CSCE374 – Robotics
Fall 2013 – Notes on the iRobot Create

This document contains some details on how to use iRobot Create robots.

1 Important Documents

These notes are intended to help you get started, but are not even close to being a replacement for the real documentation of the robot. The most important reference to consult is called:

iRobot Create Open Interface

Less important, but potentially interesting, are:

iRobot Command Module Owner's Manual
iRobot Command Module Quick Start Guide

All of these documents are available online.

2 Lab Facilities

The robots themselves live in Swearingen 3D22. Access to this room is controlled using a combination lock, for which the combination will be given in class.

The software you'll need has been installed on the workstations in this lab. Use your usual CSE department credentials to log in to these computers, and contact Pat O'Keefe if you have trouble logging in.

There should be plenty of hardware and space in the lab, but you are welcome to use your own laptops instead of the lab computers, if you so choose. As long as (1) you can demonstrate your project to me somewhere in Swearingen, (2) you don't choose hardware or software that trivializes the assignment, (3) your solution fits within the realm of common sense, then you are free to use whatever hardware or software you like.

The robots are only useful if their batteries are charged. Therefore, please be sure to return each robot to its home and plug in its charging cable when you're done working with it.

3 Which Computer Controls the Robot?

The Create robot itself has no user-accessible computer. Instead, we must connect an external computer to control the robot. There are two choices about how to do this:

- **Tethering to a "standard" computer**: Using a special serial cable, one can connect the Create to a standard laptop or desktop computer. The robot has a data port near its back right corner for this

purpose. This method provides lots of computing power, but requires the robot to remain tethered, or to carry a laptop around. In addition, because the serial connection is not always as reliable we might like, it takes special care to write robust software using this method.

- **Command Module**: One can also download programs to the green *Command Module*, which houses a microcontroller that communicates directly with the robot.

For this course, we'll stick with the Command Module. (In contrast, the Turtlebot robots we use in CSCE574 use the tethered method.)

4 Writing Programs for the Command Module

Here are the basic steps to write and execute a program for the command module:

- Use your favorite text editor to write or modify a C program.
- Prepare a makefile.
  - Obtain a copy of the standard Command Module makefile. This file, which is distributed by iRobot, is available from the course website.
  - Modify the makefile in three places:
    - Near line 59, change the value of the `TARGET` variable to the name of your program. (Hint: This should not end with "`.c`".)
    - Near line 60, change the value of the `SRC` variable to a list of source files that should be compiled into the complete executable.
    - Near line 203, change the value of the `AVRDUDE_PORT` variable to Command Module's device name. On our systems, this should usually be `/dev/ttyUSB0`. However, in some cases the trailing number may differ from 0.
- Compile the program using the command `make`. If there are compile errors, fix them and try again.
- Flash the program onto the Command Module.
  - Make sure that the Command Module is powered on.
  - Connect the workstation to the Command Module using a USB cable.
  - Press the reset button.
  - Use the command `make program` to store your program on the Command Module. If this works correctly, you should see a message something like `avrdude done`.
- Execute the program.
  - Turn the robot on, and wait for the power light to stop flashing.
  - Option 1: Remove the USB cable, then press the reset button.
  - Option 2: Leave the USB cable attached, then hold down the black button while pressing the reset button.
  - Either of these two options will restart the microcontroller and begin executing the most recent program flashed onto it.

5 Using the Command Module's Hardware

The previous section described how to write, compile, and execute programs for the Command Module. Of course, that information is not especially useful unless you know how those programs can interact with the world outside the microcontroller. Let's look at a few different ways to accomplish that.

5.1 Command Module LEDs

The Command Module is equipped with two LEDs that your program can turn or off. (This capability may not seem like much at the moment, but it's very likely that you'll find it useful for debugging later on.)

To learn how to access these LEDs, you must understand a bit about how the Command Module is organized. The microcontroller itself, a chip called the ATMega168, has three *input/output ports*, called B, C, and D. Each of these ports corresponds to 8 bits of data—one on each pin—that can be input to or output from the microcontroller. Some of those bits are available to hardware designers via the "ePorts" on the Command Module. Other bits are hard-wired to certain hardware within the Command Module, such as the LEDs. Furthermore, each pin can be configured to be used as either input or output.

We can learn from the Command Module Manual that the LEDs are attached to port D, pins 5 and 6, and that they are "active low," which means we should write a 0 to these bits to turn the LEDs on.

Here's an example program that turns on the right LED on the command module:

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

void setupRightLED(void) {
  // Set the fifth bit of the direction register for port D to 1.
  // This sets pin 5 of port D, which controls the right LED, to output mode.
  // Do this once, at the start of the program, before calling rightLEDon().
  DDRD |= (1 << 5);
}

void rightLEDon(void) {
  // Set the fifth bit of port D to 0.
  // This activates the right LED.
  PORTD &= ~(1 << 5);
}

int main(void) {
  setupRightLED();
  rightLEDon();
}
```

A few comments:

- Notice the special variable `DDRD`, which controls the input/output mode of each pin on port D. Also notice the special variable `PORTD`, which controls the actual output values from this port.
- Take special note of the bitwise operators we use to manipulate only certain bits of these values. It will be important in this course for you to be able to understand and use these operators.
- I leave it to you to write the functions to turn off or toggle the right LED, and set up and control the left LED.

5.2 Command Module Delays

There are times when it is useful to pause the execution of our program for a short interval of time. In most operating systems, we can accomplish this using standard library functions such as `sleep`, `usleep`, or `nanosleep`. Unfortunately, these functions are not available on our microcontroller.

Instead, we can get the same effect using the microcontroller's *interrupt* system to arrange for a given function to be called at regular intervals. If we use that function to decrement a global variable, then we can keep track of how much time has passed. Here's an example:

```
#include <avr/io.h>
#include <avr/signal.h>

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;
  OCR1A = 71;
  TIMSK1 = 0x02;
}

volatile uint16_t timerCount = 0;
volatile uint8_t timerRunning = 0;

void delayMs(uint16_t timeMs) {
  // Delay for the given number of milliseconds.
  // Call setupTimer() before this.
  timerCount = timeMs;
  timerRunning = 1;
  while(timerRunning) {
    // do nothing
  }
}

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

To use this, you might do something like this:

```
int main() {
  // Disable interrupts.  ("Clear interrupt bit")
  cli();

  // One-time setup operations.
  setupRightLED();
  setupLeftLED();
  setupTimer();

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

  // Toggle the LEDs once each second.
  while(2+2==4) {
    rightLEDon();
    leftLEDon();
    delayMs(1000);
    rightLEDoff();
    leftLEDoff();
    delayMs(1000);
  }

}
```

Notes:

- There's a setup step to request that the timer interrupts be generated. Note that we disable interrupts before this setup begins. This is a good idea to avoid any unexpected effects from interrupts that might occur before our setup is fully complete.
- Notice the `volatile` keyword attached to our two global variables. This lets the compiler know that the value of this variable might change unexpectedly (in this case, during the interrupt handler). Declaring a variable as `volatile` prevents the compiler from performing any optimizations that assume that the variable's value won't change.
- These variables are declared with types `uint16_t` and `uint8_t`. Here, `uint` means "unsigned integer," the number indicates the number of bits used to store that integer. It's often recommended when programming microcontrollers to use these explicit types (instead of the generic `int` and `long`) to ensure that you're getting the data type that you really need.
- Related point: Be very cautious about using floating point types like `float` and `double`. This hardware does support those type directly, so any operations on those types must be done in software. The result is that using even simple operations like addition of two floating point numbers can massively increase the size of your program—bad news, given the microcontroller's limited storage space.
- Notice the infinite loop in the `main` function here. This program never exits, but of course that's not a problem, since there's no operating system to return control to.

5.3 Sending Data to the Create

So far we have only discussed the hardware available within the Command Module itself. Now let's consider how we can communicate with the robot itself. The Command Module is connected to the Create robot by a serial connection that transmits one byte at a time.

To send a byte to the robot, there are two steps:

- Wait for the serial transmit buffer to become empty. We can achieve this by monitoring bit 5 of the status register called `UCSR0A`. When this bit becomes 0, the UART (Universal Asynchronous Receiver/Transmitter—the hardware that handles serial communications) is ready for another byte.
- Write the value we want to send to the register `UDR0`. This value will be sent asynchronously—that is, our program will continue without waiting for the operation to complete—to the robot.

Here's what that process looks like, along with some initialization code for the serial port:

```
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 = 0x18;

  // Set 8-bit data.
  UCSR0C = 0x06;
}

void byteTx(uint8_t value) {
// Transmit one byte to the robot.

  // Wait for the buffer to be empty.
  while(!(UCSR0A & 0x20)) {
    // Do nothing.
  }

  // Send the byte.
  UDR0 = value;
}
```

Here's a example main function that blinks the LEDs on both the Create and the Command Module:

```
int main(void) {
  // Disable interrupts.  ("Clear interrupt bit")
  cli();

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

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

  byteTx(128);  // Start the open interface.
  byteTx(132);  // Switch to full mode.

  // Toggle the LEDs once each second.
  while(2+2==4) {
    rightLEDon();
    leftLEDoff();
    byteTx(139); // Opcode for "Set LEDs"
    byteTx(10);  // Led bits: both on
    byteTx(0);   // Power led color: Fully green
    byteTx(255); // Power led intensity
    delayMs(1000);

    rightLEDoff();
    leftLEDon();
    byteTx(139); // Opcode for "Set LEDs"
    byteTx(0);   // Led bits: both off
    byteTx(255); // Power led color: Fully red
    byteTx(255); // Power led intensity
    delayMs(1000);

  }

}
```

Notes:

- The specific values that we send to the robot probably seem completely mysterious at this point. In fact, the document called "iRobot Create Open Interface" defines the protocol for the how robot interprets the data we send. This document will be your primary source for information about how to communicate with the robot; I won't repeat its contents here. It's likely that you'll refer to the interface document many times throughout the semester.

  In this case we are sending a "Start" command (page 7), switching the robot to "Full Control" mode (page 7) so that it will respond to our commands, and sending a "Set LEDs" command (page 9) to toggle the Create's lights on and off.

- I hope that the coding style in this example makes you want to vomit in your mouth a little (figuratively, that is). All of those "magic numbers" should, of course, be declared as constants or macros (so that the meaning is more clear) and the specific commands to the robot should be encapsulated into their own functions (to improve readability and to eliminate code duplication). I'm using this poor style here to keep the example as short as possible. You won't have the same excuse for the assignments you turn in.

5.4 Reading Data from the Create

The previous example shows how you can send data to the robot. What about moving data in the other direction? The process is very similar: We wait for the send buffer to become empty, and then write the byte we want to send to a specific register:

```
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;
}
```

Here's an example of how you might ask the robot about its bump sensors, and toggle the Command Module's LEDs to show which bump sensors are triggered:

```
int main() {
  // Disable interrupts.  ("Clear interrupt bit")
  cli();

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

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

  byteTx(128);  // Start the open interface.
  byteTx(132);  // Switch to full Mode

  while(2+2==4) {
    delayMs(100);

    // Ask the robot about its bump sensors.
    byteTx(142); // Opcode for "Read sensors"
    byteTx(7);   // Sensor packet 7: Bumps and wheel drops

    // Read the one-byte response and extract the relevant bits.
    uint8_t bumps = byteRx();
    uint8_t bumpRight = bumps & (1 << 0);
    uint8_t bumpLeft =  bumps & (1 << 1);

    // Set the command module LEDs based on this sensor data.
    if(bumpLeft) {
      leftLEDon();
    }
    else {
      leftLEDoff();
    }
    if(bumpRight) {
      rightLEDon();
    }
    else {
      rightLEDoff();
    }
  }

}
```

Notes:

- This example uses the "Read Sensors" command from the Create Open Interface. This is one of the most complicated commands, but also one of the most important. Be sure to check the number of bytes and their formats carefully.
- This is another example of bad coding style for the sake of brevity. You'll want to write functions and use appropriate constants, to hide the details of the protocol.

5.5 Sending Debug Messages Via USB

In addition to its use for flashing programs, the USB connection to the command module can also be used in the opposite direction, to send data back from a running program to your workstation. This capability can provide a very useful debugging tool.

There are three major details of which you'll need to be aware: First, the the Command Module has only one UART. As a result, if your program wants to communicate with both the robot and with the workstation, you must reconfigure the Command module each time you switch between the two. Here's a code snippet that performs the switch:

```
#define SERIAL_CREATE (1)
#define SERIAL_USB (2)

void setSerialDestination(uint8_t dest) {
// 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);
}
```

The good news here is that the `byteTx` function that we've already discussed for talking to the robot is the same one that we can use to talk to the workstation. Here's a snippet that illustrates how you might use this:

```
while(1) {
  setSerialDestination(SERIAL_CREATE);
  uint8_t bumpLeft, bumpRight;
  readBumps(&bumpLeft, &bumpRight);

  setSerialDestination(SERIAL_USB);
  if(bumpLeft) byteTx('L');
  if(bumpRight) byteTx('R');
  byteTx('\n');
}
```

You will likely want to write functions to send strings (ask Google about "null terminated string" if you're not familiar with how C represents strings) and numbers in a human-readable format.

Second, we should be sure to run the program with the USB cable plugged in. See "Option 2" in Section 4 for the details about how to start your program without removing the USB cable. Note that the Command Module stops your program and reverts to its bootloader any time you connect the USB cable.

Third, we need a way to actually see the output that your program is sending. There are a few different ways to do this, including programs such as `minicom`, `screen`, or (if you really want to) `cat`. To keep things simple, I've included source code on the course website for a simple program that opens the serial port with the correct settings and dumps its output to your terminal.