin is high.
      //printf("I'm setting right motor duty to: %d (INVERSE) \n\r",dutyTicks);
    }else{
      PTT &= BIT4LO;
      PWMDTY0 = (char)dutyTicks;
      //printf("I'm setting right motor duty to: %d \n\r",dutyTicks);
    }
  }
  //stop the boat in its tracks
  void Stop(void){
    //printf("   Now stopping\r\n");
    SetMotor(BOTH_MOTORS, FORWARD, 0);
  }
}

servo.h

#ifndef SERVO
#define SERVO

//FUNCTION PROTOTYPES
void InitServoPWM(void);
void SetServoPosition(char position, char servo_id);

#endif SERVO
//------- servo.c -------//
//-- code courtesy of BurgerStache --//
//-------//-------//-------//

//Standard Libraries
#include "headers.h"

//Initializes the PWM subystem for servos on the HELM
void InitServoPWM(void)
{
    //Initialize the clock
    PWMSCLA = POSTSCALER_A; //scale the A clock by / (2*75)
    PWMPRCLK |= 0x04;   //use clock A with M/16 scalar

    //Initialize PWM for servo
    PWME |= (BIT0HI | BIT1HI | BIT4HI); //enable PWM on bits 0, 1, 4
    MODRR |= (BIT0HI | BIT1HI | BIT4HI); //map PWM to port U on 0, 1, 4
    PWMPOL |= (BIT0HI | BIT1HI | BIT4HI); //use SA (scaled clock)
    PWMPOL |= (BIT0HI | BIT1HI | BIT4HI); //select the PWM polarity. 1 = output
initially high
    PWMCAE |= (BIT0HI | BIT1HI | BIT4HI); //center align the PWM signal

    //Set the period for all three PWM channels
    PWMPER0 = SERVO_PWM_PERIOD;
    PWMPER1 = SERVO_PWM_PERIOD;
    PWMPER4 = SERVO_PWM_PERIOD;

    //Set the initial duty cycle for all three PWM channels
    PWMDTY0 = SERVO_INIT_DUTY;
    PWMDTY1 = SERVO_INIT_DUTY;
    PWMDTY4 = SERVO_INIT_DUTY;

    //contains the count of the total number of cycles
    on either clock A or SA that will constitute the active period for PWM channel 0
}

//Public function to allow servos to be positioned to a positions 0 through 19
void SetServoPosition(char team, char servo_id)
{
    char position;

    //Scale the position input to a duty cycle and check to make sure it is not too high
    if(servo_id == ACTIVE_BASE_SERVO)
        position = team;
    else if(servo_id == RED_BOAT_NUM_SERVO)
        switch(team)
        {
            case 0:
                position = 5;
                break;
            case 12:
                position = 9;
                break;
            case 10:
                position = 13;
                break;
            case 8:
                position = 17;
                break;
            case 6:
                position = 21;
                break;
            case 4:
                position = 24;
                break;
            case 2:
                position = 27;
                break;
        }
else if(servo_id == BLUE_BOAT_NUM_SERVO)
    switch(team) {
        case 11:
            position = 6;
            break;
        case 9:
            position = 9;
            break;
        case 7:
            position = 12;
            break;
        case 5:
            position = 15;
            break;
        case 3:
            position = 19;
            break;
        case 1:
            position = 22;
            break;
        case 0:
            position = 26;
            break;
    }

    if(position > SERVO_MAX_DUTY)
        position = SERVO_MAX_DUTY;

    printf("Setting servo id %d to position %d, team %d\n", servo_id, position, team);
    //Update the appropriate PWM duty
    if(servo_id == BLUE_BOAT_NUM_SERVO)
        PWMDTY0 = position;
    if(servo_id == RED_BOAT_NUM_SERVO)
        PWMDTY1 = position;
    if(servo_id == ACTIVE_BASE_SERVO)
        PWMDTY4 = position;
}

//------------------Test Routine----------------------/
#ifdef SERVO_TEST

void main(void)
{
    char i;
    InitAll();

    //Cycle through variable pulse lengths
    while(TRUE)
    {
        for(i=0; i<30; i++)
        {
            //SetServoPosition(i, ACTIVE_BASE_SERVO);
            //SetServoPosition(i, RED_BOAT_NUM_SERVO);
            SetServoPosition(i, BLUE_BOAT_NUM_SERVO);
            printf("Position: %d \n", i);
            Wait(1000);
        }
    }
}
#endif
# xbee.h

#ifndef xbee
#define xbee

// Function Prototypes
//public functions
void InitSCI (void);
void Send218Data(unsigned char destination, unsigned char byte0, unsigned char byte1, unsigned char byte2);
unsigned char CheckXbeeRX(void);
unsigned char GetXbeeByte0(void);
unsigned char GetXbeeByte1(void);
unsigned char GetXbeeByte2(void);
unsigned char GetTeamNumber(void);
void ImprintPartner(void);
void SimulateIbutton(unsigned char us);

//private functions
static void CheckPingBack(void);
static void ResetChecksum(void);
static unsigned char GetChecksum(void);
static void SendData(unsigned char data);
static void ProcessNewData(void);
#endif

xbee.c

// xBee preliminary testing
//
// Team Burgerstache
// Created May 7, 2008
//
#include "headers.h"

//global variables
extern unsigned char GWhoAmI;

//module variables
static unsigned char CheckSum;
static unsigned char RXDataBuffer[XBEE_MESSAGE_SIZE]; //12 bytes to match 218 comm standard, plus one extra for good luck
static unsigned char RXDataBufferIndex = 0;
static unsigned char RXFlag = FALSE;
static unsigned char RXSourceMSB = 0;  //Byte 5
static unsigned char RXSourceLSB = 0;  //Byte 6
static unsigned char RXbyte0 = 0;  //Byte 9
static unsigned char RXbyte1 = 0;  //Byte 10
static unsigned char RXbyte2 = 0;  //Byte 11
static unsigned char MyPartnerDestMSB = 0x00;
static unsigned char MyPartnerDestLSB = 0x00;

/*
//Q: what if packets are dropped? Are we getting them in order? WTF?
*/

// Initialization
void InitSCI (void) {
    printf("Initializing SCI.\r\n");
    // CONFIGURE SCI
    SCI1BDH = 0x00;  // write SCI1BDH - want it to be 0
    SCI1BDL = BAUD_BITS;  // write SCI1BDL - this is 156
    SCI1CR1 = 0x00;  // write SCI1CR1 - clear register (all zeros) for proper config
SCICR2 |= BIT5HI;          // bit 5 for receive interrupt
SCICR2 |= BIT3HI | BIT2HI;  // bit 2 and 3 for tx/rx enable
SCICR2 |= BIT4HI;          // bit 4 for idle line interrupt

// Port S
DDRS &= BIT2LO;  // Input
DDRS |= BIT3HI;  // Output... not sure if we need this and should do a master
initialize elsewhere

    // INTERRUPTS
    EnableInterrupts;

    // Polling function that checks the xbee for new data
    // if there is new data, it is processed and put into module variables
    // returns true if new data was intercepted, false otherwise
unsigned char CheckXbeeRX(void) {
    if(RXFlag == TRUE) {
        RXFlag = FALSE;
        ProcessNewData(); // put data into module variables and print them out
        CheckPingBack(); // ping the admiral back if we need to do so
        printf("Data arrival on xbee complete.\r\n");
        return TRUE;
    }
    else {
        printf("No new data received! \r\n");
        return FALSE;
    }
}

// if the admiral pings us, ping it back
static void CheckPingBack(void){
    unsigned char byte0, byte1, byte2;
    // CHECK FOR SIGNEDNESS!!!!!!!
    /*
    printf("Checking to see if we should ping back to admiral...\r\n");
    printf("RXbyte0 = %d \r\n",RXbyte0);
    printf("ADMIRAL = %d \r\n",ADMIRAL);
    printf("RXbyte2 = %d \r\n",RXbyte2);
    printf("ADMIRAL_PING = %d \r\n",ADMIRAL_PING);
    */
    if((RXbyte0 == ADMIRAL) && (RXbyte2 == ADMIRAL_PING)){
        byte0 = PING_RESPONSE;
        if(GWhoAmI == IAMBOAT) { // check the boat SM if we're a boat
            if(QueryBoatSM() == BST_WAITING_FOR_IBUTTON)
                byte1 = 0x01; // if waiting for iButton
            else if(QueryBoatSM() == BST_LOOKING_FOR_HELM)
                byte1 = 0x02; // if iButton read and waiting for pairing
            else
                byte1 = 0x03; //I am a helm
            if(QueryHelmSM() == HST_WAITING_FOR_IBUTTON)
                byte1 = 0x01; // if waiting for iButton
            else if(QueryHelmSM() == HST_LOOKING_FOR_BOAT)
                byte1 = 0x02; // if iButton read and waiting for pairing
            else
                byte1 = 0x04; // if paired
        }
        byte2 = MyPartnerDestLSB; // default is 0x00
        printf("Sending admiral response to ping \r\n");
        Send218Data(TO_ADMIRAL,byte0, byte1, byte2); // do the pingback!
    }
}

    // update partner because we did the ibutton dance
    // the source of the last message we processed is now our partner
void ImprintPartner(void){
    printf("Imprint Partner \r\n");
    MyPartnerDestMSB = RXSourceMSB;
    MyPartnerDestLSB = RXSourceLSB;
}

//returns the team number, which is based on the lower nibble of the boat's address
//returns 0 if the team is not yet chosen (partner not yet imprinted)
unsigned char GetTeamNumber(void){
    return MyPartnerDestLSB;
}

//simulates an ibutton read, pairing us with our own helm (or boat)
void SimulateIbutton(unsigned char us) {
    printf("Simulate Ibutton \r\n");
    if(us == IAMBOAT) {
        MyPartnerDestMSB = 0xBC;
        MyPartnerDestLSB = 0x04;
    }else{ //we are helm
        MyPartnerDestMSB = 0xAF;
        MyPartnerDestLSB = 0x04;
    }
}

//header
unsigned char GetXbeeByte0(void){
    return RXbyte0;
}

//nav
unsigned char GetXbeeByte1(void){
    return RXbyte1;
}

//parameters
unsigned char GetXbeeByte2(void){
    return RXbyte2;
}

//Transmit command
//header, navigation, special
//If broadcast is true send a broadcast, otherwise send it to our partner
void Send218Data(unsigned char destination, unsigned char byte0, unsigned char byte1,
    unsigned char byte2){
    unsigned char destMSB, destLSB;
    unsigned char checksum;
    unsigned char options = 0x00;
    printf("Sending data...\r\n");
    //send the damn data
    SendData(START_BYTE); //start delimiter
    SendData(LENGTH_MSB); //length MSB
    SendData(LENGTH_LSB); //length LSB
    ResetChecksum();
    SendData(API_TX); //API identifier (TX request 16-bit)
    SendData(FRAME_ID); //Frame ID
    //send destination bytes according to desired dest type
    if(destination == TO_BROADCAST){
        destMSB=0xFF; //Destination address
        destLSB=0xFF;
    }else if (destination == TO_ADMIRAL){
        destMSB=ADMIRAL_ADDRESS_MSB;
        destLSB=ADMIRAL_ADDRESS_LSB;
    }
    else {
        destMSB=MyPartnerDestMSB;
        destLSB=MyPartnerDestLSB;
    }
    //send destination data
    SendData(destMSB);
SendData(destLSB);
SendData(options); //options (was 0x01 for testing), it is now initialized as a variable above
SendData(byte0); //output data 1
SendData(byte1); //output data 2
SendData(byte2); //output data 3
checksum = GetChecksum();
SendData(checksum); //checksum

//print the data
printf("Sending byte 1: %x   Start Byte \n", START_BYTE);
printf("Sending byte 2: %x   Length MSB \n", LENGTH_MSB);
printf("Sending byte 3: %x   Length LSB \n", LENGTH_LSB);
printf("Sending byte 4: %x   API_TX \n", API_TX);
printf("Sending byte 5: %x   FRAME_ID \n", FRAME_ID);
printf("Sending byte 6: %x   Dest MSB \n", destMSB);
printf("Sending byte 7: %x   Dest LSB \n", destLSB);
printf("Sending byte 8: %x   Options \n", options);
printf("Sending byte 10: %x Data Byte 1 \n", byte1);
printf("Sending byte 11: %x Data Byte 2 \n", byte2);
printf("Sending byte 12: %x Checksum \n", checksum);
printf("Sending complete.\n");

// Interrupt routine for when we get new data
// take packets from the xbee and save them into an array
// then sets a flag high that tells us there is new xbee data available for processing
void interrupt _Vec_sci1 ReadData (void)
{
  unsigned char status;
  unsigned char new_data;

  status = SCI1SR1;
  new_data = SCI1DRL; //Read data (this also clears the flag)

  if(status & BIT5HI) //Check RDRF to see if we have good data
  {
    //printf("D"); //"D" - we have data
    // Process new data by putting it into the buffer array
    RXDataBuffer[RXDataBufferIndex] = new_data;
    RXDataBufferIndex++;

    if(RXDataBufferIndex >= (XBEE_MESSAGE_SIZE))
    {
      RXDataBufferIndex = 0; //reset index
      RXFlag = TRUE; //we have new data!
      //printf("F\n"); //"F" - buffer is full
    }
  }

  if(status & BIT4HI) //reset the index if there is an idle
  {
    //printf("I"); //"I" - line is idle
    RXDataBufferIndex = 0; //reset index
  }
}

// process new data
static void ProcessNewData(void)
{
  unsigned char index = 0;
  unsigned char binary[9];
}
//printf("Processing message now... \r\n");

//Store the values of the important bytes in module variables
RXSourceMSB = RXDataBuffer[4];
RXSourceLSB = RXDataBuffer[5];
RXbyte0 = RXDataBuffer[8];
RXbyte1 = RXDataBuffer[9];
RXbyte2 = RXDataBuffer[10];

/*
 * Prints out any data that arrives
 * while(index < XBEE_MESSAGE_SIZE)
 {   
   //dec2bin(RXDataBuffer[index],binary);  //convert the data to binary for debugging
   printf("Processing byte %d: %x    ", (index + 1), (RXDataBuffer[index]));
   switch(index+1){  //switch based on what byte we read
      case 1: printf("Start Delimiter"); break;
      case 2: printf("Length MSB"); break;
      case 3: printf("Length LSB"); break;
      case 4: printf("API Identifier"); break;
      case 5: printf("Source Addr MSB"); break;
      case 6: printf("Source Addr LSB"); break;
      case 7: printf("Signal strength"); break;
      case 8: printf("Options"); break;
      case 9: printf("Data Byte 0 - Header"); break;
      case 10: printf("Data Byte 1"); break;
      case 11: printf("Data Byte 2"); break;
      case 12: printf("Checksum"); break;
   }

   printf("\r\n");
   index++;
   
   printf("Processing complete! \r\n");
}

// Sends a byte of data over SCI
static void SendData(unsigned char data)
{
   //char binary[80];
   unsigned char dummy = 0;
   //dec2bin(data,binary);  //convert the data to binary for debugging
   while((SCI1SR1 & BIT6HI) == 0);  //TC = transmit complete (stall until this is 1)
   //do nada
   dummy = SCI1SR1;  //clears TDRE by reading sci
   SCI1DRL = data;  //actually sends

   //add data to our checksum
   CheckSum += data;

   //print out the data we're sending (in binary)
   //printf("Sending data: %s \r\n", binary);

}

//this function resets our checksum
static void ResetChecksum(void){
   CheckSum = 0;
}

//get the current checksum value, and print it
static unsigned char GetChecksum(void){
   unsigned char FinalCheckSum;
   FinalCheckSum = (0xFF - CheckSum);
}
//printf("FinalCheckSum = %x\r\n", FinalCheckSum);
return FinalCheckSum;
}

//--------- TEST CODE ------------
//checks for key presses, then broadcasts a message (different depending on key= s or b)
//uses interrupt-driven read, which is processed when 'r' is pressed
#ifdef XBEE_TEST
void main (void)
{
    unsigned char keyInput;

    printf("XBee Test Code! (press any key to cont) \r\n\n");    printf("press 's' to send, 'r' to receive \r\n\n");

    //Initialize all of our ports and SCI registers
    InitAll();

    while(TRUE){
        if(kbhit() != 0){
            keyInput = getchar(); //this makes it not go into a weird loop

            //send test
            if(keyInput == 's')
                Send218Data(TO_BROADCAST, 0xAA, 0xAA, 0xAA);
            if(keyInput == 'b')
                Send218Data(TO_BROADCAST, 0xBB, 0xBB, 0xBB);

            //receive test
            if(keyInput == 'r')
                CheckXbeeRX();
        }
        //printf("still looping\r\n");
    }
}
#endif

helpers.h

#ifndef HELPERS
#define HELPERS

//Function Prototypes
//timer functions
void Wait(int ticks);
void SetTimer(unsigned char timer, int ticks);
unsigned char CheckTimerExpired(unsigned char timer);
unsigned char CheckSendTimer(void);

//other helper functions
void PrintDecAsBin(unsigned char decimal);
void TestDecToBin(void);    void dec2bin(unsigned char decimal, unsigned char *binary);
#endif

PIC Code

ibutton.asm

; ME218C Project - iButton Code
; Adam Leeper
; 5/08/08

; GENERAL DESCRIPTION:
; this code reads the iButton, communicates its value
; to the HC12, and also controls the iButton reader's
; LED light and a buzzer which sounds to indicate a successful
; read. Error checking takes place in the HC12.

; administrative stuff:

list P=PIC16F690
#include "p16F690.inc"
__config (_CP_OFF & _WDT_OFF & _PWRTE_ON & _HS_OSC)

; variable definitions:
DCount   EQU    0x2D
ACount   EQU    0x2F
TCount         EQU    0x2C
BitVal   EQU    0x2E
TempByte EQU     0x20
Byte1    EQU    0x21
Byte2    EQU    0x22
Byte3    EQU    0x23
Byte4    EQU    0x24
Byte5    EQU    0x25
Byte6    EQU    0x26
Byte7    EQU    0x27
Byte8    EQU    0x28

; port definitions:
EnablePort     EQU    PORTC
EnablePin      EQU    0 ; signal from the Master that an iButton should be read
ButtonPort     EQU    PORTA
ButtonPin      EQU    0 ; the open-drain port used for the iButton reader
Carry          EQU  0
ClockPort      EQU    PORTC ; the clock pin for the synchronous communiction
ClockPin       EQU    1 ; of iButton data to the HC12
InfoPort       EQU    PORTC ; the info pin for the same...
InfoPin        EQU    2
LEDPort        EQU    PORTA ; controls the reader's LED
LEDPin         EQU    2
LEDSink         EQU     1
BuzzPort       EQU    PORTC ; controls a buzzer
BuzzPin        EQU    4

ConfigA   equ  b'11111000' ; Config RA0 as output
ConfigB   equ  b'11111111' ; Config placeholder
ConfigC   equ  b'11111001' ; Config RC1-RC4 as outputs

SDI             equ     4         ; SDI
SCK             equ     6         ; SCK
SS              equ     6         ; SS
SDO             equ     7         ; SDO

#define     iButton     ButtonPort,ButtonPin
#define     iLED        LEDPort,LEDPin
#define     LED_ON     BSF iLED
#define     LED_OFF    BCF iLED
#define     SS_LOW     BCF PORTC,SS
#define     SS_HIGH    BSF PORTC,SS

ORG   0
Main:

CLRF PORTA
CLRF PORTB
CLRF PORTC

; Set up pins for Tx/Rx...
CALL Bank2 ; move to Bank2, for ANSEL
CLRF ANSEL ; set all pins to digital
CLRF ANSELH ; set all pins to digital

; Set up pins for Input/Output
BANKSEL TRISA ; move to Bank1, for TRIS
MOVWF TRISA ; Write PortA I/O
MOVLW ConfigA ; load ConfigA
MOVWF TRISB ; Write PortB I/O
MOVLW ConfigB
MOVWF TRISC ; Write PortC I/O
MOVLW ConfigC

; Initialize timer
CALL Bank0 ; move to Bank 0 for Timer1 stuff
CLRF T1CON ; Clear all timer 1 settings
CLRF TMR1H ; Clear timer1 high byte
CLRF TMR1L ; Clear timer1 low byte
BCF T1CON,T1CKPS1 ; set prescaler to 1:8
BCF T1CON,T1CKPS0 ; ''
BSF T1CON,TMR1ON ; turn on timer 1, starts to increment
MOVLW b'00001000' ; Set output compare to software interrupt
MOVWF CCP1CON

; Initialize SSP
BANKSEL SSPSTAT ; Bank 1
CLRF SSPSTAT ; SMP = 0, CKE = 0, and clear status bits
BANKSEL SSPCON
MOVLW b'00110010' ; Set up SPI port, Master mode, Fosc/64,
MOVWF SSPCON ; Write it to register
BSF PORTC,SDO
NOP
BSF PORTB,SCK
NOP
BSF PORTC,SS
BANKSEL TRISC
BSF TRISB,4 ; SDI
BCF TRISB,6 ; SCK
BCF TRISC,6 ; SS
BCF TRISC,7 ; SDO

CALL Bank0 ; move to Bank0, ready to go
CLRF Byte2
CLRF Byte3

Start:

BTFSS EnablePort,EnablePin ; we wait until the HC12 says it
GOTO Start ; wants to read an iButton...

Reset_State:

;MOVF SSPBUF,W ; Read SSPBUF to avoid setting overflow flag
;MOVF Byte2,W ; For ME218, we care about Byte 2 and Byte 3
;MOVWF SSPBUF

LED_ON
CALL Wait750ms ;
LED_OFF
CALL Wait750ms ;

CALL Wait490
CALL SetOUT
BCF iButton ; pulse line low
CALL Wait490 ; wait for 500us
CALL SetIN ; float line
BSF iButton
CALL Wait50 ; wait for iButton to respond with presence
BTFSC iButton ; the line is pulled low here if
CLRF ACount ; an iButton is present.
However, since the mechanical
INCF ACount,F ; bounce of the contact lasts
for a while, I make sure
CALL Wait490 ; that 20 consecutive
presence pulses have been seen
MOVLW 0xFD ; before moving on!
ADDWF ACount,W
BTFSS STATUS,Carry
GOTO Start

Send_Reset:
    BSF iButton
    CALL SetOUT
    CALL Wait490
    CALL Wait490
    BCF iButton
    CALL Wait490
    BSF iButton
    CALL Wait490

Send0x33:
    CALL SetOUT
    CALL Write1 ; this is the command to ask for the iButton's
    CALL Write1 ; unique ID number
    CALL Write0
    CALL Write0
    CALL Write1
    CALL Write1
    CALL Write0
    CALL Write0
    CALL Wait490

Get8Bytes: ; getting the 8 bytes...
    CALL GetByte ;
    MOVWF Byte1 ; Family Code Byte
    CALL GetByte
    MOVWF Byte2 ; SS Byte 1
    CALL GetByte
    MOVWF Byte3 ; SS Byte 2
    CALL GetByte
    MOVWF Byte4 ; SS Byte 3
    CALL GetByte
    MOVWF Byte5 ; SS Byte 4
    CALL GetByte
    MOVWF Byte6 ; SS Byte 5
    CALL GetByte
    MOVWF Byte7 ; SS Byte 6
    CALL GetByte
    MOVWF Byte8 ; CRC Byte

Send8Bytes:
    BANKSEL SSPSTAT
    BCF SSPSTAT,BF
    BANKSEL PORTA

    SS_LOW
    MOVF SSPBUF,W ; Read SSPBUF to avoid setting overflow flag
    MOVF Byte2,W ; For ME218, we care about Byte 2
    MOVWF SSPBUF
    BANKSEL SSPSTAT

Xmit_Loop2:
    BTFSS SSPSTAT,BF
    GOTO Xmit_Loop2
    BANKSEL PORTA
SS_HIGH

CALL Wait70

SS_LOW

MOVF SSPBUF,W ; Read SSPBUF to avoid setting overflow flag
MOVF Byte3,W ; For ME218, we care about Byte 3 also
MOVWF SSPBUF

BANKSEL SSPSTAT

Xmit_Loop3:

BTFSS SSPSTAT,BF
GOTO Xmit_Loop3

BANKSEL PORTA

; The old janky way to do it
MOVF Byte1,W ; sending the 8 bytes to the HC12...
CALL SendByte
MOVF Byte2,W
CALL SendByte
MOVF Byte3,W
CALL SendByte
MOVF Byte4,W
CALL SendByte
MOVF Byte5,W
CALL SendByte
MOVF Byte6,W
CALL SendByte
MOVF Byte7,W
CALL SendByte
MOVF Byte8,W
CALL SendByte

Finish:

CALL Wait750ms ; at the end, we give the HC12 some time to think
CALL Wait750ms ; and then see if it still needs an ibutton read
CALL Wait750ms
CALL Wait750ms
;CALL Wait490
GOTO Start

; ************************************************************

SendByte: ; starting with a byte in the W register

MOVWF TempByte ; we store that value in "TempByte"
MOVWF 0x08
MOVWF ACount

SendLoop: ; we loop the following 8 times:

BTFSS TempByte,0 ; we set the info line to follow the
BCF InfoPort,InfoPin ; value of the LSB of TempByte

BTFSC TempByte,0

BSF InfoPort,InfoPin

CALL Wait5 ; we wait a little

BSF ClockPort,ClockPin ; and pulse the clock, signalling the
Call Wait50 ; HC12 to read

BCF ClockPort,ClockPin

Call Wait5

RFF TempByte,1 ; we then rotate the file to the

right,

DECFSZ ACount,1 ; placing the next bit in the LSB

spot

GOTO SendLoop ; and repeat!

RETURN

; ************************************************************

GetByte:

LED_ON

MOVVLW 0x08

MOVWF ACount

CLRF TempByte

ByteLoop: ; we loop the following 8 times:
RRF TempByte,F ; we rotate our result register to the left
CALL RW1 ; call the read function, which sets "BitVal"
BTFSC BitVal,0 ; then read BitVal and change the MSB
BSF TempByte,7 ; of TempByte accordingly.
DECFSZ ACount,F
GOTO ByteLoop ; and do it again!
MOVF TempByte,W
LED_OFF
RETURN

; ************************************************************
Write0: ; writing zero is just a long low followed by a short hi:
BCF iButton
CALL SetOUT
CALL Wait50
CALL SetIN
CALL Wait25
RETURN

Write1: ; writing one is a short low followed by a long hi:
BCF iButton
CALL SetOUT
CALL Wait5
CALL SetIN
CALL Wait70
RETURN

RW1: ; reading a bit looks like writing a 1, but checking
CLRF BitVal
BCF iButton
CALL SetOUT
;LED_ON
CALL Wait5
CALL SetIN
CALL Wait5
BTFSC iButton
BSF BitVal,0
;LED_OFF
CALL Wait50
CALL Wait5
RETURN

; ************************************************************
Bank*
; These routines set the STATUS register with the
; correct bits to move to the desired bank.
;**********************************************************
Bank0: ; Sets RP1,RP0 = 0,0 so we move to Bank0
BCF STATUS,RP1
BCF STATUS,RP0
RETURN

Bank1: ; Sets RP1,RP0 = 0,1 so we move to Bank1
BCF STATUS,RP1
BSF STATUS,RP0
RETURN

Bank2: ; Sets RP1,RP0 = 1,0 so we move to Bank2
BSF STATUS,RP1
BCF STATUS,RP0
RETURN

Bank3: ; Sets RP1,RP0 = 1,1 so we move to Bank3
BSF STATUS,RP1
BSF STATUS,RP0
RETURN
; End of Bank setting functions
;**********************************************************
SetIN:

BANKSEL TRISA
BSF TRISA,ButtonPin

BANKSEL PORTA
RETURN

SetOUT:
BANKSEL TRISA
BCF TRISA,ButtonPin
BANKSEL PORTA
RETURN

; *************************************

Wait750ms:
BANKSEL T1CON ; move to Bank 0 for Timer1 stuff
MOVLW 0x10
MOVWF TCount
MOVLW 0xFF
MOVF CCPR1H
MOVWF 0xFF
MOVF CCPR1L

CLRF T1CON ; Clear all timer 1 settings
CLRF TMR1H ; Clear timer1 high byte
CLRF TMR1L ; Clear timer1 low byte
BSF T1CON,TMR1ON ; turn on timer 1, starts to increment

Loop: ;BTFSS PIR1,TMR1IF ; Check timer overflow flag
BTFSS PIR1,CCP1IF ; Check for output compare flag
GOTO Loop ; Loops until the timer compares
BCF PIR1,CCP1IF ; Reset timer1 CCP flag
DECFSZ TCount,F ;
GOTO Loop ; Loops until the timer compares 50 times
CLRF T1CON ; Turn off timer1
RETURN

; **************************************

Wait490:
BANKSEL T1CON ; move to Bank 0 for Timer1 stuff
MOVLW 0x09
MOVWF CCPR1H
MOVLW 0xC4
MOVWF CCPR1L

CLRF T1CON ; Clear all timer 1 settings
CLRF TMR1H ; Clear timer1 high byte
CLRF TMR1L ; Clear timer1 low byte
BSF T1CON,TMR1ON ; turn on timer 1, starts to increment

Loop490 ;BTFSS PIR1,TMR1IF ; Check timer overflow flag
BTFSS PIR1,CCP1IF ; Check for output compare flag
GOTO Loop490 ; Loops until the timer compares
BCF PIR1,CCP1IF ; Reset timer1 CCP flag
CLRF T1CON ; Turn off timer1
RETURN

; **************************************

Wait50:
BANKSEL T1CON ; move to Bank 0 for Timer1 stuff
MOVLW 0x00
MOVWF CCPR1H
MOVLW 0xC4
MOVWF CCPR1L

CLRF T1CON ; Clear all timer 1 settings
CLRF TMR1H ; Clear timer1 high byte
CLRF TMR1L ; Clear timer1 low byte
BSF T1CON,TMR1ON ; turn on timer 1, starts to increment

Loop50 ;BTFSS PIR1,TMR1IF ; Check timer overflow flag
BTFSS PIR1,CCP1IF ; Check for output compare flag
GOTO_loop50 ; Loops until the timer compares
BCF PIR1,CCP1IF ; Reset timer1 CCP flag
CLRF T1CON ; Turn off timer1
RETURN

; ****************************************
Wait5:
BANKSEL T1CON ; move to Bank 0 for Timer1 stuff
MOVLW 0x00
MOVWF CCP1H
MOVLW 0x05 ; 1 * 5 = 5 = 0x05
MOVWF CCPR1L

CLRF T1CON ; Clear timer1 settings
CLRF TMR1H ; Clear timer1 high byte
CLRF TMR1L ; Clear timer1 low byte
BSF T1CON,TMR1ON ; turn on timer 1, starts to increment
Loop5
; BTFSS PIR1,TMR1IF ; Check timer overflow flag
BTFSS PIR1,CCP1IF ; Check for output compare flag
GOTO_loop5 ; Loops until the timer compares
BCF PIR1,CCP1IF ; Reset timer1 CCP flag
CLRF T1CON ; Turn off timer1
RETURN

; ****************************************
Wait70:
BANKSEL T1CON ; move to Bank 0 for Timer1 stuff
MOVLW 0x01
MOVWF CCP1H
MOVLW 0x45 ; 65 * 5 = 325 = 0x145
MOVWF CCPR1L

CLRF T1CON ; Clear timer1 settings
CLRF TMR1H ; Clear timer1 high byte
CLRF TMR1L ; Clear timer1 low byte
BSF T1CON,TMR1ON ; turn on timer 1, starts to increment
Loop70
; BTFSS PIR1,TMR1IF ; Check timer overflow flag
BTFSS PIR1,CCP1IF ; Check for output compare flag
GOTO_loop70 ; Loops until the timer compares
BCF PIR1,CCP1IF ; Reset timer1 CCP flag
CLRF T1CON ; Turn off timer1
RETURN

; ****************************************
Wait25:
BANKSEL T1CON ; move to Bank 0 for Timer1 stuff
MOVLW 0x00
MOVWF CCP1H
MOVLW 0x7D ; 25 * 5 = 125 = 0x7D
MOVWF CCPR1L

CLRF T1CON ; Clear timer1 settings
CLRF TMR1H ; Clear timer1 high byte
CLRF TMR1L ; Clear timer1 low byte
BSF T1CON,TMR1ON ; turn on timer 1, starts to increment
Loop25
; BTFSS PIR1,TMR1IF ; Check timer overflow flag
BTFSS PIR1,CCP1IF ; Check for output compare flag
GOTO_loop25 ; Loops until the timer compares
BCF PIR1,CCP1IF ; Reset timer1 CCP flag
CLRF T1CON ; Turn off timer1
RETURN