Contents

lib2a.c

```c
#include "lib2a.h"
#include "irobserial.h"
#include "sensing.h"
#include "oi.h"
#include "cmod.h"

// Called by irobPeriodic
void iroblifePeriodic(void) {
    // Switch output to USB
    setSerialDestination(SERIAL_USB);
    // Output sensor values
    irobprintf("Charging State: %d\n", getSensorUint8(SenChargeState));
    irobprintf("Voltage: %d\n", getSensorUint16(SenVolt1));
    irobprintf("Current: %d\n", getSensorInt16(SenCurr1));
    irobprintf("Battery Temperature: %d\n", getSensorInt8(SenTemp));
    irobprintf("Battery Charge: %d\n", getSensorUint16(SenCharge1));
    irobprintf("Battery Capacity: %d\n", getSensorUint16(SenCap1));
    irobprintf("Wall Signal: %d\n", getSensorUint16(SenWallSig1));
    irobprintf("Cliff Left Signal: %d\n", getSensorUint16(SenCliffLSig1));
    irobprintf("Cliff Front Left Signal: %d\n", getSensorUint16(SenCliffFLSig1));
    irobprintf("Cliff Front Right Signal: %d\n", getSensorUint16(SenCliffFRSig1));
    irobprintf("Cliff Right Signal: %d\n", getSensorUint16(SenCliffRSig1));
    // Spacing
    byteTx('\n');
    // Switch output back to Create for updating sensor values
    setSerialDestination(SERIAL_CREATE);
}
```

lib2a.h

```c
#ifndef LIB2A_H
#define LIB2A_H

//! Called by irobPeriodic
void iroblifePeriodic(void);

#endif
```

lib2b.c

```c
#include "lib2b.h"
#include "sensing.h"
#include "oi.h"
#include "driving.h"

//! Previous IR sensor value
uint8_t irPrevious = 0;

//! True iff the robot should not move at all
uint8_t cannotRotateOrAdvance(void) {
    return getSensorUint8(SenBumpDrop) & MASK_WHEEL_DROP;
}

//! True iff the robot should not drive forward
uint8_t cannotAdvance(void) {
    return getSensorUint8(SenBumpDrop) || !(getSensorUint16(SenCliffLSig1) &&
            getSensorUint16(SenCliffFLSig1) && getSensorUint16(SenCliffFRSig1)
            && getSensorUint16(SenCliffRSig1));
}

//! Begin or continue driving forward if it is allowed, otherwise stop
void driveForwardIfAllowable(void) {
    if (cannotAdvance()) {
        // Shouldn't be moving forward: just stop
        driveStop();
    } else {
        // Forward being pressed and able to move forward: drive!
        drive(SPEED, RadStraight);
    }
}

//! Begin or continue turning
void turnContinuous(int16_t radius) {
    drive(SPEED, radius);
}

//! Called by overTurn periodically while turning.
//! Updates sensors and stops if unsafe to continue.
void doWhileTurning(void) {
    // Get most recent sensor values
    updateSensors();
    if (cannotRotateOrAdvance()) {
        // Shouldn't be moving: just stop
        driveStop();
    }
    // Keep going
}

//! Turn an extra 30 degrees. Does not return until fully turned.
//! Calls doWhileTurning periodically for sensor updating and safety.
void overTurn(int16_t radius) {
    driveAngleTFunc(SPEED, radius, OVER_TURN_ANGLE,
            &doWhileTurning, UPDATE_SENSOR_DELAY_PERIOD,
            UPDATE_SENSOR_DELAY_CUTOFF);
}

//! Called by irobPeriodic
void iroblifePeriodic(void) {
    // Get most recent sensor values
    updateSensors();
    // Get IR sensor value
    uint8_t ir = getSensorUint8(SenIRChar);
    if (cannotRotateOrAdvance()) {
        // Shouldn't be moving: just stop
        driveStop();
    } else {
        switch (ir) {
            case IR_FORWARD:
                // Drive forward if allowable
                driveForwardIfAllowable();
                break;
            case IR_LEFT:
                // Turn left
                turnContinuous(RadCCW);
                break;
            case IR_RIGHT:
                // Turn right
                turnContinuous(RadCW);
                break;
            default:
                // Movement button isn't being pressed
                switch (irPrevious) {
                    case IR_LEFT:
                        // Turn an extra 30 degrees
                        overTurn(RadCCW);
                        break;
                    case IR_RIGHT:
                        // Turn an extra 30 degrees
                        overTurn(RadCW);
                        break;
                    default:
                        // No buttons pressed and no over-turning left: stop
                        driveStop();
                }
                break;
        }
    }
    // Bookkeeping
    irPrevious = ir;
}
```

lib2b.h

```c
#ifndef LIB2B_H
#define LIB2B_H

#include <stdint.h>

// Driving constants
#define SPEED           (500)
#define OVER_TURN_ANGLE (30)

//! True iff the robot should not move at all
uint8_t cannotRotateOrAdvance(void);

//! True iff the robot should not drive forward
uint8_t cannotAdvance(void);

//! Begin or continue driving forward if it is allowed, otherwise stop
void driveForwardIfAllowable(void);

//! Begin or continue turning
void turnContinuous(int16_t radius);

//! Called by overTurn periodically while turning.
//! Updates sensors and stops if unsafe to continue.
void doWhileTurning(void);

//! Turn an extra 30 degrees. Does not return until fully turned.
//! Calls doWhileTurning periodically for sensor updating and safety.
void overTurn(int16_t radius);

//! Called by irobPeriodic
void iroblifePeriodic(void);

#endif
```

proj2a.c

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

#include "lib2a.h"

// Delay constant
#define IROB_PERIOD_MS  (1000)

int main(void) {
    // Submit to irobPeriodic
    setIrobPeriodicImpl(&iroblifePeriodic);

    // Initialize the Create
    irobInit();

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

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

proj2b.c

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

#include "lib2b.h"

int main(void) {
    // Submit to irobPeriodic
    setIrobPeriodicImpl(&iroblifePeriodic);

    // Initialize the Create
    irobInit();

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

        // Delay to avoid overflows
        delayMs(UPDATE_SENSOR_DELAY_PERIOD);
        // Wait if sensors coming in so next loop has clean start
        waitForSensors();
    }
}
```

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 byteTx(uint8_t value) {
  // Transmit one byte to the robot.
  // Wait for the buffer to be empty.
  while(!(UCSR0A & 0x20)) ;

  // Send the byte.
  UDR0 = value;
}

uint8_t byteRx(void) {
  // Receive one byte from the robot.
  // Call setupSerialPort() first.
  // Wait for a byte to arrive in the recieve buffer.
  while(!(UCSR0A & 0x80)) ;

  // Return that byte.
  return UDR0;
}

void baud(uint8_t baud_code) {
  // Switch the baud rate on both Create and module
  if(baud_code <= 11)
  {
    byteTx(CmdBaud);
    UCSR0A |= _BV(TXC0);
    byteTx(baud_code);
    // Wait until transmit is complete
    while(!(UCSR0A & _BV(TXC0))) ;

    cli();

    // Switch the baud rate register
    if(baud_code == Baud115200) {
      UBRR0 = Ubrr115200;
    } else if(baud_code == Baud57600) {
      UBRR0 = Ubrr57600;
    } else if(baud_code == Baud38400) {
      UBRR0 = Ubrr38400;
    } else if(baud_code == Baud28800) {
      UBRR0 = Ubrr28800;
    } else if(baud_code == Baud19200) {
      UBRR0 = Ubrr19200;
    } else if(baud_code == Baud14400) {
      UBRR0 = Ubrr14400;
    } else if(baud_code == Baud9600) {
      UBRR0 = Ubrr9600;
    } else if(baud_code == Baud4800) {
      UBRR0 = Ubrr4800;
    } else if(baud_code == Baud2400) {
      UBRR0 = Ubrr2400;
    } else if(baud_code == Baud1200) {
      UBRR0 = Ubrr1200;
    } else if(baud_code == Baud600) {
      UBRR0 = Ubrr600;
    } else if(baud_code == Baud300) {
      UBRR0 = Ubrr300;
    }
    sei();

    delayMs(100);
  }
}
```

utils/cmod.h

```
#ifndef INCLUDE_CMOD_H
#define INCLUDE_CMOD_H

  #include <avr/io.h>
  #include <avr/interrupt.h>

  // Setup the I/O pins.
  void setupIOPins(void);

  // Setup the serial port: Baud rate, transmit/recieve, packet size.
  void setupSerialPort(void);

  // Contains a collection of commands that allows me to "start" immediately
  // after calling this command.
  void initializeCommandModule(void);

  // Send and receive data from the Command Module
  void byteTx(uint8_t value);
  uint8_t byteRx(void);

  // Switch the baud rate on both Create and module
  void baud(uint8_t baud_code);
#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 drive(int16_t velocity, int16_t radius) {
    // Send the start driving command to the Create
    byteTx(CmdDrive);
    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);
    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;
    }
    // Start driving
    drive(velocity, radius);
    // Halt execution of new commands on the Create until reached angle
    byteTx(WaitForAngle);
    byteTx((uint8_t)((angle >> 8) & 0x00FF));
    byteTx((uint8_t)(angle & 0x00FF));
    // Stop the Create
    driveStop();
}

```

```

// # TIMER-BASED COMMANDS #

void driveDistanceTFunc(int16_t velocity, int16_t distance, void (*func)(void),
        uint16_t period_ms, uint16_t cutoff_ms) {
    // Calculate the delay
    uint32_t time_ms = (1000 * (uint32_t)distance) / (uint32_t)velocity;
    // Start driving
    drive(velocity, RadStraight);
    // Wait delay
    delayMsFunc(time_ms, func, period_ms, cutoff_ms);
    // Stop the Create
    driveStop();
}

void driveAngleTFunc(int16_t velocity, int16_t radius, int16_t angle,
        void (*func)(void), uint16_t period_ms, uint16_t cutoff_ms) {
    // Calculate the delay
    uint32_t time_ms = (PIe5 * TENTH_RADIUS * (uint32_t)angle)
        / (1800 * (uint32_t)velocity);
    // Start driving
    drive(velocity, radius);
    // Wait delay
    delayMsFunc(time_ms, func, period_ms, cutoff_ms);
    // Stop the Create
    driveStop();
}
```

utils/driving.h

```
#ifndef DRIVING_H
#define DRIVING_H

#include <stdint.h>

// # BASIC COMMANDS #

//! Drive at a certain speed in a certain direction.
/*!
 *  Returns immediately.
 *
 *  Directions: straight, clockwise, counterclockwise.
 *
 *  \param velocity     The speed in mm/s.
 *  \param radius       Either RadStraight, RadCW, or RadCCW (see oi.h).
 */
void drive(int16_t velocity, int16_t radius);

//! Stop the robot.
void driveStop(void);


// # OPCODE-BASED COMMANDS #

//! Drive a certain distance at a certain speed.
/*!
 *  Drive a certain distance using the Create wait for distance opcode.
 *
 *  \param velocity     The speed in mm/s.
 *  \param distance     The distance to travel in mm.
 */
void driveDistanceOp(int16_t velocity, int16_t distance);

//! Rotate a certain angle at a certain speed.
/*!
 *  Drive a certain angle using the Create wait for angle opcode.
 *
 *  \param velocity     The speed in mm/s.
 *  \param radius       Either RadCW or RadCCW (see oi.h).
 *  \param angle        The angle to rotate in degrees.
 */
void driveAngleOp(int16_t velocity, int16_t radius, int16_t angle);


// # TIMER-BASED COMMANDS #

//! Drive a certain distance at a certain speed.
/*!
 *  Drive a certain distance using a timer.
 *
 *  \param velocity     The speed in mm/s.
 *  \param distance     The distance to travel in mm.
 *  \param func         The function to execute periodically.
 *  \param period_ms    The interval to execute the function.
 *  \param cutoff_ms    The number of milliseconds before the end to stop
 *                      attempting to start the function.
 */
void driveDistanceTFunc(int16_t velocity, int16_t distance, void (*func)(void),
        uint16_t period_ms, uint16_t cutoff_ms);

//! Drive a certain angle at a certain speed.
/*!
 *  Drive a certain angle using a timer.
 *
 *  \param velocity     The speed in mm/s.
 *  \param radius       Either RadCW or RadCCW (see oi.h).
 *  \param angle        The angle to rotate in degrees.
 *  \param func         The function to execute periodically.
 *  \param period_ms    The interval to execute the function.
 *  \param cutoff_ms    The number of milliseconds before the end to stop
 *                      attempting to start the function.
 */
void driveAngleTFunc(int16_t velocity, int16_t radius, int16_t angle,
        void (*func)(void), uint16_t period_ms, uint16_t cutoff_ms);

#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;
    irobledUpdate();
}
```

utils/irobled.h

```
#ifndef IROBLED_H
#define IROBLED_H

#include <stdint.h>

// Colors for the power led.
#define POWER_LED_GREEN   (0x00)
#define POWER_LED_ORANGE  (0x40)
#define POWER_LED_RED     (0xFF)

// Bits for the other leds.
#define NEITHER_ROBOT_LED (0x00)
#define PLAY_ROBOT_LED    (0x02)
#define ADVANCE_ROBOT_LED (0x08)
#define BOTH_ROBOT_LED    (0x0A)

//! Send an led command to the Create.
void irobledCmd(uint8_t bits, uint8_t color, uint8_t intensity);
//! Update the leds. Probably won't have to use.
void irobledUpdate(void);
//! Initialize the leds to red for power and off for the others.
void irobledInit(void);

//! Set the color and intensity of the power led.
void powerLedSet(uint8_t color, uint8_t intensity);

// Functions for modifying one or both of the other leds.
void robotLedSetBits(uint8_t bits);
void robotLedOn(uint8_t led);
void robotLedOff(uint8_t led);
void robotLedToggle(uint8_t led);

#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);
}

// Ensure that the robot is OFF.
void powerOffRobot(void) {
  // If Create's power is on, turn it off
  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;
    }
  }
}
```

utils/iroblib.h

```c
#ifndef INCLUDE_IROBLIB_H
#define INCLUDE_IROBLIB_H

#include <avr/io.h>
#include <avr/interrupt.h>

// Constants
#define RESET_SONG 0
#define START_SONG 1

void defineSongs(void);
  // Songs
  // Indicator that the robot is Powered on and has reset.

void powerOnRobot(void);
void powerOffRobot(void);
  // Power the create On/Off.
#endif
```

utils/iroblife.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 irobPeriodicImplNull(void) {
}

void (*irobPeriodicImpl)(void) = &irobPeriodicImplNull;

void setIrobPeriodicImpl(void (*func)(void)) {
    irobPeriodicImpl = 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.
    // As a precaution for the robot and your grade.
    driveStop();

    // Play the reset song and wait while it plays.
    byteTx(CmdPlay);
    byteTx(RESET_SONG);
    delayMs(750);

    // Turn the power button on to red.
    irobledInit();
```

```c
}

void irobPeriodic(void) {
    // Call the user's periodic function
    irobPeriodicImpl();
    // Exit if the black button on the command module is pressed.
    if(UserButtonPressed) {
        irobEnd();
    }
}

void irobEnd(void) {
    // Stop the Create
    driveStop();
    // Power off the Create
    powerOffRobot();
    // Exit the program
    exit(1);
}
```

utils/iroblife.h

```c
#ifndef IROBLIFE_H
#define IROBLIFE_H

/*
 *  The irobPeriodic function in this library calls a function given to
 *  setIrobPeriodicImpl. The default value does nothing, but you can give
 *  it another function as a hook for periodically executed code.
 */

//! Default periodic function. Does nothing.
void irobPeriodicImplNull(void);
//! Set the function that irobPeriodic calls.
void setIrobPeriodicImpl(void (*func)(void));

//! Initialize the Create. Call this at the beginning of your main.
void irobInit(void);
//! Periodic operations. Call this in your main loop.
//! Calls the function last given to setIrobPeriodicImpl.
void irobPeriodic(void);
//! Stops and shuts down the Create, then exits. Call this to end the program.
void irobEnd(void);

#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(10);
    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);
}

void irobnprintf(uint16_t size, const char* format, ...) {
    // Create a buffer
    char formatted[size];
    char* fp = &formatted[0];
    va_list ap;
    va_start(ap, format);
    // Format the string
    vsnprintf(fp, size, format, ap);
    va_end(ap);
    // Print the string
    irobprint(fp);
}
```

utils/irobserial.h

```
#ifndef IROBSERIAL_H
#define IROBSERIAL_H

#include <stdint.h>
#include <stdarg.h>

#define SERIAL_CREATE       (1)
#define SERIAL_USB          (2)
#define SERIAL_SWITCHING    (0xFF)

#define PRINTF_BUFFER_SIZE  (0xFF)

//! Set the serial output (CREATE or USB)
//! Takes some time.
void setSerialDestination(uint8_t dest);

//! Get the serial output (CREATE or USB)
uint8_t getSerialDestination(void);

//! Print a string
void irobprint(char* str);

//! Print a formatted string (Max length: 255 bytes)
void irobprintf(const char* format, ...);

//! Print a formatted string (for strings longer than 255 bytes)
void irobnprintf(uint16_t size, const char* format, ...);

#endif
```

utils/oi.h

```
/* oi.h
 *
 * Definitions for the Open Interface
 */

#ifndef OI_H
#define OI_H

// Command values
#define CmdStart        128
#define CmdBaud         129
#define CmdControl      130
#define CmdSafe         131
#define CmdFull         132
#define CmdSpot         134
#define CmdClean        135
#define CmdDemo         136
#define CmdDrive        137
#define CmdMotors       138
#define CmdLeds         139
#define CmdSong         140
#define CmdPlay         141
#define CmdSensors      142
#define CmdDock         143
#define CmdPWMMotors    144
#define CmdDriveWheels  145
#define CmdOutputs      147
#define CmdSensorList   149
#define CmdIRChar       151
#define WaitForDistance 156
#define WaitForAngle    157


// Sensor byte indices - offsets in packets 0, 5 and 6
#define SenBumpDrop     0
#define SenWall         1
#define SenCliffL       2
#define SenCliffFL      3
#define SenCliffFR      4
#define SenCliffR       5
#define SenVWall        6
#define SenOverC        7
#define SenIRChar       10
#define SenButton       11
#define SenDist1        12
#define SenDist0        13
#define SenAng1         14
#define SenAng0         15
#define SenChargeState  16
#define SenVolt1        17
#define SenVolt0        18
#define SenCurr1        19
#define SenCurr0        20
#define SenTemp         21
```

```
#define SenCharge1      22
#define SenCharge0      23
#define SenCap1         24
#define SenCap0         25
#define SenWallSig1     26
#define SenWallSig0     27
#define SenCliffLSig1   28
#define SenCliffLSig0   29
#define SenCliffFLSig1  30
#define SenCliffFLSig0  31
#define SenCliffFRSig1  32
#define SenCliffFRSig0  33
#define SenCliffRSig1   34
#define SenCliffRSig0   35
#define SenInputs       36
#define SenAInput1      37
#define SenAInput0      38
#define SenChAvailable  39
#define SenOIMode       40
#define SenOISong       41
#define SenOISongPlay   42
#define SenStreamPckts  43
#define SenVel1         44
#define SenVel0         45
#define SenRad1         46
#define SenRad0         47
#define SenVelR1        48
#define SenVelR0        49
#define SenVelL1        50
#define SenVelL0        51


// Sensor packet sizes
#define Sen0Size        26
#define Sen1Size        10
#define Sen2Size        6
#define Sen3Size        10
#define Sen4Size        14
#define Sen5Size        12
#define Sen6Size        52

// Sensor bit masks
#define WheelDropFront  0x10
#define WheelDropLeft   0x08
#define WheelDropRight  0x04
#define BumpLeft        0x02
#define BumpRight       0x01
#define BumpBoth        0x03
#define BumpEither      0x03
#define WheelDropAll    0x1C
#define ButtonAdvance   0x04
#define ButtonPlay      0x01


// LED Bit Masks
#define LEDAdvance       0x08
#define LEDPlay         0x02
```

```
#define LEDsBoth        0x0A

// OI Modes
#define OIPassive       1
#define OISafe          2
#define OIFull          3


// Baud codes
#define Baud300         0
#define Baud600         1
#define Baud1200        2
#define Baud2400        3
#define Baud4800        4
#define Baud9600        5
#define Baud14400       6
#define Baud19200       7
#define Baud28800       8
#define Baud38400       9
#define Baud57600       10
#define Baud115200      11


// Drive radius special cases
#define RadStraight     32768
#define RadCCW          1
#define RadCW           -1



// Baud UBRRx values
#define Ubrr300         3839
#define Ubrr600         1919
#define Ubrr1200        959
#define Ubrr2400        479
#define Ubrr4800        239
#define Ubrr9600        119
#define Ubrr14400       79
#define Ubrr19200       59
#define Ubrr28800       39
#define Ubrr38400       29
#define Ubrr57600       19
#define Ubrr115200      9


// Command Module button and LEDs
#define UserButton        0x10
#define UserButtonPressed (!(PIND & UserButton))

#define LED1              0x20
#define LED1Off           (PORTD |= LED1)
#define LED1On            (PORTD &= ~LED1)
#define LED1Toggle        (PORTD ^= LED1)

#define LED2              0x40
#define LED2Off           (PORTD |= LED2)
#define LED2On            (PORTD &= ~LED2)
```

```
#define LED2Toggle        (PORTD ^= LED2)

#define LEDBoth           0x60
#define LEDBothOff        (PORTD |= LEDBoth)
#define LEDBothOn         (PORTD &= ~LEDBoth)
#define LEDBothToggle     (PORTD ^= LEDBoth)


// Create Port
#define RobotPwrToggle      0x80
#define RobotPwrToggleHigh (PORTD |= 0x80)
#define RobotPwrToggleLow  (PORTD &= ~0x80)

#define RobotPowerSense    0x20
#define RobotIsOn          (PINB & RobotPowerSense)
#define RobotIsOff         !(PINB & RobotPowerSense)

// Command Module ePorts
#define LD2Over         0x04
#define LD0Over         0x02
#define LD1Over         0x01

#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
        sensorIndex = 0;
        usartActive = 1;
        // Request all sensor data
        requestPacket(PACKET_ALL);
    }
}

void waitForSensors(void) {
    // Sensors data are coming in if usartActive is true
    while(usartActive);
}

void delayAndUpdateSensors(uint32_t time_ms) {
    // Update sensors while waiting
    delayMsFunc(time_ms, &updateSensors, 1, UPDATE_SENSOR_DELAY_CUTOFF);
}

uint8_t getSensorUint8(uint8_t index) {
    // Already in the right format
    return sensors[index];
}

int8_t getSensorInt8(uint8_t index) {
    uint8_t x = getSensorUint8(index);
    // Convert to signed; not implementation-dependent, and optimizes away
    return x < (1 << 7) ? x : x - (1 << 8);
}

uint16_t getSensorUint16(uint8_t index1) {
    // Combine msB and lsB
    return (sensors[index1] << 8) | sensors[index1 + 1];
}

int16_t getSensorInt16(uint8_t index1) {
    uint16_t x = getSensorUint16(index1);
    // Convert to signed; more opaque hex values b/c avr complains for 1 << 16
    return x < 0x8000 ? x : x - 0x10000;
}
```

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)
 *          Either Bumper       (v & MASK_BUMPER)
 *      Create Buttons          (packetId = PACKET_BUTTONS):
 *          Advance Button      (v & MASK_BTN_ADVANCE)
 *          Play Button         (v & MASK_BTN_PLAY)
 *
 *  \param packetId     The ID of the packet to retrieve, as defined by the
 *                      Create Open Interface.
 */
uint8_t read1ByteSensorPacket(uint8_t packetId);

//! Request all packets (will be retrieved by USART)
void updateSensors(void);

//! Wait for all packets to be recieved by USART
void waitForSensors(void);

//! delayMs that updates sensors
void delayAndUpdateSensors(uint32_t time_ms);

//! Get an unsigned 1-byte sensor value
uint8_t getSensorUint8(uint8_t index);

//! Get a signed 1-byte sensor value
int8_t getSensorInt8(uint8_t index);

//! Get an unsigned 2-byte sensor value, indexed by the more significant
//! (lower index) byte
uint16_t getSensorUint16(uint8_t index1);

//! Get a signed 2-byte sensor value, indexed by the more significant
//! (lower index) byte
int16_t getSensorInt16(uint8_t index1);

#endif
```

utils/timer.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 interrupt 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) {
```

```
    // Initialize the conditions for the delay loop
    uint32_t lastExec = time_ms;
    uint32_t nextExec = lastExec - period_ms;
    // Start the timer
    delayTimerRunning = 1;
    delayTimerCount = time_ms;
    // Wait until the timer runs out (delayTimerCount decrements every ms)
    while(delayTimerRunning) {
        // If it's before the cutoff and time for the next execution
        if (delayTimerCount > cutoff_ms && delayTimerCount <= nextExec) {
            // Execute the function
            lastExec = delayTimerCount;
            nextExec = lastExec - period_ms;
            func();
        }
    }
}
```

utils/timer.h

```c
#ifndef INCLUDE_TIMER_H
#define INCLUDE_TIMER_H

#include <avr/io.h>
#include <avr/interrupt.h>

// Interrupts.
ISR(TIMER1_COMPA_vect);

// Timer functions
void setupTimer(void);
void delayMs(uint32_t time_ms);

// Declaration of timer variables
extern volatile uint32_t delayTimerCount;
extern volatile uint8_t  delayTimerRunning;

//! Wait milliseconds, execute a function periodically.
/*!
 *  Executes a function at an interval until a cutoff has passed, returning
 *  after a total number of milliseconds have passed.
 *
 *  \param time_ms      The total number of seconds to wait.
 *  \param func         The function to execute periodically.
 *  \param period_ms    The interval to execute the function.
 *  \param cutoff_ms    The number of milliseconds before the end to stop
 *                      attempting to start the function.
 */
void delayMsFunc(uint32_t time_ms, void (*func)(void), uint16_t period_ms,
        uint16_t cutoff_ms);

#endif
```