Contents

lib4.c

```c
#include "lib4.h"
#include "sensing.h"
#include "driving.h"
#include "oi.h"
#include "fixedqueue.h"
#include "irobserial.h"
#include "irchar.h"
#include "irobled.h"

#define PID_DT  (IROB_PERIOD_MS)

#define PRED    (irPrevRegion() & IR_MASK_RED_BUOY)
#define PGREEN  (irPrevRegion() & IR_MASK_GREEN_BUOY)
#define PFIELD  (irPrevRegion() & IR_MASK_FORCE_FIELD)
#define PANY    (PRED || PGREEN || PFIELD)
#define PALL    (PRED && PGREEN && PFIELD)
#define RED     (irRegion() & IR_MASK_RED_BUOY)
#define GREEN   (irRegion() & IR_MASK_GREEN_BUOY)
#define FIELD   (irRegion() & IR_MASK_FORCE_FIELD)
#define ANY     (RED || GREEN || FIELD)
#define ALL     (RED && GREEN && FIELD)

//#define CHARGING    (getSensorInt16(SenCurr1) >= CURRENTTHOLD)
#define CHARGING    (getSensorUint8(SenChAvailable))

int16_t utk = 0;
int16_t etk = 0;
int16_t etk_1 = 0;
int16_t esum = 0;
FixedQueue esumQueue = 0;

uint8_t bumpDrop = 0;
uint8_t prevBumpDrop = 0;

uint8_t started = 0;
uint8_t docking = 0;
uint8_t comingFromFront = 0;
uint8_t dockingFinal = 0;
uint8_t onDock = 0;

int16_t jimmyAngle = 0;

/**
 * initilaization function for a pid controller.
 */
void pidSetup(void) {
    while (esumQueue == 0) {
        esumQueue = newFixedQueue(PID_QSIZE);
    }
}
/**
 * A function to clear the value of the esumQueue
 * which is used in the integral calculations for
 * the PID controller.
```

```
 */
void pidCleanup(void) {
    if (esumQueue != 0) {
        freeFixedQueue(esumQueue);
        esumQueue = 0;
    }
}
/**
 * Takes the next input for the pid controler
 * utilizes constants:
 * PID_SET_POINT - the set point or goal
 * PID_KP - the multiplier for the current distance
 * PID_KI - the multiplier for the "integral" term
 * PID_KD - the multiplier for the "derivative" term
 * This then sets utk which is essentially the result
 * of this calculation
 * Important notes:
 * The derivative terms only look at this and the previous value
 * the integral term currently has no time mitigation and uses
 * a fixed size queue (now has damping based on size)
 *
 * @param vtk the current value for the pid controller
 */
void pidStep(uint16_t vtk) {
    etk_1 = etk;
    etk = ((int16_t)vtk) - PID_SET_POINT;
    esum += etk;
    esum -= pushPop(esumQueue, etk);
    int16_t p = PID_KP*etk;
    int16_t i = PID_KI*esum*PID_DT / PID_QSIZE; // damping
    int16_t d = PID_KD*(etk-etk_1)/PID_DT;
    irobprintf("etk_1: %d\netk: %d\nesum: %d\nutk: %d\np: %d\ni: %d\nd: %d\n",
            etk_1, etk, esum, utk, p, i, d);
    utk = p + i + d;
}

/**
 * A drive function which utilizes the utk value (modified in pidStep)
 * this also utilizes
 * DRIVE_DIVISOR - directly mitigates UTK
 * SPEED - the default speed
 */
void updateMotors(void) {
    int16_t deltaDrive = utk / DRIVE_DIVISOR;
    driveDirect(SPEED - deltaDrive, SPEED + deltaDrive);
}

// Do while turning after bump
void doWhileTurning(void) {
    updateSensors();
    uint8_t bumpDrop = getSensorUint8(SenBumpDrop);
    if (bumpDrop & MASK_WHEEL_DROP) {
        driveStop();
    }
    if (docking) {
        updateIR();
        dockingDiagnostics();
```

```
112     }
113 }
114
115 void move(int16_t distance) {
116     int16_t speed = docking ? DOCKING_SPEED : SPEED;
117     driveDistanceTFunc(speed, distance, &doWhileTurning,
118             UPDATE_SENSOR_DELAY_PERIOD, UPDATE_SENSOR_DELAY_CUTOFF);
119 }
120 void turn(int16_t radius, int16_t angle) {
121     int16_t speed = docking ? DOCKING_SPEED : SPEED;
122     driveAngleTFunc(speed, radius, angle, &doWhileTurning,
123             UPDATE_SENSOR_DELAY_PERIOD, UPDATE_SENSOR_DELAY_CUTOFF);
124 }
125
126 uint8_t noBump(void) {
127     return !(getSensorUint8(SenBumpDrop) & MASK_BUMP) && notCharging();
128 }
129 void jimmyBump(void) {
130     // Drive forward until bump
131     drivePredicateFunc(JIMMY_SPEED, RadStraight, &noBump,
132             &doWhileTurning, UPDATE_SENSOR_DELAY_PERIOD,
133             UPDATE_SENSOR_DELAY_CUTOFF);
134 }
135 uint8_t notCharging(void) {
136     return !CHARGING;
137 }
138 void jimmyTurn(int16_t radius) {
139     // Turn until charging
140     driveAnglePFunc(JIMMY_SPEED, radius, jimmyAngle, &notCharging,
141             &doWhileTurning, UPDATE_SENSOR_DELAY_PERIOD,
142             UPDATE_SENSOR_DELAY_CUTOFF);
143 }
144 void jimmy(void) {
145     // Increase angle every time
146     jimmyAngle += JIMMY_ANGLE;
147     // Turn right, then back to center
148     jimmyTurn(RadCW);
149     jimmyTurn(RadCCW);
150     // Bump the dock
151     jimmyBump();
152     // Turn left, then back to center
153     jimmyTurn(RadCCW);
154     jimmyTurn(RadCW);
155     // Bump the dock
156     jimmyBump();
157 }
158
159 void dock(void) {
160     if (!dockingFinal && !PGREEN && GREEN) {
161         // Move an extra robot radius
162         move(IROB_RAD_TURN);
163         // Turn to line up
164         turn(RadCW, FIELD_TURN);
165         dockingFinal = 1;
166     } else if (dockingFinal && RED && !GREEN) {
167         // Course correction
168         drive(DOCKING_SPEED, RadCCW);
```

```
169     } else if (dockingFinal && !RED && GREEN) {
170         // Course correction
171         drive(DOCKING_SPEED, RadCW);
172     } else {
173         // Normally just go straight
174         drive(DOCKING_SPEED, RadStraight);
175     }
176 }
177 
178 void dockingDiagnostics(void) {
179     // Robot LEDs for IR fields
180     robotLedSetBits(NEITHER_ROBOT_LED);
181     powerLedSet(POWER_LED_ORANGE, 0);
182     if (smoothRed() > 0x60)     robotLedOn(PLAY_ROBOT_LED);
183     if (smoothGreen() > 0x60)   robotLedOn(ADVANCE_ROBOT_LED);
184     if (irRegion() & IR_MASK_FORCE_FIELD) powerLedSet(POWER_LED_ORANGE, 0xFF);
185     // Command module LEDs for charging.
186     cmdLED1Set(0);
187     cmdLED2Set(0);
188     if (CHARGING) {
189         cmdLED1Set(1);
190     } else {
191         cmdLED2Set(1);
192     }
193 }
194 
195 // Called by irobPeriodic
196 void iroblifePeriodic(void) {
197     // Get bump & wheel drop sensor
198     prevBumpDrop = bumpDrop;
199     bumpDrop = getSensorUint8(SenBumpDrop);
200     // IR
201     updateIR();
202     dockingDiagnostics();
203     if (onDock) {
204         // Final connection on dock
205         if (CHARGING) {
206             driveStop();
207         } else {
208             jimmy();
209         }
210     } else if (bumpDrop & MASK_WHEEL_DROP) {
211         // Cliff
212         driveStop();
213     } else if (bumpDrop & MASK_BUMP) {
214         if (dockingFinal) {
215             // We are now on the dock
216             driveStop();
217             onDock = 1;
218         } else {
219             // Turn until no longer bumping
220             drive(SPEED, RadCCW);
221             started = 1;
222         }
223     } else if (prevBumpDrop & MASK_BUMP) {
224         turn(RadCCW, OVERTURN);
225     } else if (docking) {
```

```
226         dock();
227     } else if (!docking && FIELD) {
228         // Begin docking
229         // If we were already in red, we're coming from front.
230         comingFromFront = PRED;
231         turn(RadCCW, FIELD_TURN);
232         if (!comingFromFront) {
233             // Turn again to be perpendicular
234             move(FIELD_CLEARANCE);
235             turn(RadCW, FIELD_TURN);
236         }
237         // We are now docking
238         docking = 1;
239     } else if (!started) {
240         // Go straight until wall
241         drive(SPEED, RadStraight);
242     } else {
243         // PID
244 #ifdef LOG_OVER_USB
245         setSerialDestination(SERIAL_USB);
246 #endif
247         uint16_t wallSignal = getSensorUint16(SenWallSig1);
248         pidStep(wallSignal);
249 #ifdef LOG_OVER_USB
250         irobprintf("wallSignal: %u\ndeltaDrive: %d\n\n", utk / DRIVE_DIVISOR);
251         setSerialDestination(SERIAL_CREATE);
252 #endif
253         updateMotors();
254     }
255 }
```

lib4.h

```
#ifndef LIB2A_H
#define LIB2A_H

#include <stdint.h>

// Delay constant
#define IROB_PERIOD_MS  (32)

// PID settings
#define PID_SET_POINT   (32)
#define PID_KP          (256)
#define PID_KI          (1)
#define PID_KD          (64)
#define DRIVE_DIVISOR   (128)
#define PID_QSIZE       (128)

// Charging current threshold
#define CURRENTTHOLD    (-150)

// # Drive settings #
// Speed settings
#define SPEED           (100)
#define DOCKING_SPEED   (50)
#define JIMMY_SPEED     (30)
// Angle settings
#define OVERTURN        (10)
#define FIELD_TURN      (90)
#define FRONT_TURN      (60)
#define JIMMY_ANGLE     (10)
// Distance settings
#define FIELD_CLEARANCE (300)
#define IROB_RAD_TURN   (150)

//#define LOG_OVER_USB

void pidSetup(void);

void pidCleanup(void);

void pidStep(uint16_t vtk);

void updateMotors(void);

void doWhileTurning(void);

void move(int16_t distance);
void turn(int16_t radius, int16_t angle);

uint8_t noBump(void);
void jimmyBump(void);
uint8_t notCharging(void);
void jimmyTurn(int16_t radius);
void jimmy(void);

```

```
55 void dock(void);
56 void dockingDiagnostics(void);
57
58 //! Called by irobPeriodic
59 void iroblifePeriodic(void);
60
61 #endif
```

proj4.c

```c
#include "iroblife.h"
#include "sensing.h"

#include "lib4.h"

int main(void) {
    // Submit to iroblife
    setIrobInitImpl(&pidSetup);
    setIrobPeriodicImpl(&iroblifePeriodic);
    setIrobEndImpl(&pidCleanup);

    // Initialize the Create
    irobInit();

    // Infinite operation loop
    for(;;) {
        // Periodic execution
        irobPeriodic();

        // Delay for the loop; one second
        delayAndUpdateSensors(IROB_PERIOD_MS);
    }
}
```

utils/cmod.c

```c
#include "cmod.h"
#include "oi.h"
#include "timer.h"

void initializeCommandModule(void){
    // Disable interrupts. ("Clear interrupt bit")
    cli();

    // One-time setup operations.
    setupIOPins();
    setupTimer();
    setupSerialPort();

    // Enable interrupts. ("Set interrupt bit")
    sei();
}

void setupIOPins(void) {
    // Set I/O pins
    DDRB  = 0x10;
    PORTB = 0xCF;
    DDRC  = 0x00;
    PORTC = 0xFF;
    DDRD  = 0xE6;
    PORTD = 0x7D;
}

void setupSerialPort(void) {
    // Set the transmission speed to 57600 baud, which is what the Create expects,
    // unless we tell it otherwise.
    UBRR0 = 19;

    // Enable both transmit and receive.
    UCSR0B = (_BV(RXCIE0) | _BV(TXEN0) | _BV(RXEN0));
        // UCSR0B = 0x18;

    // Set 8-bit data.
    UCSR0C = (_BV(UCSZ00) | _BV(UCSZ01));
        // UCSR0C = 0x06;
}

void waitForEmptyTxBuffer(void) {
    while(!(UCSR0A & 0x20)) ;
}

void byteTx(uint8_t value) {
    // Transmit one byte to the robot.
    // Wait for the buffer to be empty.
    waitForEmptyTxBuffer();

    // Send the byte.
    UDR0 = value;
}

```

```
55  void uint16Tx(uint16_t value) {
56      // Transmit two bytes to the robot.
57      byteTx((uint8_t)((value >> 8) & 0x00FF));
58      byteTx((uint8_t)(value & 0x00FF));
59  }
60
61  uint8_t byteRx(void) {
62      // Receive one byte from the robot.
63      // Call setupSerialPort() first.
64      // Wait for a byte to arrive in the recieve buffer.
65      while(!(UCSR0A & 0x80)) ;
66
67      // Return that byte.
68      return UDR0;
69  }
70
71  void baud(uint8_t baud_code) {
72    // Switch the baud rate on both Create and module
73    if(baud_code <= 11)
74    {
75      byteTx(CmdBaud);
76      UCSR0A |= _BV(TXC0);
77      byteTx(baud_code);
78      // Wait until transmit is complete
79      while(!(UCSR0A & _BV(TXC0))) ;
80
81      cli();
82
83      // Switch the baud rate register
84      if(baud_code == Baud115200) {
85        UBRR0 = Ubrr115200;
86      } else if(baud_code == Baud57600) {
87        UBRR0 = Ubrr57600;
88      } else if(baud_code == Baud38400) {
89        UBRR0 = Ubrr38400;
90      } else if(baud_code == Baud28800) {
91        UBRR0 = Ubrr28800;
92      } else if(baud_code == Baud19200) {
93        UBRR0 = Ubrr19200;
94      } else if(baud_code == Baud14400) {
95        UBRR0 = Ubrr14400;
96      } else if(baud_code == Baud9600) {
97        UBRR0 = Ubrr9600;
98      } else if(baud_code == Baud4800) {
99        UBRR0 = Ubrr4800;
100     } else if(baud_code == Baud2400) {
101       UBRR0 = Ubrr2400;
102     } else if(baud_code == Baud1200) {
103       UBRR0 = Ubrr1200;
104     } else if(baud_code == Baud600) {
105       UBRR0 = Ubrr600;
106     } else if(baud_code == Baud300) {
107       UBRR0 = Ubrr300;
108     }
109     sei();
110
111     delayMs(100);
```

```
112     }
113 }
```

utils/cmod.h

```
1  #ifndef INCLUDE_CMOD_H
2  #define INCLUDE_CMOD_H
3
4  #include <avr/io.h>
5  #include <avr/interrupt.h>
6  #include <stdint.h>
7
8  // Setup the I/O pins.
9  void setupIOPins(void);
10
11 // Setup the serial port: Baud rate, transmit/recieve, packet size.
12 void setupSerialPort(void);
13
14 // Contains a collection of commands that allows me to "start" immediately
15 // after calling this command.
16 void initializeCommandModule(void);
17
18 // Wait for the transmit buffer to be empty.
19 void waitForEmptyTxBuffer(void);
20
21 // Send and receive data from the Command Module
22 void byteTx(uint8_t value);
23 void uint16Tx(uint16_t value);
24 uint8_t byteRx(void);
25
26 // Switch the baud rate on both Create and module
27 void baud(uint8_t baud_code);
28
29 #endif
```

utils/driving.c

```c
#include <stdint.h>
#include "driving.h"
#include "oi.h"
#include "cmod.h"
#include "timer.h"

// Weird constants because squeezing out precision
#define PIe5            314159
#define TENTH_RADIUS    13

// # BASIC COMMANDS #

void driveDirect(uint16_t left, uint16_t right) {
    // Send the direct drive command to the Create
    byteTx(CmdDriveWheels);
    uint16Tx(right);
    uint16Tx(left);
}

void drive(int16_t velocity, int16_t radius) {
    // Send the start driving command to the Create
    byteTx(CmdDrive);
    uint16Tx(velocity);
    uint16Tx(radius);
    /*byteTx((uint8_t)((velocity >> 8) & 0x00FF));
    byteTx((uint8_t)(velocity & 0x00FF));
    byteTx((uint8_t)((radius >> 8) & 0x00FF));
    byteTx((uint8_t)(radius & 0x00FF));*/
}

void driveStop(void) {
    drive(0, RadStraight);
}


// # OPCODE-BASED COMMANDS #

void driveDistanceOp(int16_t velocity, int16_t distance) {
    // Start driving
    drive(velocity, RadStraight);
    // Halt execution of new commands on the Create until reached distance
    byteTx(WaitForDistance);
    uint16Tx(distance);
    /*byteTx((uint8_t)((distance >> 8) & 0x00FF));
    byteTx((uint8_t)(distance & 0x00FF));*/
    // Stop the Create
    driveStop();
}

void driveAngleOp(int16_t velocity, int16_t radius, int16_t angle) {
    // Wait for angle opcode compatibility
    if (radius == RadCW) {
        angle = -angle;
    }
```

```
55      // Start driving
56      drive(velocity, radius);
57      // Halt execution of new commands on the Create until reached angle
58      byteTx(WaitForAngle);
59      uint16Tx(angle);
60      /*byteTx((uint8_t)((angle >> 8) & 0x00FF));
61      byteTx((uint8_t)(angle & 0x00FF));*/
62      // Stop the Create
63      driveStop();
64  }
65
66
67  // # TIMER-BASED COMMANDS #
68
69  void driveDistanceTFunc(int16_t velocity, int16_t distance, void (*func)(void),
70          uint16_t period_ms, uint16_t cutoff_ms) {
71      // Calculate the delay
72      uint32_t time_ms = (1000 * (uint32_t)distance) / (uint32_t)velocity;
73      // Start driving
74      drive(velocity, RadStraight);
75      // Wait delay
76      delayMsFunc(time_ms, func, period_ms, cutoff_ms);
77      // Stop the Create
78      driveStop();
79  }
80
81  void driveAngleTFunc(int16_t velocity, int16_t radius, int16_t angle,
82          void (*func)(void), uint16_t period_ms, uint16_t cutoff_ms) {
83      // Calculate the delay
84      uint32_t time_ms = (PIe5 * TENTH_RADIUS * (uint32_t)angle)
85          / (1800 * (uint32_t)velocity);
86      // Start driving
87      drive(velocity, radius);
88      // Wait delay
89      delayMsFunc(time_ms, func, period_ms, cutoff_ms);
90      // Stop the Create
91      driveStop();
92  }
93
94  // # PREDICATE-BASED COMMANDS #
95
96  void drivePredicateFunc(int16_t velocity, int16_t radius,
97          uint8_t (*pred)(void), void (*func)(void), uint16_t period_ms,
98          uint16_t cutoff_ms) {
99      // Start driving
100     drive(velocity, radius);
101     // Wait
102     delayPredicateFunc(pred, func, period_ms, cutoff_ms);
103     // Stop the Create
104     driveStop();
105 }
106
107
108 void driveDistancePFunc(int16_t velocity, int16_t distance,
109         uint8_t (*pred)(void), void (*func)(void), uint16_t period_ms,
110         uint16_t cutoff_ms) {
111     // Calculate the delay
```

```
112     uint32_t time_ms = (1000 * (uint32_t)distance) / (uint32_t)velocity;
113     // Start driving
114     drive(velocity, RadStraight);
115     // Wait delay
116     delayMsPredicateFunc(time_ms, pred, func, period_ms, cutoff_ms);
117     // Stop the Create
118     driveStop();
119 }
120
121 void driveAnglePFunc(int16_t velocity, int16_t radius, int16_t angle,
122         uint8_t (*pred)(void), void (*func)(void), uint16_t period_ms,
123         uint16_t cutoff_ms) {
124     // Calculate the delay
125     uint32_t time_ms = (PIe5 * TENTH_RADIUS * (uint32_t)angle)
126         / (1800 * (uint32_t)velocity);
127     // Start driving
128     drive(velocity, radius);
129     // Wait delay
130     delayMsPredicateFunc(time_ms, pred, func, period_ms, cutoff_ms);
131     // Stop the Create
132     driveStop();
133 }
```

utils/driving.h

```
1  #ifndef DRIVING_H
2  #define DRIVING_H
3
4  #include <stdint.h>
5
6  // # BASIC COMMANDS #
7
8  //! Directly drive the Create motors.
9  /*!
10  *  Returns immediately.
11  *
12  *  \param left     Speed of the left motor in mm/s.
13  *  \param right    Speed of the right motor in mm/s.
14  */
15 void driveDirect(uint16_t left, uint16_t right);
16
17 //! Drive at a certain speed in a certain direction.
18 /*!
19  *  Returns immediately.
20  *
21  *  Directions: straight, clockwise, counterclockwise.
22  *
23  *  \param velocity     The speed in mm/s.
24  *  \param radius       Either RadStraight, RadCW, or RadCCW (see oi.h).
25  */
26 void drive(int16_t velocity, int16_t radius);
27
28 //! Stop the robot.
29 void driveStop(void);
30
31
32 // # OPCODE-BASED COMMANDS #
33
34 //! Drive a certain distance at a certain speed.
35 /*!
36  *  Drive a certain distance using the Create wait for distance opcode.
37  *
38  *  \param velocity     The speed in mm/s.
39  *  \param distance     The distance to travel in mm.
40  */
41 void driveDistanceOp(int16_t velocity, int16_t distance);
42
43 //! Rotate a certain angle at a certain speed.
44 /*!
45  *  Drive a certain angle using the Create wait for angle opcode.
46  *
47  *  \param velocity     The speed in mm/s.
48  *  \param radius       Either RadCW or RadCCW (see oi.h).
49  *  \param angle        The angle to rotate in degrees.
50  */
51 void driveAngleOp(int16_t velocity, int16_t radius, int16_t angle);
52
53
54 // # TIMER-BASED COMMANDS #
```

```
55
56  //! Drive a certain distance at a certain speed.
57  /*!
58   *  Drive a certain distance using a timer.
59   *
60   *  \param velocity     The speed in mm/s.
61   *  \param distance     The distance to travel in mm.
62   *  \param func         The function to execute periodically.
63   *  \param period_ms    The interval to execute the function.
64   *  \param cutoff_ms    The number of milliseconds before the end to stop
65   *                      attempting to start the function.
66   */
67  void driveDistanceTFunc(int16_t velocity, int16_t distance, void (*func)(void),
68          uint16_t period_ms, uint16_t cutoff_ms);
69
70  //! Drive a certain angle at a certain speed.
71  /*!
72   *  Drive a certain angle using a timer.
73   *
74   *  \param velocity     The speed in mm/s.
75   *  \param radius       Either RadCW or RadCCW (see oi.h).
76   *  \param angle        The angle to rotate in degrees.
77   *  \param func         The function to execute periodically.
78   *  \param period_ms    The interval to execute the function.
79   *  \param cutoff_ms    The number of milliseconds before the end to stop
80   *                      attempting to start the function.
81   */
82  void driveAngleTFunc(int16_t velocity, int16_t radius, int16_t angle,
83          void (*func)(void), uint16_t period_ms, uint16_t cutoff_ms);
84
85  // # PREDICATE-BASED COMMANDS #
86
87  void drivePredicateFunc(int16_t velocity, int16_t radius,
88          uint8_t (*pred)(void), void (*func)(void), uint16_t period_ms,
89          uint16_t cutoff_ms);
90
91
92  void driveDistancePFunc(int16_t velocity, int16_t distance,
93          uint8_t (*pred)(void), void (*func)(void), uint16_t period_ms,
94          uint16_t cutoff_ms);
95
96  void driveAnglePFunc(int16_t velocity, int16_t radius, int16_t angle,
97          uint8_t (*pred)(void), void (*func)(void), uint16_t period_ms,
98          uint16_t cutoff_ms);
99
100 #endif
```

utils/fixedqueue.c

```c
#include "fixedqueue.h"
#include <stdlib.h>

FixedQueue newFixedQueue(size_t size) {
    FixedQueue q = malloc(sizeof(*q));
    if (q == 0) {
        return 0;
    }
    q->array = calloc(size, sizeof(*q->array));
    if (q->array == 0) {
        free(q);
        return 0;
    }
    q->size = size;
    q->head = 0;
    return q;
}

void freeFixedQueue(FixedQueue q) {
    free(q->array);
    free(q);
}

int16_t pushPop(FixedQueue q, int16_t value) {
    int16_t r = q->array[q->head];
    q->array[q->head++] = value;
    q->head %= q->size;
    return r;
}
```

utils/fixedqueue.h

```
1  #ifndef FIXEDQUEUE_H
2  #define FIXEDQUEUE_H
3
4  #include <stdint.h>
5  #include <stddef.h>
6
7  typedef struct FixedQueueStruct {
8      int16_t *array;
9      size_t size;
10     size_t head;
11 } * FixedQueue;
12
13 FixedQueue newFixedQueue(size_t size);
14
15 void freeFixedQueue(FixedQueue q);
16
17 int16_t pushPop(FixedQueue q, int16_t value);
18
19 #endif
```

utils/irchar.c

```c
#include "irchar.h"
#include "sensing.h"
#include "oi.h"

uint8_t _updateIR(uint8_t avg, uint8_t mask);

uint8_t redRunningAverage = 0;
uint8_t greenRunningAverage = 0;
uint8_t fieldRunningAverage = 0;

uint8_t region = IR_NONE;
uint8_t prevRegion = IR_NONE;

uint8_t irAny(void) {
    uint8_t ir = getSensorUint8(SenIRChar);
    if (ir == IR_NONE) return 0;
    return ((ir & IR_RESERVED) == IR_RESERVED);
}

uint8_t irAll(void) {
    return (getSensorUint8(SenIRChar) == IR_ALL);
}

uint8_t irCheck(uint8_t mask) {
    if (!irAny())   return 0;
    return (getSensorUint8(SenIRChar) & mask);
}

uint8_t irRed(void) {
    return irCheck(IR_MASK_RED_BUOY);
}

uint8_t irGreen(void) {
    return irCheck(IR_MASK_GREEN_BUOY);
}

uint8_t irForceField(void) {
    return irCheck(IR_MASK_FORCE_FIELD);
}

uint8_t _updateIR(uint8_t avg, uint8_t mask) {
    // Calculate running average
    if (irCheck(mask)) {
        avg = (((uint16_t)avg) * (SMOOTHING_INTENSITY - 1) + 0xFF)\
              / SMOOTHING_INTENSITY;
    } else {
        avg /= SMOOTHING_INTENSITY;
    }
    return avg;
}

void updateIR(void) {
    // Calculate running averages
    redRunningAverage = _updateIR(redRunningAverage, IR_MASK_RED_BUOY);
```

```
55      greenRunningAverage = _updateIR(greenRunningAverage, IR_MASK_GREEN_BUOY);
56      fieldRunningAverage = _updateIR(fieldRunningAverage, IR_MASK_FORCE_FIELD);
57      // Determine which (smoothed) region we're in
58      prevRegion = region;
59      uint8_t red = (smoothRed() > 0x60);
60      uint8_t green = (smoothGreen() > 0x60);
61      uint8_t field = (smoothForceField() > 0x60);
62      if (red && green && field) {
63          region = IR_ALL;
64      } else if (red && green) {
65          region = IR_RED_AND_GREEN;
66      } else if (red && field) {
67          region = IR_RED_AND_FIELD;
68      } else if (green && field) {
69          region = IR_GREEN_AND_FIELD;
70      } else if (red) {
71          region = IR_RED_BUOY;
72      } else if (green) {
73          region = IR_GREEN_BUOY;
74      } else if (field) {
75          region = IR_FORCE_FIELD;
76      } else {
77          region = IR_NOWHERE;
78      }
79  }
80
81  uint8_t smoothRed(void) {
82      return redRunningAverage;
83  }
84
85  uint8_t smoothGreen(void) {
86      return greenRunningAverage;
87  }
88
89  uint8_t smoothForceField(void) {
90      return fieldRunningAverage;
91  }
92
93  uint8_t irRegion(void) {
94      return region;
95  }
96
97  uint8_t irPrevRegion(void) {
98      return prevRegion;
99  }
```

utils/irchar.h

```
#ifndef IRCHAR_H
#define IRCHAR_H

#include <stdint.h>

// 1-0x101
#define SMOOTHING_INTENSITY     (16)

#define IR_RESERVED         (240)
#define IR_NOWHERE          (IR_RESERVED)
#define IR_RED_BUOY         (248)
#define IR_GREEN_BUOY       (244)
#define IR_FORCE_FIELD      (242)
#define IR_RED_AND_GREEN    (252)
#define IR_RED_AND_FIELD    (250)
#define IR_GREEN_AND_FIELD  (246)
#define IR_ALL              (254)
#define IR_NONE             (255)

#define IR_MASK_RED_BUOY    (IR_RED_BUOY ^ IR_RESERVED)
#define IR_MASK_GREEN_BUOY  (IR_GREEN_BUOY ^ IR_RESERVED)
#define IR_MASK_FORCE_FIELD (IR_FORCE_FIELD ^ IR_RESERVED)

uint8_t irAny(void);
uint8_t irAll(void);
uint8_t irCheck(uint8_t mask);
uint8_t irRed(void);
uint8_t irGreen(void);
uint8_t irForceField(void);

void updateIR(void);
uint8_t smoothRed(void);
uint8_t smoothGreen(void);
uint8_t smoothForceField(void);

uint8_t irRegion(void);
uint8_t irPrevRegion(void);

#endif
```

utils/irobled.c

```c
#include <stdint.h>
#include "irobled.h"
#include "cmod.h"
#include "oi.h"

// The current state of the leds.
struct {
    uint8_t bits;
    uint8_t color;
    uint8_t intensity;
} iroblibState;

void irobledCmd(uint8_t bits, uint8_t color, uint8_t intensity) {
    // Modify the state
    iroblibState.bits = bits;
    iroblibState.color = color;
    iroblibState.intensity = intensity;
    // Update
    irobledUpdate();
}

void irobledUpdate(void) {
    // Send the led command using the current state
    byteTx(CmdLeds);
    byteTx(iroblibState.bits);
    byteTx(iroblibState.color);
    byteTx(iroblibState.intensity);
}

void irobledInit(void) {
    irobledCmd(NEITHER_ROBOT_LED, POWER_LED_ORANGE, 0xFF);
}

void powerLedSet(uint8_t color, uint8_t intensity) {
    irobledCmd(iroblibState.bits, color, intensity);
}

void robotLedSetBits(uint8_t bits) {
    iroblibState.bits = bits;
    irobledUpdate();
}

void robotLedOn(uint8_t led) {
    iroblibState.bits |= led;
    irobledUpdate();
}

void robotLedOff(uint8_t led) {
    iroblibState.bits &= ~led;
    irobledUpdate();
}

void robotLedToggle(uint8_t led) {
    iroblibState.bits ^= led;
```

```
55      irobledUpdate();
56  }
57
58  void cmdLED1Set(uint8_t bit) {
59      if (bit) {
60          LED1On;
61      } else {
62          LED1Off;
63      }
64  }
65
66  void cmdLED2Set(uint8_t bit) {
67      if (bit) {
68          LED2On;
69      } else {
70          LED2Off;
71      }
72  }
```

utils/irobled.h

```
1  #ifndef IROBLED_H
2  #define IROBLED_H
3
4  #include <stdint.h>
5
6  // Colors for the power led.
7  #define POWER_LED_GREEN   (0x00)
8  #define POWER_LED_ORANGE  (0x40)
9  #define POWER_LED_RED     (0xFF)
10
11 // Bits for the other leds.
12 #define NEITHER_ROBOT_LED (0x00)
13 #define PLAY_ROBOT_LED    (0x02)
14 #define ADVANCE_ROBOT_LED (0x08)
15 #define BOTH_ROBOT_LED    (0x0A)
16
17 //! Send an led command to the Create.
18 void irobledCmd(uint8_t bits, uint8_t color, uint8_t intensity);
19 //! Update the leds. Probably won't have to use.
20 void irobledUpdate(void);
21 //! Initialize the leds to red for power and off for the others.
22 void irobledInit(void);
23
24 //! Set the color and intensity of the power led.
25 void powerLedSet(uint8_t color, uint8_t intensity);
26
27 // Functions for modifying one or both of the other leds.
28 void robotLedSetBits(uint8_t bits);
29 void robotLedOn(uint8_t led);
30 void robotLedOff(uint8_t led);
31 void robotLedToggle(uint8_t led);
32
33 void cmdLED1Set(uint8_t bit);
34 void cmdLED2Set(uint8_t bit);
35
36 #endif
```

utils/iroblib.c

```c
#include "iroblib.h"
#include "oi.h"
#include "cmod.h"
#include "timer.h"

// Define songs to be played later
void defineSongs(void) {
  // Reset song
  byteTx(CmdSong);
  byteTx(RESET_SONG);
  byteTx(4);
  byteTx(60);
  byteTx(6);
  byteTx(72);
  byteTx(6);
  byteTx(84);
  byteTx(6);
  byteTx(96);
  byteTx(6);

  // Start song
  byteTx(CmdSong);
  byteTx(START_SONG);
  byteTx(6);
  byteTx(69);
  byteTx(18);
  byteTx(72);
  byteTx(12);
  byteTx(74);
  byteTx(12);
  byteTx(72);
  byteTx(12);
  byteTx(69);
  byteTx(12);
  byteTx(77);
  byteTx(24);
}

// Ensure that the robot is On.
void powerOnRobot(void) {
  // If Create's power is off, turn it on
  if(!RobotIsOn) {
    while(!RobotIsOn) {
      RobotPwrToggleLow;
      delayMs(500);  // Delay in this state
      RobotPwrToggleHigh;  // Low to high transition to toggle power
      delayMs(100);  // Delay in this state
      RobotPwrToggleLow;
    }
    delayMs(3500);  // Delay for startup
  }

  // Flush the buffer
  while( (UCSR0A & 0x80) && UDR0);
```

```
55  }
56
57  // Ensure that the robot is OFF.
58  void powerOffRobot(void) {
59    // If Create's power is on, turn it off
60    if(RobotIsOn) {
61      while(RobotIsOn) {
62        RobotPwrToggleLow;
63        delayMs(500);  // Delay in this state
64        RobotPwrToggleHigh;  // Low to high transition to toggle power
65        delayMs(100);  // Delay in this state
66        RobotPwrToggleLow;
67      }
68    }
69  }
```

utils/iroblib.h

```
1  #ifndef INCLUDE_IROBLIB_H
2  #define INCLUDE_IROBLIB_H
3
4  #include <avr/io.h>
5  #include <avr/interrupt.h>
6
7  // Constants
8  #define RESET_SONG 0
9  #define START_SONG 1
10
11 void defineSongs(void);
12   // Songs
13   // Indicator that the robot is Powered on and has reset.
14
15 void powerOnRobot(void);
16 void powerOffRobot(void);
17   // Power the create On/Off.
18 #endif
```

utils/iroblife.c

```c
#include <stdlib.h>
#include <stdint.h>
#include "iroblife.h"

#include "timer.h"
#include "cmod.h"
#include "iroblib.h"
#include "oi.h"

#include "sensing.h"
#include "irobled.h"
#include "driving.h"
#include "irobserial.h"

void irobImplNull(void) {
}

void (*irobInitImpl)(void) = &irobImplNull;
void (*irobPeriodicImpl)(void) = &irobImplNull;
void (*irobEndImpl)(void) = &irobImplNull;

void setIrobInitImpl(void (*func)(void)) {
    irobInitImpl = func;
}

void setIrobPeriodicImpl(void (*func)(void)) {
    irobPeriodicImpl = func;
}

void setIrobEndImpl(void (*func)(void)) {
    irobEndImpl = func;
}

void irobInit(void) {
    // Set up Create and module
    initializeCommandModule();
    // Set Create as default serial destination
    setSerialDestination(SERIAL_CREATE);

    // Is the Robot on
    powerOnRobot();
    // Start the create
    byteTx(CmdStart);
    // Set the baud rate for the Create and Command Module
    baud(Baud57600);
    // Define some songs so that we know the robot is on.
    defineSongs();
    // Deprecated form of safe mode. I use it because it will
    // turn of all LEDs, so it's essentially a reset.
    byteTx(CmdControl);
    // We are operating in FULL mode.
    byteTx(CmdFull);

    // Make sure the robot stops.
```

```
55      // As a precaution for the robot and your grade.
56      driveStop();
57
58      // Play the reset song and wait while it plays.
59      byteTx(CmdPlay);
60      byteTx(RESET_SONG);
61      delayMs(750);
62
63      // Turn the power button on to orange.
64      irobledInit();
65
66      // Call the user's init function
67      irobInitImpl();
68  }
69
70  void irobPeriodic(void) {
71      // Call the user's periodic function
72      irobPeriodicImpl();
73      // Exit if the black button on the command module is pressed.
74      if(UserButtonPressed) {
75          irobEnd();
76      }
77  }
78
79  void irobEnd(void) {
80      // Call the user's end function
81      irobEndImpl();
82      // Stop the Create
83      driveStop();
84      // Power off the Create
85      powerOffRobot();
86      // Exit the program
87      exit(1);
88  }
```

utils/iroblife.h

```
1  #ifndef IROBLIFE_H
2  #define IROBLIFE_H
3
4  /*
5   *  The irobPeriodic function in this library calls a function given to
6   *  setIrobPeriodicImpl. The default value does nothing, but you can give
7   *  it another function as a hook for periodically executed code.
8   */
9
10 //! Default periodic function. Does nothing.
11 void irobImplNull(void);
12 //! Set the function that irobInit calls.
13 void setIrobInitImpl(void (*func)(void));
14 //! Set the function that irobPeriodic calls.
15 void setIrobPeriodicImpl(void (*func)(void));
16 //! Set the function that irobEnd calls.
17 void setIrobEndImpl(void (*func)(void));
18
19 //! Initialize the Create. Call this at the beginning of your main.
20 void irobInit(void);
21 //! Periodic operations. Call this in your main loop.
22 //! Calls the function last given to setIrobPeriodicImpl.
23 void irobPeriodic(void);
24 //! Stops and shuts down the Create, then exits. Call this to end the program.
25 void irobEnd(void);
26
27 #endif
```

utils/irobserial.c

```c
#include <stdint.h>
#include <stdarg.h>
#include <stdio.h>
#include "irobserial.h"
#include "cmod.h"
#include "oi.h"
#include "timer.h"

uint8_t serialDestination = SERIAL_SWITCHING;

void setSerialDestination(uint8_t dest) {
    serialDestination = SERIAL_SWITCHING;
    // Which serial port should byteTx and byteRx talk to?
    // Ensure any pending bytes have been sent. Without this, the last byte
    // sent before calling this might seem to disappear.
    delayMs(10);
    // Configure the port.
    if (dest == SERIAL_CREATE) {
        PORTB &= ~0x10 ;
    } else {
        PORTB |= 0x10 ;
    }
    // Wait a bit to let things get back to normal. According to the docs, this
    // should be at least 10 times the amount of time needed to send one byte.
    // This is less than 1 millisecond. We are using a much longer delay to be
    // super extra sure.
    delayMs(20);
    serialDestination = dest;
}

uint8_t getSerialDestination(void) {
   return serialDestination;
}

void irobprint(char* str) {
    char c;
    // Null-terminated string
    while ((c = *(str++)) != '\0') {
        // Print each byte
        byteTx(c);
    }
}

char printfBuffer[PRINTF_BUFFER_SIZE];

void irobprintf(const char* format, ...) {
    char* fp = &printfBuffer[0];
    va_list ap;
    va_start(ap, format);
    // Format the string
    vsnprintf(fp, PRINTF_BUFFER_SIZE, format, ap);
    va_end(ap);
    // Print the string
    irobprint(fp);
```

```
55 }
56
57 void irobnprintf(uint16_t size, const char* format, ...) {
58     // Create a buffer
59     char formatted[size];
60     char* fp = &formatted[0];
61     va_list ap;
62     va_start(ap, format);
63     // Format the string
64     vsnprintf(fp, size, format, ap);
65     va_end(ap);
66     // Print the string
67     irobprint(fp);
68 }
```

utils/irobserial.h

```
1  #ifndef IROBSERIAL_H
2  #define IROBSERIAL_H
3
4  #include <stdint.h>
5  #include <stdarg.h>
6
7  #define SERIAL_CREATE       (1)
8  #define SERIAL_USB          (2)
9  #define SERIAL_SWITCHING    (0xFF)
10
11 #define PRINTF_BUFFER_SIZE  (0xFF)
12
13 //! Set the serial output (CREATE or USB)
14 //! Takes some time.
15 void setSerialDestination(uint8_t dest);
16
17 //! Get the serial output (CREATE or USB)
18 uint8_t getSerialDestination(void);
19
20 //! Print a string
21 void irobprint(char* str);
22
23 //! Print a formatted string (Max length: 255 bytes)
24 void irobprintf(const char* format, ...);
25
26 //! Print a formatted string (for strings longer than 255 bytes)
27 void irobnprintf(uint16_t size, const char* format, ...);
28
29 #endif
```

utils/oi.h

```
1  /* oi.h
2   *
3   * Definitions for the Open Interface
4   */
5
6  #ifndef OI_H
7  #define OI_H
8
9  // Command values
10 #define CmdStart        128
11 #define CmdBaud         129
12 #define CmdControl      130
13 #define CmdSafe         131
14 #define CmdFull         132
15 #define CmdSpot         134
16 #define CmdClean        135
17 #define CmdDemo         136
18 #define CmdDrive        137
19 #define CmdMotors       138
20 #define CmdLeds         139
21 #define CmdSong         140
22 #define CmdPlay         141
23 #define CmdSensors      142
24 #define CmdDock         143
25 #define CmdPWMMotors    144
26 #define CmdDriveWheels  145
27 #define CmdOutputs      147
28 #define CmdSensorList   149
29 #define CmdIRChar       151
30 #define WaitForDistance 156
31 #define WaitForAngle    157
32
33
34 // Sensor byte indices - offsets in packets 0, 5 and 6
35 #define SenBumpDrop     0
36 #define SenWall         1
37 #define SenCliffL       2
38 #define SenCliffFL      3
39 #define SenCliffFR      4
40 #define SenCliffR       5
41 #define SenVWall        6
42 #define SenOverC        7
43 #define SenIRChar       10
44 #define SenButton       11
45 #define SenDist1        12
46 #define SenDist0        13
47 #define SenAng1         14
48 #define SenAng0         15
49 #define SenChargeState  16
50 #define SenVolt1        17
51 #define SenVolt0        18
52 #define SenCurr1        19
53 #define SenCurr0        20
54 #define SenTemp         21
```

```
55  #define SenCharge1      22
56  #define SenCharge0      23
57  #define SenCap1         24
58  #define SenCap0         25
59  #define SenWallSig1     26
60  #define SenWallSig0     27
61  #define SenCliffLSig1   28
62  #define SenCliffLSig0   29
63  #define SenCliffFLSig1  30
64  #define SenCliffFLSig0  31
65  #define SenCliffFRSig1  32
66  #define SenCliffFRSig0  33
67  #define SenCliffRSig1   34
68  #define SenCliffRSig0   35
69  #define SenInputs       36
70  #define SenAInput1      37
71  #define SenAInput0      38
72  #define SenChAvailable  39
73  #define SenOIMode       40
74  #define SenOISong       41
75  #define SenOISongPlay   42
76  #define SenStreamPckts  43
77  #define SenVel1         44
78  #define SenVel0         45
79  #define SenRad1         46
80  #define SenRad0         47
81  #define SenVelR1        48
82  #define SenVelR0        49
83  #define SenVelL1        50
84  #define SenVelL0        51
85
86
87  // Sensor packet sizes
88  #define Sen0Size        26
89  #define Sen1Size        10
90  #define Sen2Size        6
91  #define Sen3Size        10
92  #define Sen4Size        14
93  #define Sen5Size        12
94  #define Sen6Size        52
95
96  // Sensor bit masks
97  #define WheelDropFront  0x10
98  #define WheelDropLeft   0x08
99  #define WheelDropRight  0x04
100 #define BumpLeft        0x02
101 #define BumpRight       0x01
102 #define BumpBoth        0x03
103 #define BumpEither      0x03
104 #define WheelDropAll    0x1C
105 #define ButtonAdvance   0x04
106 #define ButtonPlay      0x01
107
108
109 // LED Bit Masks
110 #define LEDAdvance       0x08
111 #define LEDPlay         0x02
```

```
112 #define LEDsBoth        0x0A
113
114 // OI Modes
115 #define OIPassive       1
116 #define OISafe          2
117 #define OIFull          3
118
119
120 // Baud codes
121 #define Baud300         0
122 #define Baud600         1
123 #define Baud1200        2
124 #define Baud2400        3
125 #define Baud4800        4
126 #define Baud9600        5
127 #define Baud14400       6
128 #define Baud19200       7
129 #define Baud28800       8
130 #define Baud38400       9
131 #define Baud57600       10
132 #define Baud115200      11
133
134
135 // Drive radius special cases
136 #define RadStraight     32768
137 #define RadCCW          1
138 #define RadCW           -1
139
140
141
142 // Baud UBRRx values
143 #define Ubrr300         3839
144 #define Ubrr600         1919
145 #define Ubrr1200        959
146 #define Ubrr2400        479
147 #define Ubrr4800        239
148 #define Ubrr9600        119
149 #define Ubrr14400       79
150 #define Ubrr19200       59
151 #define Ubrr28800       39
152 #define Ubrr38400       29
153 #define Ubrr57600       19
154 #define Ubrr115200      9
155
156
157 // Command Module button and LEDs
158 #define UserButton        0x10
159 #define UserButtonPressed (!(PIND & UserButton))
160
161 #define LED1              0x20
162 #define LED1Off           (PORTD |= LED1)
163 #define LED1On            (PORTD &= ~LED1)
164 #define LED1Toggle        (PORTD ^= LED1)
165
166 #define LED2              0x40
167 #define LED2Off           (PORTD |= LED2)
168 #define LED2On            (PORTD &= ~LED2)
```

```
169 #define LED2Toggle        (PORTD ^= LED2)
170
171 #define LEDBoth           0x60
172 #define LEDBothOff        (PORTD |= LEDBoth)
173 #define LEDBothOn         (PORTD &= ~LEDBoth)
174 #define LEDBothToggle     (PORTD ^= LEDBoth)
175
176
177 // Create Port
178 #define RobotPwrToggle      0x80
179 #define RobotPwrToggleHigh (PORTD |= 0x80)
180 #define RobotPwrToggleLow  (PORTD &= ~0x80)
181
182 #define RobotPowerSense    0x20
183 #define RobotIsOn          (PINB & RobotPowerSense)
184 #define RobotIsOff         !(PINB & RobotPowerSense)
185
186 // Command Module ePorts
187 #define LD2Over         0x04
188 #define LD0Over         0x02
189 #define LD1Over         0x01
190
191 #endif
```

utils/sensing.c

```c
#include <stdint.h>
#include "sensing.h"
#include "cmod.h"
#include "timer.h"
#include "oi.h"
#include "irobserial.h"

volatile uint8_t usartActive = 0;
volatile uint8_t sensorIndex = 0;
volatile uint8_t sensorBuffer[Sen6Size];
volatile uint8_t sensors[Sen6Size];

void requestPacket(uint8_t packetId) {
    byteTx(CmdSensors);
    byteTx(packetId);
}

uint8_t read1ByteSensorPacket(uint8_t packetId) {
    // Send the packet ID
    requestPacket(packetId);
    // Read the packet byte
    return byteRx();
}

ISR(USART_RX_vect) {
    // Cache the retrieved byte
    uint8_t tmpUDR0;
    tmpUDR0 = UDR0;
    // Don't do anything if we're not looking
    if (usartActive) {
        if (getSerialDestination() == SERIAL_CREATE) {
            // New sensor data from the create
            sensorBuffer[sensorIndex++] = tmpUDR0;
        } else {
            // Probably input from the computer, loop old values around
            sensorBuffer[sensorIndex] = sensors[sensorIndex];
            sensorIndex++;
        }
        if (sensorIndex >= Sen6Size) {
            // Reached end of sensor packet
            usartActive = 0;
        }
    }
}

void updateSensors(void) {
    // Don't do anything if sensors are still coming in
    if (!usartActive) {
        uint8_t i;
        for (i = 0; i < Sen6Size; i++) {
            // Copy in the sensor buffer so the most recent data is available
            sensors[i] = sensorBuffer[i];
        }
        // Bookkeeping
```

```
55          sensorIndex = 0;
56          usartActive = 1;
57          // Request all sensor data
58          requestPacket(PACKET_ALL);
59      }
60  }
61
62  void waitForSensors(void) {
63      // Sensors data are coming in if usartActive is true
64      while(usartActive);
65  }
66
67  void delayAndUpdateSensors(uint32_t time_ms) {
68      // Update sensors while waiting
69      delayMsFunc(time_ms, &updateSensors, 1, UPDATE_SENSOR_DELAY_CUTOFF);
70  }
71
72  uint8_t getSensorUint8(uint8_t index) {
73      // Already in the right format
74      return sensors[index];
75  }
76
77  int8_t getSensorInt8(uint8_t index) {
78      uint8_t x = getSensorUint8(index);
79      // Convert to signed; not implementation-dependent, and optimizes away
80      return x < (1 << 7) ? x : x - (1 << 8);
81  }
82
83  uint16_t getSensorUint16(uint8_t index1) {
84      // Combine msB and lsB
85      return (sensors[index1] << 8) | sensors[index1 + 1];
86  }
87
88  int16_t getSensorInt16(uint8_t index1) {
89      uint16_t x = getSensorUint16(index1);
90      // Convert to signed; more opaque hex values b/c avr complains for 1 << 16
91      return x < 0x8000 ? x : x - 0x10000;
92  }
```

utils/sensing.h

```
#ifndef SENSING_H
#define SENSING_H

#include <stdint.h>

#define UPDATE_SENSOR_DELAY_PERIOD      (1)
#define UPDATE_SENSOR_DELAY_CUTOFF      (10)


#define PACKET_BUMPS_AND_WHEEL_DROPS    (7)
#define MASK_WHEEL_DROP_CASTER          (1 << 4)
#define MASK_WHEEL_DROP_LEFT            (1 << 3)
#define MASK_WHEEL_DROP_RIGHT           (1 << 2)
#define MASK_WHEEL_DROP                 (0x1C)
#define MASK_BUMP_LEFT                  (1 << 1)
#define MASK_BUMP_RIGHT                 (1 << 0)
#define MASK_BUMP                       (0x03)

#define PACKET_BUTTONS                  (18)
#define MASK_BTN_ADVANCE                (1 << 2)
#define MASK_BTN_PLAY                   (1 << 0)

#define IR_LEFT                         (129)
#define IR_FORWARD                      (130)
#define IR_RIGHT                        (131)

#define PACKET_ALL                      (6)

//! Request a sensor packet. \see read1ByteSensorPacket(uint8_t)
/*!
 *  \deprecated {
 *      This uses the old, non-USART-based way of retrieving sensor data.
 *  }
 */
void requestPacket(uint8_t packetId);

//! Read in a 1-byte sensor packet.
/*!
 *  \deprecated {
 *      This uses the old, non-USART-based way of retrieving sensor data.
 *  }
 *
 *  What is a sensor packet? A byte (or bytes) containing data from a set of
 *  sensors, often shifted and ORed together. See the Create Open Interface
 *  documentation for more.
 *
 *  Currently Available Sensor Packets  (v = read1ByteSensorPacket(packetId)):
 *      Bumps and Wheel Drops   (packetId = PACKET_BUMPS_AND_WHEEL_DROPS):
 *          Caster Drop         (v & MASK_WHEEL_DROP_CASTER)
 *          Left Wheel Drop     (v & MASK_WHEEL_DROP_LEFT)
 *          Right Wheel Drop    (v & MASK_WHEEL_DROP_RIGHT)
 *          Any Wheel Drop      (v & MASK_WHEEL_DROP)
 *          Left Bumper         (v & MASK_BUMP_LEFT)
 *          Right Bumper        (v & MASK_BUMP_RIGHT)
```

```
55   *          Either Bumper       (v & MASK_BUMPER)
56   *      Create Buttons          (packetId = PACKET_BUTTONS):
57   *          Advance Button      (v & MASK_BTN_ADVANCE)
58   *          Play Button         (v & MASK_BTN_PLAY)
59   *
60   *  \param packetId     The ID of the packet to retrieve, as defined by the
61   *                      Create Open Interface.
62   */
63  uint8_t read1ByteSensorPacket(uint8_t packetId);
64
65  //! Request all packets (will be retrieved by USART)
66  void updateSensors(void);
67
68  //! Wait for all packets to be recieved by USART
69  void waitForSensors(void);
70
71  //! delayMs that updates sensors
72  void delayAndUpdateSensors(uint32_t time_ms);
73
74  //! Get an unsigned 1-byte sensor value
75  uint8_t getSensorUint8(uint8_t index);
76
77  //! Get a signed 1-byte sensor value
78  int8_t getSensorInt8(uint8_t index);
79
80  //! Get an unsigned 2-byte sensor value, indexed by the more significant
81  //! (lower index) byte
82  uint16_t getSensorUint16(uint8_t index1);
83
84  //! Get a signed 2-byte sensor value, indexed by the more significant
85  //! (lower index) byte
86  int16_t getSensorInt16(uint8_t index1);
87
88  #endif
```

utils/timer.c

```c
#include <stdint.h>
#include "timer.h"    // Declaration made available here


// Timer variables defined here
volatile uint32_t delayTimerCount = 0;   // Definition checked against declaration
volatile uint8_t  delayTimerRunning = 0; // Definition checked against declaration


// Chris -- moved to sensing.c
/*ISR(USART_RX_vect) {  //SIGNAL(SIG_USART_RECV)
    // Serial receive interrupt to store sensor values

    // CSCE 274 students, I have only ever used this method
    // when retrieving/storing a large amount of sensor data.
    // You DO NOT need it for this assignment. If i feel it
    // becomes relevant, I will show you how/when to use it.
}*/

//SIGNAL(SIG_OUTPUT_COMPARE1A)
ISR(TIMER1_COMPA_vect) {
    // Interrupt handler called every 1ms.
    // Decrement the counter variable, to allow delayMs to keep time.
    if(delayTimerCount != 0) {
        delayTimerCount--;
    } else {
        delayTimerRunning = 0;
    }
}

void setupTimer(void) {
    // Set up the timer 1 interupt to be called every 1ms.
    // It's probably best to treat this as a black box.
    // Basic idea: Except for the 71, these are special codes, for which details
    // appear in the ATMega168 data sheet. The 71 is a computed value, based on
    // the processor speed and the amount of "scaling" of the timer, that gives
    // us the 1ms time interval.
    TCCR1A = 0x00;
    // TCCR1B = 0x0C;
    TCCR1B = (_BV(WGM12) | _BV(CS12));
    OCR1A = 71;
    // TIMSK1 = 0x02;
    TIMSK1 = _BV(OCIE1A);
}

// Delay for the specified time in ms without updating sensor values
void delayMs(uint32_t time_ms) {
    delayTimerRunning = 1;
    delayTimerCount = time_ms;
    while(delayTimerRunning) ;
}

void delayMsFunc(uint32_t time_ms, void (*func)(void), uint16_t period_ms,
        uint16_t cutoff_ms) {
```

```
55      // Initialize the conditions for the delay loop
56      uint32_t lastExec = time_ms;
57      uint32_t nextExec = lastExec - period_ms;
58      // Start the timer
59      delayTimerRunning = 1;
60      delayTimerCount = time_ms;
61      // Wait until the timer runs out (delayTimerCount decrements every ms)
62      while(delayTimerRunning) {
63          // If it's before the cutoff and time for the next execution
64          if (delayTimerCount > cutoff_ms && delayTimerCount <= nextExec) {
65              // Execute the function
66              lastExec = delayTimerCount;
67              nextExec = lastExec - period_ms;
68              func();
69          }
70      }
71  }
72
73  void delayPredicateFunc(uint8_t (*pred)(void), void (*func)(void),
74          uint16_t period_ms, uint16_t cutoff_ms) {
75      // Initialize the conditions for the delay loop
76      uint32_t lastExec = 0xFFFFFFFF;
77      uint32_t nextExec = lastExec - period_ms;
78      // Start the timer
79      delayTimerRunning = 1;
80      delayTimerCount = 0xFFFFFFFF;
81      // Wait until the timer runs out (delayTimerCount decrements every ms)
82      while(pred()) {
83          // If it's before the cutoff and time for the next execution
84          if (delayTimerCount > cutoff_ms && delayTimerCount <= nextExec) {
85              // Execute the function
86              lastExec = delayTimerCount;
87              nextExec = lastExec - period_ms;
88              func();
89          }
90      }
91      delayMs(1);
92  }
93
94  void delayMsPredicateFunc(uint32_t time_ms, uint8_t (*pred)(void),
95          void (*func)(void), uint16_t period_ms, uint16_t cutoff_ms) {
96      // Initialize the conditions for the delay loop
97      uint32_t lastExec = time_ms;
98      uint32_t nextExec = lastExec - period_ms;
99      // Start the timer
100     delayTimerRunning = 1;
101     delayTimerCount = time_ms;
102     // Wait until the timer runs out (delayTimerCount decrements every ms)
103     while(delayTimerRunning && pred()) {
104         // If it's before the cutoff and time for the next execution
105         if (delayTimerCount > cutoff_ms && delayTimerCount <= nextExec) {
106             // Execute the function
107             lastExec = delayTimerCount;
108             nextExec = lastExec - period_ms;
109             func();
110         }
111     }
```

```
112 }
```

utils/timer.h

```
1  #ifndef INCLUDE_TIMER_H
2  #define INCLUDE_TIMER_H
3
4  #include <avr/io.h>
5  #include <avr/interrupt.h>
6
7  // Interrupts.
8  ISR(TIMER1_COMPA_vect);
9
10 // Timer functions
11 void setupTimer(void);
12 void delayMs(uint32_t time_ms);
13
14 // Declaration of timer variables
15 extern volatile uint32_t delayTimerCount;
16 extern volatile uint8_t  delayTimerRunning;
17
18 //! Wait milliseconds, execute a function periodically.
19 /*!
20  *  Executes a function at an interval until a cutoff has passed, returning
21  *  after a total number of milliseconds have passed.
22  *
23  *  \param time_ms      The total number of seconds to wait.
24  *  \param func         The function to execute periodically.
25  *  \param period_ms    The interval to execute the function.
26  *  \param cutoff_ms    The number of milliseconds before the end to stop
27  *                      attempting to start the function.
28  */
29 void delayMsFunc(uint32_t time_ms, void (*func)(void), uint16_t period_ms,
30         uint16_t cutoff_ms);
31
32 void delayPredicateFunc(uint8_t (*pred)(void), void (*func)(void),
33         uint16_t period_ms, uint16_t cutoff_ms);
34
35 void delayMsPredicateFunc(uint32_t time_ms, uint8_t (*pred)(void),
36         void (*func)(void), uint16_t period_ms, uint16_t cutoff_ms);
37
38 #endif
```