```
/* main.c
 * Developed by Adam Ziegler
 * CSCI460 Senior Capstone project, 2010
 *
 * Demonstrates IR communication protocol between two Creates.
 * One partnered activity is available:
 *
 * 1)  Robot Duet - one robot generates a random song, then transfers
 * it to the other robot, and the pair "sing" together
 *
 * A second activity was drafted, but not implemented:
 *
 * 2)  Robot Dance - one robot generates a random set of moves, then
 * transfers it to the other robot; the pair separate and "dance"
 *
 * Adapted from "drive" sample program bundled with Create module.
 */

// Includes
#include <avr/interrupt.h>
#include <avr/io.h>
#include <stdlib.h>
#include "oi.h"

// song constants - defined by "drive" example program
#define RESET_SONG      0
#define START_SONG      1
#define BUMP_SONG       2
#define END_SONG        3

// custom song constants
#define FAILED_SEARCH_SONG                  4
#define POSSIBLE_TARGET_SONG            5
#define TARGET_LOST_SONG                        6
#define TARGET_ACQ_SONG                         7
#define BEGIN_ACTIVITY_SONG                 8
#define COMPLETED_ACTIVITY_SONG         9
#define FAILED_ACTIVITY_SONG            10
#define DUET_SONG                                       11

// state constants
#define STATE_NONE                          0           // catch-all state
#define STATE_IDLE                          1           // robot awaiting activity
specification
#define STATE_SEARCH                    2           // robot looking for a partner
#define STATE_TRACK                         3           // robot moving toward target
#define STATE_BEACON                    4           // robot linking IR physically
#define STATE_RELOCATE              5           // robot moving to new random position
#define STATE_LINKED                    6           // robots confirming linked IR
#define STATE_DUET_LEADER           7           // partnered activity - robots play a duet
of random notes (transmits song)
#define STATE_DUET_BACKUP           8           // partnered activity - robots play a duet
of random notes (receieves song)
//#define STATE_DANCE_LEADER  9           // partnered activity - robots synchronize
random movements (transmits dance) // dance not implemented
//#define STATE_DANCE_BACKUP  10    // partnered activity - robots synchronize random
movements (receieves dance) // dance not implemented

//TODO:  conform protocol to IR packets
#define IR_BEACON                   0x01 // used in SEARCH state to find robot locational
beacon
```

```
// IR opcode masks
#define IR_NONE                         0xFF // no IR signal received (according to
iRobot manuals)
#define IR_LOW_NIBBLE        0x0F // synonymous with opdata
#define IR_OPDATA                 0x0F // synonymous with low nibble
#define IR_HIGH_NIBBLE  0xF0 // synonymous with opcode
#define IR_OPCODE                 0xF0 // synonymous with high nibble

// data-flexible = low nibble (4 bits) may contain opcode bits OR high nibble (4 bits)
may contain data bits
// an 'opcode group' is a set of opcodes used in a single state
// opcode groups need to have exclusive opcode bits (i.e., only one bit in all opcode
bits can be set)
// maximum of eight opcodes per group (all opcode bits, no data bits)
// begin data-flexible opcode groups
#define IR_NEXT                         0x10 // ready for next packet of data (no data
bits)
#define IR_READY                  0x20 // ready for next group of packets (no data bits)
#define IR_DONE                         0x40 // signals correct checksum, and
transmission termination (no data bits)
#define IR_RETRY                  0x80 // checksum failed, retry sending data (no data
bits)

#define IR_SYNC1                  0x10 // used in BEACON and LINKED states to verify IR
transmitter alignment (no data bits)
#define IR_SYNC2                  0x20 // used in LINKED state to verify IR transmitter
alignment (no data bits)

#define IR_CHECKSUM               0x20 // used to verify data properly sent (5 data
bits)
#define IR_NOTE_LENGTH  0x40 // note length (6 data bits) **
#define IR_NOTE_TONE         0x80 // note tone (7 data bits) **
// end data-flexible opcodes

// activity and role constants
#define SELECT_DUET                     0
#define SELECT_DANCE              1
#define ACT_DUET_LEADER      2
#define ACT_DUET_BACKUP      3
#define ACT_DANCE_LEADER     4
#define ACT_DANCE_BACKUP     5
#define ROLE_LEADER                     6
#define ROLE_FOLLOWER             7

// Global variables; defined by example "drive" program
volatile uint16_t timer_cnt = 0;
volatile uint8_t timer_on = 0;
volatile uint8_t sensors_flag = 0;
volatile uint8_t sensors_index = 0;
volatile uint8_t sensors_in[Sen6Size];
volatile uint8_t sensors[Sen6Size];
volatile uint8_t recvbyte = 0;
// unused, but kept for reference
//volatile int16_t distance = 0;
//volatile int16_t angle = 0;

// cycle variables - determine time to stay in certain states
volatile uint16_t searchCycles = 0;                 // STATE_SEARCH, looking for
locational IR beacon
volatile uint16_t beaconCycles = 0;                 // STATE_BEACON, aligning data IR
transmitters
```

```
volatile uint16_t relocateCycles = 0;               // STATE_RELOCATE, moving/rotating
count
volatile uint16_t relocateMaxCycles = 0; // STATE_RELOCATE, moving/rotating maximum
volatile uint16_t linkedCycles = 0;                 // STATE_LINKED, aligning data IR
transmitters

// loop variables
volatile uint8_t i;
volatile uint8_t j;

// task variables
volatile uint8_t currentState = STATE_IDLE;
volatile uint8_t oldState = STATE_NONE;
volatile uint8_t irbyte = IR_NONE;
volatile uint8_t selectedActivity = SELECT_DUET;
volatile uint8_t duetNote = 0;
volatile uint8_t duetLength = 0;
volatile uint8_t buttonState = 0;

// robot role - only define one of these per robot (hard-coded roles)
//volatile uint8_t botRole = ROLE_LEADER;
volatile uint8_t botRole = ROLE_FOLLOWER;

// duet variables
#define DUET_LENGTH          2
volatile uint8_t duet_song[32];               // note:  max duet/song length is 16
volatile uint8_t duetChecksum = 0;

// dance variables - not implemented
//#define DANCE_LENGTH  4
//volatile uint8_t dance_steps[32];
//volatile uint8_t danceChecksum = 0;

// Functions
void byteTx(uint8_t value);
uint8_t byteRx(void);
void delayMs(uint16_t time_ms);
void delayAndUpdateSensors(unsigned int time_ms);
void initialize(void);
void powerOnRobot(void);
void baud(uint8_t baud_code);
void drive(int16_t velocity, int16_t radius);
void defineSongs(void);

// main program
int main (void)
{
  // Set up Create and module
  initialize();
  LEDBothOff;
  powerOnRobot();
  byteTx(CmdStart);
  baud(Baud57600);
  defineSongs();
  byteTx(CmdControl);
  byteTx(CmdFull);

  // Stop just as a precaution
  drive(0, RadStraight);

  // Play the reset song and wait while it plays
```

```
  byteTx(CmdPlay);
  byteTx(RESET_SONG);
  delayAndUpdateSensors(750);

  // main program loop
  for(;;)
  {
      // detect current state
      switch(currentState)
      {
                  case STATE_IDLE:
                        // wait for button input, to determine starting point for activity
                        for(;;)
                        {
                              delayAndUpdateSensors(15);
                              if(sensors[SenButton] == ButtonPlay)
                              {
                                    // Play button pushed, search for partner
                                    selectedActivity = SELECT_DUET;
                                    currentState = STATE_SEARCH;
                                    break;
                              }
                              else if(sensors[SenButton] == ButtonAdvance)
                              {
                                    // Advance button pushed, assume linked
                                    //selectedActivity = SELECT_DANCE; // dance not
implemented
                                    selectedActivity = SELECT_DUET;
                                    currentState = STATE_LINKED;
                                    break;
                              }
                        }

                        // make sure buttons are reset
                        delayAndUpdateSensors(1000);
                        break;

                  case STATE_SEARCH:
                        // searching for an IR beacon (the virtual wall mounted to another
bot)
                        // reset search cycles
                        searchCycles = 0;

                        // search for IR beacon
                        while(!sensors[SenVWall])
                        {
                              // gradually widen search arc
                              if(searchCycles < 1000) // we are searching in one location
for 15000 ms max.
                              {
                                    searchCycles += 1;

                                    // randomize on search cycles
                                    srand(searchCycles);
                              }
                              else
                              {
                                    // couldn't find beacon at largest turn
radius...relocate and try again
                                    oldState = STATE_SEARCH;
                                    currentState = STATE_RELOCATE;
```

```
                        break;
                    }

                    // increase turn radius by 1 mm every 5 search cycles
                    if((searchCycles % 5) == 0)
                    {
                        // new drive command for larger radius
                        drive(300, (searchCycles / 5));
                    }

                    // hazard checks
                    if(sensors[SenCliffL] || sensors[SenCliffFL] ||
sensors[SenCliffFR] || sensors[SenCliffR] || (sensors[SenBumpDrop] & BumpEither))
                    {
                        // back up a bit
                        drive(-200, RadStraight);
                        delayMs(1000);
                    }

                    // retrieve new sensor values;
                    delayAndUpdateSensors(15);
                }

                // full stop!
                drive(0, RadStraight);

                // small wait
                delayMs(100);

                // check if leaving state for right reason
                if(currentState == STATE_SEARCH)
                {
                    // target acquired!
                    LED1On;

                    //TODO:  new target acquired song
                    // play target acquire song
                    byteTx(CmdPlay);
                    byteTx(START_SONG);

                    // change state
                    currentState = STATE_TRACK;
                }
                else
                {
                    // the other reason...no beacon found!
                    // play beacon not found song
                    //TODO:  new beacon not found song
                    byteTx(CmdPlay);
                    byteTx(END_SONG);
                }

                break;

            case STATE_TRACK:
                // go forward on direct path to acquired target, slowly
                drive(100, RadStraight);

                // keep watching for the acquired beacon
                while(sensors[SenVWall])
                {
```

```
                        // check if we hit something
                        if(sensors[SenBumpDrop] & BumpEither)
                        {
                              // we hit something!  is it our target?
                              // full stop!
                              drive(0, RadStraight);

                              // play potential connect song
                              //TODO:  potential connect song
                              byteTx(CmdPlay);
                              byteTx(RESET_SONG);

                              // switch to beacon state
                              currentState = STATE_BEACON;

                              // exit this state loop
                              break;
                        }

                        // hazard checks
                        if(sensors[SenCliffL] || sensors[SenCliffFL] ||
sensors[SenCliffFR] || sensors[SenCliffR])
                        {
                              // turn a bit
                              drive(200, RadCCW);
                              delayMs(1000);

                              // drive forward a bit
                              drive(200, RadStraight);
                              delayAndUpdateSensors(1000);

                              // probably lost the beacon, so try it again
                              break;
                        }

                        // update sensors
                        delayAndUpdateSensors(16);
                  }

                  // check if leaving state loop for right reason
                  if(currentState == STATE_TRACK)
                  {
                        // keep going a bit
                        drive(150, RadStraight);

                        // small wait
                        delayMs(700);

                        // ...then stop!
                        drive(0, RadStraight);

                        // lost the beacon...
                        LED1Off;

                        //TODO:  new song for target lost
                        // play target lost song
                        byteTx(CmdPlay);
                        byteTx(END_SONG);

                        // try to reacquire
                        currentState = STATE_SEARCH;
```

```
                        }
                        else
                        {
                              // the other reason...we hit something!
                              // backup a bit
                              drive(-100, RadStraight);

                              // small wait
                              delayMs(700);

                              // ...then stop!
                              drive(0, RadStraight);

                              // small wait
                              delayMs(100);
                        }

                        break;

                  case STATE_BEACON:
                        // disable interrupts (necessary for any IR data transfer)
                        cli();

                        // reset beacon cycle count
                        beaconCycles = 0;

                        // spin around slowly to acquire acknowledging IR signal of
potential target
                        drive(100, RadCW);

                        // transmit pulse until received a sync1 packet
                        while(irbyte != IR_SYNC1)
                        {
                              // increment beacon cycle count
                              beaconCycles += 1;

                              // randomize on beacon cycles
                              srand(beaconCycles);

                              // check if we've been attempting this enough
                              if(beaconCycles > 1000) // spinning for 10000 ms
                              {
                                    // couldn't find IR beacon...maybe we hit a non-target
object
                                    LED1Off;

                                    // play failed beacon song
                                    //TODO:  new failed beacon song
                                    byteTx(CmdPlay);
                                    byteTx(BUMP_SONG);

                                    // relocate, then try to reacquire beacon
                                    oldState = STATE_SEARCH;
                                    currentState = STATE_RELOCATE;
                                    break;
                              }

                              // transmit beacon sync
                              byteTx(CmdIRChar);
                              byteTx(IR_SYNC1);
```

```
                        // force sensor update,  get IR byte
                        byteTx(CmdSensors);
                        byteTx(17);
                        irbyte = byteRx();
                  }

                  // full stop!
                  drive(0, RadStraight);

                  // check if leaving state for proper reason
                  if(currentState == STATE_RELOCATE)
                  {
                        // could not find other bot's IR transmitter
                        // reenable interrupts
                        sei();
                  }
                  else
                  {
                        // linked with other bot's IR transmitter
                        currentState = STATE_LINKED;
                  }

                  // full stop!
                  drive(0, RadStraight);

                  break;

            case STATE_LINKED:
                  // bot can see other's IR data transmitter
                  // deactivate interrupts again, just in case
                  cli();

                  // reset linked cycles
                  linkedCycles = 0;

                  // reset status LEDs
                  LEDBothOff;

                  // reset received signal
                  irbyte = IR_NONE;

                  // alternate between sync1 and sync2
                  while(irbyte != IR_SYNC2)
                  {
                        // another cycle
                        linkedCycles += 1;

                        // (re)seed random number
                        srand(linkedCycles);

                        // have we waited long enough?
                        if(linkedCycles > 1000) // waited for roughly 15000 ms with
no (1000 * 15 ms updates)
                        {
                              // timeout waiting for partner
                              // return to search state
                              currentState = STATE_SEARCH;
                              break;
                        }

                        // transmit sync1
```

```
                        byteTx(CmdIRChar);
                        byteTx(IR_SYNC1);

                        // transmit sync2
                        byteTx(CmdIRChar);
                        byteTx(IR_SYNC2);

                        // force sensor update, get IR byte
                        byteTx(CmdSensors);
                        byteTx(17);
                        irbyte = byteRx();
                  }

                  // check if leaving state for right reason
                  if(currentState == STATE_SEARCH)
                  {
                        // could not sync IR transmitters...reattempt search
                        // reenable interrupts
                        sei();
                  }
                  else
                  {
                        // had to go with hard-coded roles, since self-generated
ones didn't work out

                        // check this bot's role
                        if(botRole == ROLE_LEADER)
                        {
                              // check selected activity
                              if(selectedActivity == SELECT_DUET)
                              {
                                    // leader of duet
                                    currentState = STATE_DUET_LEADER;
                              }
                              else if(selectedActivity == SELECT_DANCE)
                              {
                                    // leader of dance - not implemented
//                                  currentState = STATE_DANCE_LEADER;
                              }
                        }
                        else if (botRole == ROLE_FOLLOWER)
                        {
                              // check selected activity
                              if(selectedActivity == SELECT_DUET)
                              {
                                    // follower of duet
                                    currentState = STATE_DUET_BACKUP;
                              }
                              else if(selectedActivity == SELECT_DANCE)
                              {
                                    // follower of dance - not implemented
//                                  currentState = STATE_DANCE_BACKUP;
                              }
                        }
                  }

                  break;

            case STATE_DUET_LEADER:
                  // initiate song generation
                  byteTx(CmdSong);
```

```
                        byteTx(DUET_SONG);
                        byteTx(DUET_LENGTH);

                        // generate random notes and lengths for number of song notes
                        for(i = 0; i < DUET_LENGTH; i += 1)
                        {
                              // generate tone and length
                              duet_song[i * 2] = rand() % 61 + 40; // 40 to 100; 0x28 to
0x64
                              duet_song[i * 2 + 1] = rand() % 25 + 8; // 8 to 32; 0x08 to
0x20

                              // add data to checksum
                              duetChecksum += duet_song[i * 2];
                              duetChecksum += duet_song[i * 2 + 1];

                              // program note
                              byteTx(duet_song[i * 2]);
                              byteTx(duet_song[i * 2 + 1]);
                        }

                        // play generated song
                        byteTx(CmdPlay);
                        byteTx(DUET_SONG);

                        sei();
                        delayMs(1000);
                        cli();

                        // clamp checksum to 5 bits and add opcode
                        duetChecksum = IR_CHECKSUM | (duetChecksum & 0x1F);

                        // reset received byte
                        irbyte = IR_NONE;

                        // retry transmission until success!
                        while(irbyte != IR_DONE)
                        {
                              // transmit all bytes
                              for(i = 0; i < DUET_LENGTH; i += 1)
                              {
                                    // reset received byte
                                    irbyte = IR_NONE;

                                    // transmit until confirmation received (IR_NEXT from
receiver)
                                    while(irbyte != IR_NEXT)
                                    {
                                          // transmit note tone
                                          byteTx(CmdIRChar);
                                          byteTx(duet_song[i * 2] | IR_NOTE_TONE); // 30
to 126, add opcode

                                          // get transmitted signal
                                          byteTx(CmdSensors);
                                          byteTx(17);
                                          irbyte = byteRx();
                                    }

                                    // reset received byte
                                    irbyte = IR_NONE;
```

```
                                    // transmit until confirmation received (IR_READY from
receiver)
                                    while(irbyte != IR_READY)
                                    {
                                          // transmit note length
                                          byteTx(CmdIRChar);
                                          byteTx(duet_song[i * 2 + 1] | IR_NOTE_LENGTH);
// 4 to 32, add opcode

                                          // get transmitted signal
                                          byteTx(CmdSensors);
                                          byteTx(17);
                                          irbyte = byteRx();
                                    }

                                    sei();
                                    LEDBothOn;
                                    delayMs(100);
                                    LEDBothOff;
                                    delayMs(100);
                                    cli();
                              }

                              // reset received byte
                              irbyte = IR_NONE;

                              // transmit checksum, then wait for a return signal
                              while((irbyte == IR_NONE) || ((irbyte != IR_DONE) && (irbyte
!= IR_RETRY)))
                              {
                                    // transmit checksum
                                    byteTx(CmdIRChar);
                                    byteTx(duetChecksum);

                                    // get transmitted signal
                                    byteTx(CmdSensors);
                                    byteTx(17);
                                    irbyte = byteRx();
                              }

                              // double check
                              if(irbyte == IR_DONE)
                              {
                                    // play activity completed song
                                    //TODO:  activity completed song
                                    byteTx(CmdPlay);
                                    byteTx(END_SONG);
                                    break;
                              }
                        }

                        // activity completed, return to idle mode
                        currentState = STATE_IDLE;

                        // reenable interrupts
                        sei();
                        break;

                  case STATE_DUET_BACKUP:
                        // backup singer - waits to receive notes from the leader
```

```
                        // reset checksum
                        duetChecksum = 0;

                        // expecting to receive this many pairs of bytes (DUET_LENGTH)
                        for(i = 0; i < DUET_LENGTH; i += 1)
                        {
                              // reset received byte
                              irbyte = IR_NONE;

                              // wait for note tone
                              while(!(irbyte & IR_NOTE_TONE) || (irbyte == IR_NONE) ||
(irbyte == IR_READY)) // reject IR_READY bounce-back
                              {
                                    // transmit IR_READY signal - leader expects this to
confirm length received (from prior loop)
                                    byteTx(CmdIRChar);
                                    byteTx(IR_READY);

                                    // get transmitted signal
                                    byteTx(CmdSensors);
                                    byteTx(17);
                                    irbyte = byteRx();
                              }

                              // received the note tone, add to song and checksum
                              duet_song[i * 2] = irbyte & 0x7F; // mask out note tone
opcode; up to 7-bit data (40 to 100 incl. offset)
                              duetChecksum += duet_song[i * 2];

                              sei();
                              LEDBothOn;
                              delayMs(200);
                              LEDBothOff;
                              delayMs(200);
                              cli();

                              // reset received byte
                              irbyte = IR_NONE;

                              // wait for note length
                              while(!(irbyte & IR_NOTE_LENGTH) || (irbyte == IR_NONE) ||
(irbyte == IR_NEXT)) // reject IR_NEXT bounce-back
                              {
                                    // transmit IR_NEXT signal - leader expects this to
confirm tone received
                                    byteTx(CmdIRChar);
                                    byteTx(IR_NEXT);

                                    // get transmitted signal
                                    byteTx(CmdSensors);
                                    byteTx(17);
                                    irbyte = byteRx();
                              }

                              // received the note length, add to song and checksum
                              duet_song[i * 2 + 1] = irbyte & 0x3F; // mask out note
length opcode; up to 6-bit data (8 to 32)
                              duetChecksum += duet_song[i * 2 + 1];

                              sei();
                              LEDBothOn;
```

```
                        delayMs(200);
                        LEDBothOff;
                        delayMs(200);
                        cli();
                  }

                  // reset received byte
                  irbyte = IR_NONE;

                  // wait for song checksum
                  while(((irbyte & IR_HIGH_NIBBLE) != IR_CHECKSUM) || (irbyte ==
IR_NONE) || (irbyte == IR_READY)) // ignore IR_READY bounceback
                  {
                        // transmit IR_READY, since leader expecting it to terminate
data transfer
                        byteTx(CmdIRChar);
                        byteTx(IR_READY);

                        // get transmitted signal
                        byteTx(CmdSensors);
                        byteTx(17);
                        irbyte = byteRx();
                  }

                  // clamp our calculated checksum to 5 bits and add opcode
                  duetChecksum = IR_CHECKSUM | (irbyte & 0x1F);

                  // got the checksum here
                  // check the checksum (ha!)
                  if(irbyte == duetChecksum)
                  {
                        // checksums match
                        // initiate duet song
                        byteTx(CmdSong);
                        byteTx(DUET_SONG);
                        byteTx(DUET_LENGTH);

                        // build song from received notes
                        for(i = 0; i < DUET_LENGTH; i += 1)
                        {
                              byteTx(duet_song[i * 2]);                 // note tone
                              byteTx(duet_song[i * 2 + 1]); // note length
                        }

                        // play received song
                        byteTx(CmdPlay);
                        byteTx(DUET_SONG);

                        // transmit burst of "done" packets, hopefully leader
catches at least one
                        for(j = 0; j < 100; j += 1)
                        {
                              byteTx(CmdIRChar);
                              byteTx(IR_DONE);
                        }

                        //TODO:  activity completed song
                        byteTx(CmdPlay);
                        byteTx(END_SONG);

                        // activity completed, return to idle mode
```

```
                        currentState = STATE_IDLE;

                        // reenable interrupts
                        sei();
                  }
                  else
                  {
                        // uh oh, checksum mismatch
                        byteTx(CmdPlay);
                        byteTx(BUMP_SONG);

                        // reset received byte
                        irbyte = IR_NONE;

                        // transmit until leader starts going again
                        while(!(irbyte & IR_NOTE_TONE) || (irbyte == IR_NONE))
                        {
                              // transmit retry packet
                              byteTx(CmdIRChar);
                              byteTx(IR_RETRY);

                              // get transmitted signal
                              byteTx(CmdSensors);
                              byteTx(17);
                              irbyte = byteRx();
                        }
                  }

                  break;

// "dance" activity not implemented
//                case STATE_DANCE_LEADER:
//                      break;
//
//                case STATE_DANCE_BACKUP:
//                      break;

                  case STATE_RELOCATE:
                        // pick random turn cycle count (this number * 15ms = total turn
time; from 1125 ms to 2250 ms)
                        relocateMaxCycles = (rand() % 75) + 75;

                        // reset relocation cycles
                        relocateCycles = 0;

                        // initiate turn at random speed (from 150 to 300), in random
direction (CCW or CW)
                        drive((rand() % 150) + 150, relocateMaxCycles % 2 == 0 ? 1 : -1);

                        // turn for random amount of time
                        while(relocateCycles < relocateMaxCycles)
                        {
                              relocateCycles += 1;

                              // check hazards
                              if(sensors[SenCliffL] || sensors[SenCliffFL] ||
sensors[SenCliffFR] || sensors[SenCliffR] || (sensors[SenBumpDrop] & BumpEither))
                              {
                                    // back up a bit
                                    drive(-200, RadStraight);
                                    delayMs(1000);
```

```
                                    drive(0, RadStraight);
                              }

                              // update sensors
                              delayAndUpdateSensors(15);
                        }

                        // pick random drive cycle count (this number * 15ms = total drive
time; from 1125 ms to 2250 ms)
                        relocateMaxCycles = (rand() % 75) + 75;

                        // reset relocation cycles
                        relocateCycles = 0;

                        // initiate drive at random speed (from 150 to 300)
                        drive((rand() % 150) + 150, RadStraight);

                        // drive for random amount of time
                        while(relocateCycles < relocateMaxCycles)
                        {
                              relocateCycles += 1;

                              // check hazards
                              if(sensors[SenCliffL] || sensors[SenCliffFL] ||
sensors[SenCliffFR] || sensors[SenCliffR] || (sensors[SenBumpDrop] & BumpEither))
                              {
                                    // back up a bit
                                    drive(-200, RadStraight);
                                    delayMs(1000);
                                    drive(0, RadStraight);
                                    break;
                              }

                              // update sensors
                              delayAndUpdateSensors(15);
                        }

                        // return to prior state
                        currentState = oldState;
                        oldState = STATE_NONE;

                        // full stop!
                        drive(0, RadStraight);

                        // small wait
                        delayMs(100);

                        break;
            }
      }

      // end of program
      return 0;
}

// Serial receive interrupt to store sensor values
SIGNAL(SIG_USART_RECV)
{
      uint8_t temp;

      temp = UDR0;
```

```
  if(sensors_flag)
  {
    sensors_in[sensors_index++] = temp;
    if(sensors_index >= Sen6Size)
      sensors_flag = 0;
  }
  else
  {
      recvbyte = temp;
  }
}

// Timer 1 interrupt to time delays in ms
SIGNAL(SIG_OUTPUT_COMPARE1A)
{
  if(timer_cnt)
    timer_cnt--;
  else
    timer_on = 0;
}

// Transmit a byte over the serial port
void byteTx(uint8_t value)
{
  while(!(UCSR0A & _BV(UDRE0))) ;
  UDR0 = value;
}

// receive byte from serial port
uint8_t byteRx(void)
{
      while(!(UCSR0A & 0x80)) ;
            return UDR0;
}


// Delay for the specified time in ms without updating sensor values
void delayMs(uint16_t time_ms)
{
  timer_on = 1;
  timer_cnt = time_ms;
  while(timer_on) ;
}

// Delay for the specified time in ms and update sensor values
void delayAndUpdateSensors(uint16_t time_ms)
{
  uint8_t temp;

  timer_on = 1;
  timer_cnt = time_ms;
  while(timer_on)
  {
    if(!sensors_flag)
    {
      for(temp = 0; temp < Sen6Size; temp++)
        sensors[temp] = sensors_in[temp];

      // Update running totals of distance and angle
      // unused, but kept for reference
```

```
//      distance += (int)((sensors[SenDist1] << 8) | sensors[SenDist0]);
//      angle += (int)((sensors[SenAng1] << 8) | sensors[SenAng0]);

      byteTx(CmdSensors);
      byteTx(6);
      sensors_index = 0;
      sensors_flag = 1;
    }
  }
}

// Initialize the Mind Control's ATmega168 microcontroller
void initialize(void)
{
  cli();

  // Set I/O pins
  DDRB = 0x10;
  PORTB = 0xCF;
  DDRC = 0x00;
  PORTC = 0xFF;
  DDRD = 0xE6;
  PORTD = 0x7D;

  // Set up timer 1 to generate an interrupt every 1 ms
  TCCR1A = 0x00;
  TCCR1B = (_BV(WGM12) | _BV(CS12));
  OCR1A = 71;
  TIMSK1 = _BV(OCIE1A);

  // Set up the serial port with rx interrupt
  UBRR0 = 19;
      UCSR0B = (_BV(RXCIE0) | _BV(TXEN0) | _BV(RXEN0));
      UCSR0C = (_BV(UCSZ00) | _BV(UCSZ01));

  // Turn on interrupts
  sei();
}

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
  }
}

// Switch the baud rate on both Create and module
void baud(uint8_t baud_code)
{
  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);
  }
}

// Send Create drive commands in terms of velocity and radius
void drive(int16_t velocity, int16_t radius)
{
  byteTx(CmdDrive);
  byteTx((uint8_t)((velocity >> 8) & 0x00FF));
  byteTx((uint8_t)(velocity & 0x00FF));
  byteTx((uint8_t)((radius >> 8) & 0x00FF));
  byteTx((uint8_t)(radius & 0x00FF));
}

// 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);

  // Bump song
  byteTx(CmdSong);
  byteTx(BUMP_SONG);
  byteTx(2);
  byteTx(74);
  byteTx(12);
  byteTx(59);
  byteTx(24);

  // End song
  byteTx(CmdSong);
  byteTx(END_SONG);
  byteTx(5);
  byteTx(77);
  byteTx(18);
  byteTx(74);
  byteTx(12);
  byteTx(72);
  byteTx(12);
  byteTx(69);
  byteTx(12);
  byteTx(65);
  byteTx(24);
}
```