Coding a solo iRobot Create (and making it sing)
Steven Knipe
Los Altos Community Team 0636 (Cupertino division), knipesteven@gmail.com

Coding a solo iRobot Create
(and making it sing)

1. Introduction+Warning

The iRobot Create is a valuable tool, but many are not using it to its full potential. The Create has its own processor on it, allowing it to be used as a third, independent robot. Even if a third robot is not what you want, using Create-specific commands provided by the manufacturer allows for significantly increased precision in driving and the potential for a super-powered motor. The Create also has a speaker which can be used to play music. I have written a library (distributed with this paper) that provides an interface to do these types of things.

I am not responsible for anything that happens as a consequence of using this code. While executing code this way, the Create will not respond- even to a stop command. This library is not thread-safe. We haven't had many problems so far, but I can offer no guarantees.

2. Sources/Thanks

I would like to thank Jeremy Rand (Then Norman Advanced Robotics, now Team SNARC) for his excellent papers several years ago that contain much of this information. I would also like to thank iRobot for releasing the Open Interface documentation, which can be found at http://www.irobot.com/filelibrary/pdfs/hrd/create/Create%20Open%20Interface_v2.pdf. I would also like to thank KIPR for hosting all of the code for the LINK and CBC at https://github.com/kipr. Making this code available made it vastly easier to figure out which functions I need to call.

3. Sending Bytes

All communication to the Create is done at the individual byte level. The LINK code is extremely object-oriented which obfuscates this fact, but a more clear version can be found in both the CBC code located at https://github.com/kipr/cbc/blob/master/userlib/libcbc/src/create.c and in the XBC source code which is located in the lib folder of the installation. Each system has its own way to send those bytes. The XBC uses the command serial_write_byte(int byte). The CBC uses create_write_byte(char byte), and the LINK provides that function as well. It uses a 'char' data type because a single ASCII character is composed of a single byte, which can be represented by either the ASCII character itself or as a number directly.

While everything must be sent one byte at a time, things like a speed value (-500 to +500) require more than a single byte to store. These types of values use two bytes put together (a high byte and a low byte). To make this simpler, we use a function to split the integer input into a two-byte output with binary arithmetic. This will be the only time we ever use binary.

```
#define get_high_byte2(a) ((a)>>8)
#define get_low_byte2(a) ((a)&255)
void create_write_int(int integer)
{
	create_write_byte(get_high_byte2(integer));
	create_write_byte(get_low_byte2(integer));
}
```

4. Driving and Waiting

The standard KIPR "create_drive_direct" command sends several bytes to tell the Create to drive each wheel at a specific speed. These bytes are described on page 9 of the Create Open Interface. The first byte value of 145 is called the 'opcode' which identifies the requested command. This is followed by two separate 2-byte integers for the left and right wheel speeds. If you look at the CBC (or XBC) code for create_drive_direct, this is exactly what you'll find:

```
create_write_byte(145);
create_write_byte(HIGH_BYTE(r_speed));
create_write_byte(LOW_BYTE(r_speed));
create_write_byte(HIGH_BYTE(l_speed));
create_write_byte(LOW_BYTE(l_speed));
```

What is sent after these first bytes makes the driving experience different. To get the Create to drive a certain distance, most of you probably use either time or a loop that constantly asks the Create how far it's gone and sends the next command once it has gone far enough. The loop version means that the Create knows how far it's gone. Wouldn't it be nice if the Create could just handle that for you? Actually, it can. Opcodes 156 and 157 are "wait" commands for distance and angle respectively, both taking a 2-byte value. Opcode 158 waits for a specific "event", which can be things like the pressing a button or hitting the bump sensor on the Create. See page 16 of the open interface document for more details.

A single drive command just turns on the wheels using certain values and then waits for a certain distance or angle. Doing this repeatedly allows the program to send an entire drive path at once, cutting the LINK out of the loop entirely. This is far more reliable since there is no communications lag between the LINK and the Create. To drive forward a certain distance, you can just send a drive command, wait a given distance, then send the stop command (which is Drive at 0 speed).

5. Communicating Back

However, this means that the LINK doesn't know when the Create is done moving. Once you send a drive path, the Create will complete the entire drive path, never communicating back to the LINK. This means that you have no way of knowing when the Create is finished doing its job. This is a fixable problem. We can ask for data from the Create, and wait for it to respond. This is actually extremely easy on the LINK thanks to KIPR providing the create_read_block() command. This in turn has been simplified in my library to the "create_block()" function which requests button data from the Create, and waits for the Create to respond with that data. This, combined with the preset drive commands, allows a user to simply send a set of drive commands and end the drive path with a create_block().

6. Upsides and Downsides

Offloading the drive path processing to the Create gives one major upside: precision. Note the difference between precision and accuracy: the robot will do almost the same thing every time (precision), but the action may not be reflective of the the numbers you entered (accuracy). For example, if you're running at maximum speed, to do a 90 degree turn, you actually need to turn approximately 65 degrees. The effects of this error significantly decreases at a slower speed, but this means that increasing the speed has a drastic effect on the overall path of the robot. The robot remains repeatable even at higher speeds. Note that stopping the Create does introduce some variability, as does using the create_block() function. If you can keep everything in a single drive path though, it is very precise compared to the alternatives.

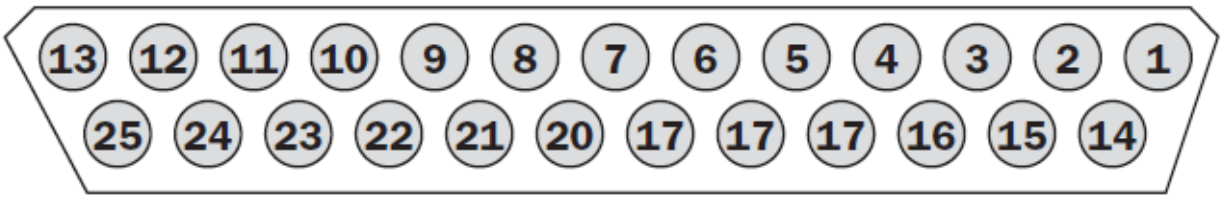
The ability to use a third robot should not be discounted either. The easiest method is to use a Create as a blocker that just drives to the opponent's side. Note however that the other two robots will require two motors each for driving. In recent years, this leaves very few motors for manipulators.

While there are many upsides to using these tricks, there are downsides as well. The biggest one is that the Create cannot respond to any new commands while it's in the middle of waiting - such as waiting for a distance while driving. This includes, for example, a "stop" command sent at the end of the round. This behavior caused one of the Los Altos Community Teams to lose a match during our regionals when they used the standard "shut_down_in" command, which does not stop the Create while it's driving. The best solution we've found is to avoid using the Create's sensors as a wait event, so that as long as the wheels can move it will finish. If possible, we also like to have a small servo attached to the power button of the Create. When time runs out, it can simply press the button and shut off the Create.

7. Super Motor

Using just the drive and wait commands, you can have an independent blocker. But what if you want to do something more? Few robots can make do with solely drive commands - they need manipulators! Luckily, we have three motor ports - two that are a bit weaker than normal, but the last is theoretically 1.5x more powerful and actually appears to be much more than that. NEITHER MYSELF NOR KIPR ARE RESPONSIBLE FOR BROKEN MOTORS OR ANY INJURIES. IF YOU USE

THIS MOTOR PORT TO ITS FULL POWER, YOU WILL BREAK MOTORS. (And to the KIPR support desk: yes, all the broken motors that we've given to you are ones we've broken in normal operation.) So, how do these motors work? First, one must look at the Open Interface Cargo Bay connector explanation found on page 4. There is an error in that for some reason the numbers 18 and 19 have been replaced by 17s.



The pins that we care about the most here are three Low Side Drivers in the Create cargo bay (the bunch of pins in the center of the robot). If you plug a motor into one of the three 1.5A Switched Vpwr ports (pin 10, 11, or 12) and into a low side driver (22 or 23 for normal, 24 for the extra power) you can use a motor by opening the low side driver using opcode 144 (assuming you want to have variable power).

There are two easy ways to set up the motors - one powerful motor from 24 to 10, or two weaker ones from 12 to 23 and 11 to 22. If you want a strong motor and a weak motor (or all three), you will need to use the motor extension cables to plug the pins into different sockets.

10	Switched Vpwr	Provides battery power @ 1.5 A when Create is powered on.
11	Switched Vpwr	Provides battery power @ 1.5 A when Create is powered on.
12	Switched Vpwr	Provides battery power @ 1.5 A when Create is powered on.

22	Low side driver 0	0.5A low side driver from Create
23	Low side driver 1	0.5A low side driver from Create
24	Low side driver 2	1.5A low side driver from Create

These motors are run by the current provided by the Switched Vpwr, which is monodirectional. As such, your motors connected here can only turn one direction (dependant upon how the motor is plugged into the cargo bay).

8. Sensors

The Create wait event has a hidden use. Using the cargo bay connector, you can plug sensors

into the Create and wait for them to trigger. Using this, you can make the Create wait for a light sensor. As far as I know, the Create can only read digital sensors. Supposedly it will interpret the light sensor's analog signal in such a way that it will usually be correct - turning the sensor's analog input into a simple digital signal. However, I have been unable to get this to work reliably thus far and instead use a simple touch sensor.

If you look at the sensors connectors, you'll see that there are only two pins that are actually connected to the wires. One should go into a digital output, and the other should be placed in a digital input. To use the sensor you turn on the output, then do a wait event for either the digital input or its reverse, depending on what you're waiting for.

9. Message Buffer and Scripting

There is one more limitation that should be mentioned. The LINK (or equivalent) sends a series of bytes to the Create which are interpreted as commands. If a command is a wait, the Create will stop reading until the wait command has completed. The bytes after the wait command are stored in a buffer. That buffer has a maximum of 255 bytes. A drive+wait set takes 8 bytes total, so you can store a maximum of 31 drive commands in a single drive path. While most programs won't reach that limit, it is something to be conscious of. And sending many move commands to a Create that's waiting for an event can cause unexpected behavior (if a command is cut off in the middle).

If your program really does need just a few more drive commands, however, there is a trick you can use. The Create allows you to store a "script", 100 bytes long, that can be called with a mere 1 byte. If you preload the script onto the Create, you can get 354 total bytes of programming or potentially even more if you have some repetition you can stick into the script. This is rarely used but is described in more detail on page 15 of the Open Interface.

10. Singing

Now for the fun part: making the Create sing! If you use the libraries I provided, it's actually quite simple. You start defining a song by calling init_song(), then send a series of note() commands. You load the song with song(). It can then be played at any time with the play() command. The play() command can only be used when the Create is connected to a KIPR controller. If you want the Create to sing independently, you can use the sing() command instead of play(). However, you may notice a small pause every 16 notes since the songs are played at 1/64th of a second, but the wait command has a precision of only 1/10th of a second.

The Create defines a "song" as a set of 16 notes and can store up to 16 songs. This gives a maximum length of 256 notes. If more is required, you can load data into "song" slots that have already been played while another is playing, but my current library does not support that. Note that starting the next set of 16 notes will only work if the Create's buffer is empty - which means you do NOT want to try to make it sing while it's driving. All of this is handled automatically for you if you use the play() command, which will block execution of other code until it is complete.

All of the byte code information can be found on pages 11 through 13 of the Open Interface.

There is a list of frequencies for the notes that you would want to play, but I often found it annoying to constantly cross-reference between the music and the actual notes. By using a #define'd scale at the top, it became far easier to write music. I also have a basic syntax for adding or subtracting one octave (which is 12 separate notes once you include the flats/sharps).

11. Reversal

Often my team has found that we want to mount a claw on the Create such that we're driving "backwards" the vast majority of the time. Since this library provides an abstraction layer, it can be relatively easily switched such that the flat side of the Create is what you code as "forward". I have included a separate library of the Create code that's reversed. Since create_drive_direct is hardcoded, it has been replaced with the create_direct function which does the same thing, but backwards.