

KIPR Link Manual



Version: BB2014.1.1

Copyright 2013 KISS Institute for Practical Robotics. All rights reserved.

KIPR makes no warranty for the use of its products and assumes no responsibility for any errors which may appear in this document nor does it make a commitment to update the information contained herein. KIPR products are not intended for use in medical, life saving or life sustaining applications. KIPR retains the right to make changes to these specifications at any time, without notice.

BOTBALL[®], BYO-BOT[®], BOTGUY, and the BOTGUY design and character are trademarks and/or service marks of KISS Institute for Practical Robotics and may not be used without express written permission.

LEGO, iRobot, and iRobot Create are registered marks of their respective owners.

The KISS Institute is a 501-c3 nonprofit organization. Our mission is to improve the public's understanding of science, technology, engineering, and math; develop the skills, character, and aspirations of students; and contribute to the enrichment of our school systems, communities, and the nation.

Contents

1. KIPR Link --- 6

| | |
|--------------------------|---|
| About the KIPR Link | 6 |
| KIPR Link Basic Features | 6 |
| Input and Output | 6 |
| Other Features | 7 |
| Included Hardware | 7 |
| KIPR Link Features | 8 |

2. Quick Start --- 9

| | |
|--|----|
| Turning On Your KIPR Link | 9 |
| Checking the Firmware Version on Your KIPR Link | 9 |
| Installing the KISS Platform on Your Computer - the KISS IDE | 10 |
| Mac (OS X 10.4 and higher - 64 bit processors) | 10 |
| Windows (XP, Vista, or 7) | 10 |
| Windows 8 | 11 |
| Downloading and Running a Program for Your KIPR Link | 12 |
| 1. Connect the KIPR Link to your Computer | 12 |
| 2. Launch the KISS IDE | 12 |
| 3. Create a New Program | 12 |
| 4. Select the Communications Port | 12 |
| 5. Edit/Save/Download the Program File | 13 |
| 6. Locate the Program File on the KIPR Link | 13 |
| 7. Compile and Run the Program on the KIPR Link | 13 |
| 8. Observe Results on the KIPR Link Display | 14 |

3. Programming for the KIPR Link --- 15

| | |
|--|----|
| Using the C Programming Language with the KIPR Link and the KISS IDE | 15 |
| KIPR Link Function Libraries | 17 |
| Using Sensors | 17 |
| KIPR Link Library Functions for Sensors | 18 |
| Using Servo Motors | 19 |
| KIPR Link Library Functions for Servo Motors | 20 |
| Using DC Drive Motors | 21 |

| | |
|--|-----------|
| KIPR Link Library Functions for DC Motors | 22 |
| Other Functions Commonly Used With the KIPR Link | 23 |
| 4. KIPR Link Vision System | 24 |
| About Color Vision Tracking and QR Codes | 24 |
| Setting Up KIPR Link Color Tracking Channels | 25 |
| Setting Up a KIPR Link QR Scanning Channel | 27 |
| Verifying Channel Behavior | 28 |
| KIPR Link Vision Library Functions | 28 |
| Sample color tracking program controlling a servo motor | 30 |
| Set Up | 30 |
| Code | 31 |
| Sample color tracking program controlling motor lights | 32 |
| Set Up | 32 |
| Code | 33 |
| Sample program for decoding a QR code | 34 |
| Code | 34 |
| 5. Troubleshooting | 35 |
| 6. Appendices | 38 |
| Updating the KIPR Link Firmware | 38 |
| Controlling an iRobot Create with the KIPR Link | 39 |
| iRobot Create KIPR Link Library Functions | 40 |
| Sample Program for Controlling an iRobot Create with the KIPR Link | 43 |
| Set Up | 43 |
| Code | 43 |
| Writing an iRobot Create Script | 44 |
| Example | 45 |
| Code | 45 |
| Sample Program for Using KIPR Link Digital Output to Light an LED | 48 |
| Setup | 48 |
| Code | 48 |
| Sample Program Using a Thread for Monitoring a Sensor | 49 |
| Code | 49 |

| | |
|---|-----|
| File I/O for a USB Flash Drive Plugged into the KIPR Link | 50 |
| Creating your own sensor | 50 |
| Tools Needed | 52 |
| Supplies needed | 52 |
| Method | 52 |
| Creating your own motor | 54 |
| Tools Needed | 54 |
| Supplies needed | 54 |
| Method | 54 |
| Setting the sensor ports for 5V or 3.3V | 566 |
| Warning! This modification requires opening your KIPR Link case, which will void your warranty. | |
| KIPR assumes no liability for the accuracy of these instructions and following them is strictly at your own risk regarding any damage which might occur to either person or equipment employed. | 56 |
| KIPR Link Main Library Functions | 58 |
| KIPR Link Vision Library Functions | 67 |
| KIPR Link iRobot Create Library Functions | 69 |
| Create serial interface functions | 69 |
| Create configuration functions | 70 |
| Create movement functions | 70 |
| Create sensor functions | 70 |
| Create battery functions | 69 |
| Create built-in script functions | 69 |
| LED and music functions | 69 |

1. KIPR Link

About the KIPR Link

The KIPR Link is a Linux-based robot controller designed by the KISS Institute for Practical Robotics (KIPR). It is most easily accessed by installing the KISS IDE (Integrated Development Environment) on your computer, software created and maintained by KIPR to support the KIPR Link.

Featuring significant hardware and usability improvements over its predecessor, the KIPR Link is both a beginner-friendly choice for newcomers to robotics, and a powerful, feature-rich device that will appeal to experts.

KIPR Link Basic Features

- GNU/Linux based operating system
- Open-source robot control software
- Integrated color vision system
- 800MHz ARMv5te processor
- Spartan-6 FPGA
- Integrated battery and charge system
- Internal speaker
- 320 x 240 color touch screen

Input and Output

- 1 - 3 axis 10-bit accelerometer (software selectable 2/4/8g)
- 8 - digital I/O ports (hardware selectable 3.3V or 5V)
- 8 - 3.3V (5V tolerant) 10-bit analog input ports
- 4 - servo motor ports
- 4 - PID motors ports with full 10-bit back EMF and PID motor control
- 1 - 3.3V (5V tolerant) TTL serial port

- 2 - USB 2.0 (type A) host ports for connecting devices
- 1 - USB Micro-B port to connect to your computer
- 1 - physical button
- 1 - IR emitter
- 1 - IR receiver
- 1 - HDMI port

Other Features

- All sensor inputs have software enabled pull up resistor (digital 47k, analog 15k)
- Motor current up to 1A per port
- Servo ports output 6V
- I2C interface (with additional hardware)
- Arm 7 debug port (with additional hardware)
- JTAG port
- V_{cc} maximum current 500mA @3.3V, 1A @5V
- 7.4V 2000mAh Lithium Polymer battery pack (2s1p) 8C max discharge
- 1GB micro SD for storage
- Internal 802.11 b/g wifi

Included Hardware

The basic KIPR Link hardware includes the KIPR Link, USB cable, power adapter and USB camera. It is important that only the supplied charger (or one with the same specifications) is used to charge the KIPR Link. The KIPR Link should only be charged when under adult supervision and should not be left on charge unattended. When off, the KIPR Link should reach full charge within 90 minutes. Charging with the wrong charger may damage the KIPR Link and will void the warranty.



KIPR Link



USB cable
(type A-microB)



USB Camera



AC Power Adapter

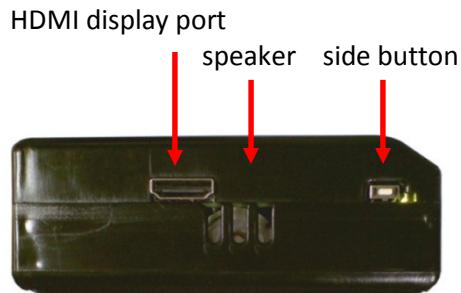
13.5v 1000mA
REGULATED (switching)

KIPR Link Features

Color touch screen (320x240 dpi),
WiFi, 2000 mAh LiPo battery,
800mhz ARMv5te CPU, FPGA,
imbedded Linux



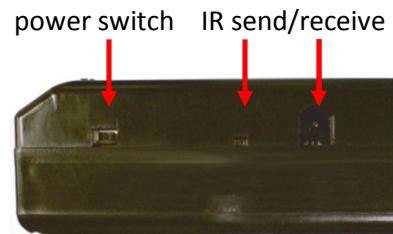
8 analog ports, 8 digital
I/O ports, 4 motor ports,
4 servo ports



Top View



Side View



Side View



Mounting holes have
Lego Technic compatible
spacing

Bottom View

TTL serial
micro USB
(for computer connection)



Back View

2. Quick Start

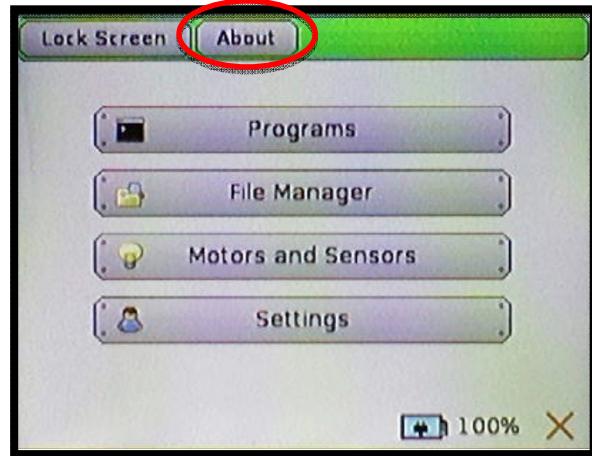
Turning On Your KIPR Link

Plug your AC power adapter into the wall and into the back of the KIPR Link. You will see a green charger present light and a colored charge status light at the power input. Slide the power switch to the ON position to boot the KIPR Link. The KIPR Link will take 40-45 seconds to boot into the user interface. When the KIPR Link is fully booted you will see a screen similar to the one below.

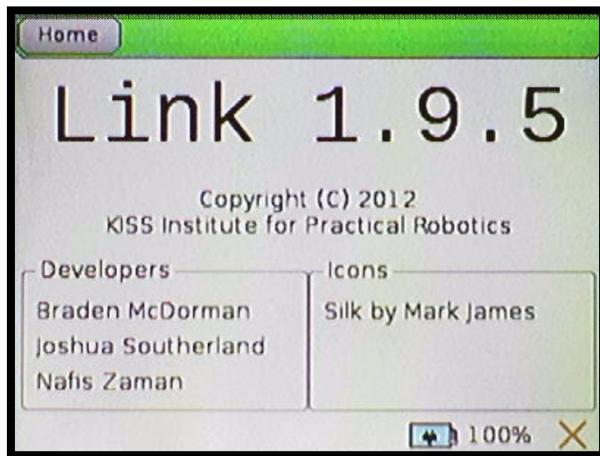
Checking the Firmware Version on Your KIPR Link

The Link is controlled by a software component called the KIPR Link Firmware. The web page at <http://www.kipr.org/kiss-platform-link-firmware> specifies the most recent release, its download link, and instructions for installing it from a USB flash drive.

To check the version of the software currently running on your KIPR Link, boot your KIPR Link and press the *About* button at the top of the home screen.



If the firmware version is out of date, to update your KIPR Link to the latest release go to the above website, download the zipped firmware image file to your computer, then follow the instructions for updating firmware, which are given on the web site and are in the Appendices of this manual.



Installing the KISS Platform on Your Computer - the KISS IDE

Mac (OS X 10.4 and higher - 64 bit processors)

The current version of the platform for Mac systems may be downloaded from

<http://www.kipr.org//kiss-platform-mac-os-x>.

First, **install the most recent release of Xcode for your version of OS X**. You can find the Xcode installer on your OS X install disks, or download it from the Apple Developer Connection at <http://developer.apple.com> (you must register for a free account to login and download the software).

If you are using OS X 10.8 (Mountain Lion), you will have make a system adjustment to allow the KISS IDE to run as an unverified app. Directions from Apple Support for doing this are on the web site <http://support.apple.com/kb/HT5290>.

Next, double click on KISS-x.x.x.dmg file to mount the disk image. **Copy the KISS-C folder in the disk image to the Applications folder on your Mac.** You need to keep the KISS-C application and the library folders in the same KISS-C folder (programs you write can be kept wherever you wish).

There is no need to install a USB driver; appropriate drivers come with OS X.

Windows (XP, Vista, or 7)

The current version of the platform for Windows systems may be downloaded from

<http://www.kipr.org//kiss-platform-windows>.

If you are using Windows XP, **connect the KIPR Link to the computer** using the included USB cable and turn on the Link. For Windows Vista and Windows 7, it is not necessary connect the Link to your computer.

Double click on KISS-C installer(KISS-x.x.x.exe). That will open the KISS-C Installation Wizard. Click Next to begin the installation process. Windows 8 will require you to make some setting changes first as outlined on the web site <http://www.kipr.org//kiss-platform-windows>.

On the *Choose Components* screen, if you haven't done an earlier install that included the driver, make sure that the Link driver is selected as well as KISS-C, then click *Next* to choose an install location. It is recommended that you install KISS-C in the default folder, i.e., the Program Files folder. **Click *Install* to begin installing KISS-C on your computer.**

At the point the driver gets installed you will be prompted that the driver is unsigned. This is normal, so agree to the *install anyway* option. In Windows XP, you may be prompted to install the driver, in which case click *Next* and then click *Next* again (search for driver) for Windows XP to find and install the driver.

KISS-C will be added to your program menu. A KISS-C shortcut will be placed on your desktop.

Windows 8

DO NOT PLUG THE KIPR LINK INTO THE COMPUTER UNTIL INSTRUCTED TO DO SO!

The current version of the KISS platform for Windows systems may be downloaded from
<http://www.kipr.org//kiss-platform-windows>.

Follow the following steps in order to install KISS-C on Windows 8:

1. Boot into Windows 8 normally
2. Move mouse into top right corner and click on the search icon
 - a. Search for "Settings"
 - b. Click on *Settings*
 - c. Search for "General"
 - d. Click on *General Settings*
3. Choose *Advanced Startup*
 - a. Click on *Restart Now*
4. Click on *Troubleshoot*
5. Click on *Advanced Options*
6. Click on *Startup Settings*
7. Click on *Restart*
8. Choose option 7
9. Log in to Windows 8
- 10. Plug your KIPR Link into your computer via USB cable**
11. Turn on your KIPR Link
12. Install the KISS IDE by double clicking the downloaded KISS-C installer and proceeding in the same manner for earlier versions of Windows
13. Search for "Device Manager"
 - a. You should see "Gadget Serial v2.4"
 - b. Right click on *Gadget Serial v2.4* and choose *Update Driver Software*
 - c. Then choose to search automatically
 - d. A low numbered COM port is good (anything less than 100 should work fine)

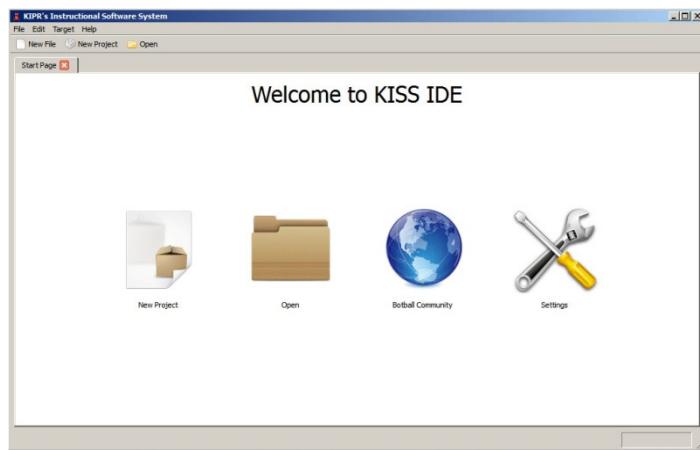
Downloading and Running a Program for Your KIPR Link

1. Connect the KIPR Link to your Computer

Use the supplied USB cable to connect the KIPR Link to your computer and turn it on.

2. Launch the KISS IDE

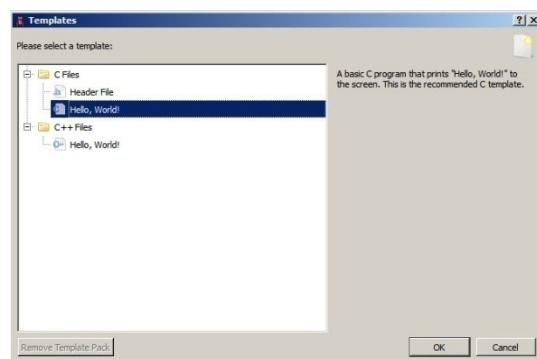
Click on the KISS IDE icon to launch the KISS IDE and bring up its opening screen.



3. Create a New Program

The template selection screen will appear.

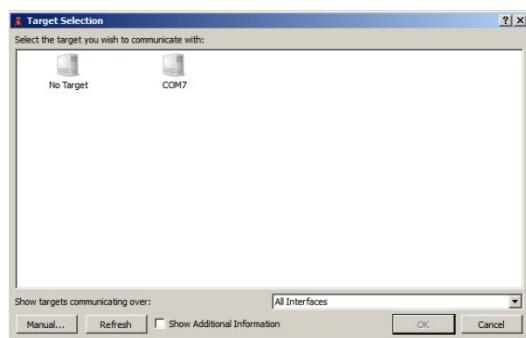
Click on the C "Hello, World!"Template to create a new program to use for testing.



4. Select the Communications Port

The Target Selection dialog will appear.

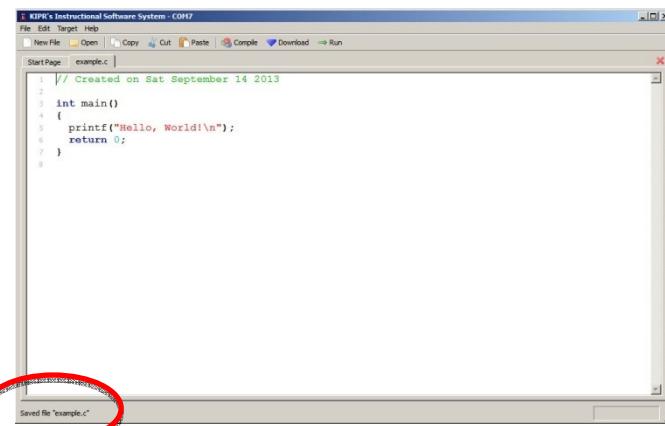
Select the communications port on your computer for the KIPR Link (for Windows, it will normally be the highest numbered port).



5. Edit/Save/Download the Program File

The template appears as a tabbed entry on the IDE where it may be edited, saved, compiled, and downloaded to the KIPR Link.

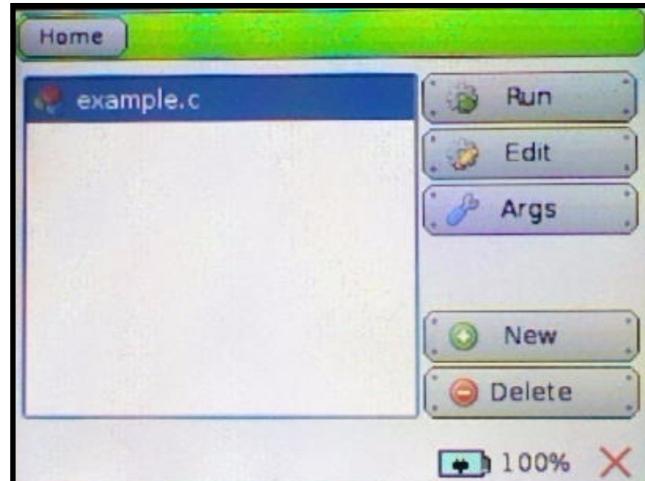
Use File..Save As.. to save the program to disk. *Compile* or *Download* will automatically save the program, asking for a file name if the program hasn't been saved before. *Compile* tests whether or not a program will compile. *Download* sends the program file to the KIPR Link without checking it for errors. The result of the



action is given in the lower left corner of the window. Download the program to verify this works OK.

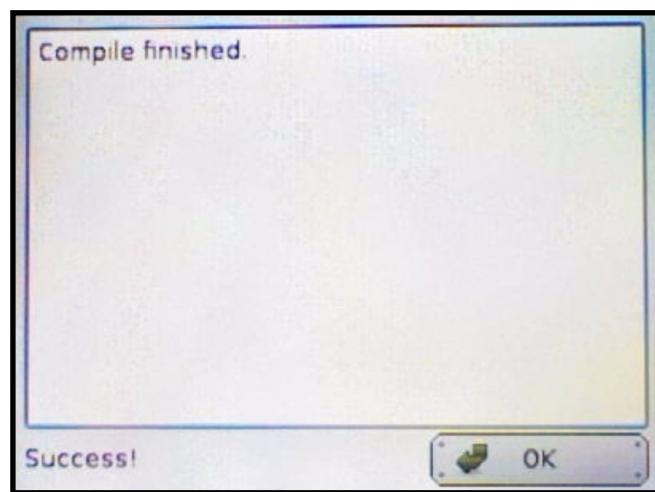
6. Locate the Program File on the KIPR Link

From the *Home* screen on the KIPR Link, press the *Programs* button. Your program will appear under the same file name you selected for it in the IDE. Press on the file name to highlight it.



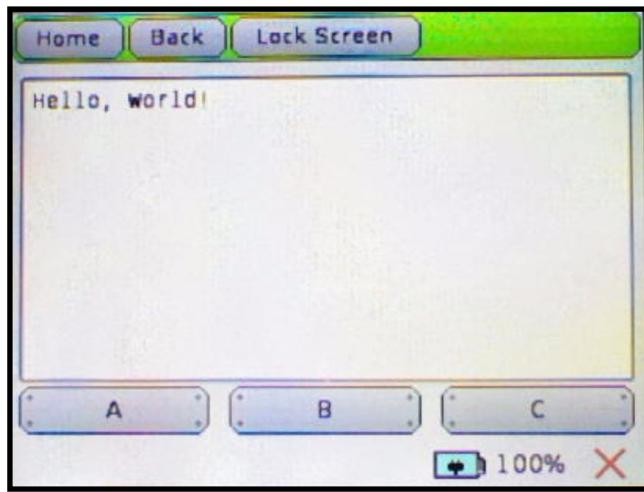
7. Compile and Run the Program on the KIPR Link

Press the *Run* button and the program will compile, bringing up the compiler report screen. If the compilation was successful, the program is now ready to run. Otherwise, any errors produced by compilation will be reported here. Press *OK* to run the program, or return to step 6 and *Run* will now execute the program without recompiling it.



8. Observe Results on the KIPR Link Display

From the compiler report screen (if the program compiled error free), pressing *OK* will bring up the console where program output is displayed. If you go back to step 6 and run the program again, it will run this time without re-compiling and take you directly to the console.



If you have successfully completed these steps, then CONGRATULATIONS! You have verified that your KIPR Link and KISS IDE are ready to use for developing your own programs.

The next section of this guide illustrates how to use the KIPR Link library functions and the KIPR Link user interface with KIPR motors and sensors.

It is followed by a section that describes how to configure the camera for use with the KIPR Link vision library functions.

A trouble shooting guide is included that covers common problems users have been known to encounter in trying to use the KIPR Link.

The appendices provide information of interest to some, but not all users, including a description of how to update the KIPR Link Firmware and listings of the special program libraries provided for using the KIPR Link with KIPR motors and sensors, the USB camera, and the iRobot Create module (available from the KIPR online store at <http://botballstore.org/>).

3. Programming for the KIPR Link

Using the C Programming Language with the KIPR Link and the KISS IDE

The C programming language is the most widely used systems language. It has been adapted for use with the KIPR Link via specialized function libraries. The C compiler used with the KIPR Link and the KISS IDE is the ANSI C compiler included with Linux. There are other compiler environments for the KIPR Link as well which are not discussed here (check with KIPR for the development status of these). The *Help .. Documentation* tab for the KISS IDE provides an on-line guide for using C with the KISS IDE and the KIPR Link. It includes documentation for the specialized function libraries and images of the KIPR motors and sensors available for the KIPR Link from the KIPR online store. The KISS IDE environment also includes a simulator for the KIPR Link, not discussed here.

For more information on ANSI C programming pick up an ANSI C programming guide from your local book store. A recommended beginner's book is "Absolute Beginner's Guide to C" (2nd Edition) by Greg Perry. The KISS IDE on-line guide also incorporates a short C tutorial, best suited for those already familiar with another programming language or who are rusty with C.

To write a C program for the KIPR Link, set up your KIPR Link as described in the Quick Start Guide, proceeding to the "Edit/Save/Download the Program File" step. Use one of the C templates. Your program will have a file tab on the KISS IDE interface, which will have a tab labeled "*** Untitled**" until you save it, after which the label on the tab will be the file name under which you are saving it (for a C template, the KISS IDE automatically saves with a ".c" suffix, which the system uses as a signal that the C compiler is the correct one for the file). Each time you repeat this process, you will get a new file tab. You can use the tabs to move among several programs, perhaps for an action like copy/paste. The file that is visible is the one being edited.

```
// Created on Sat September 14 2013
int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

Edit and add to the template to suit your needs. When you think your program is ready for testing, press the KISS IDE *Compile* button to check for compiler errors. If you have not already saved a version of your program, you will be prompted to name the file and choose a directory to save it in.

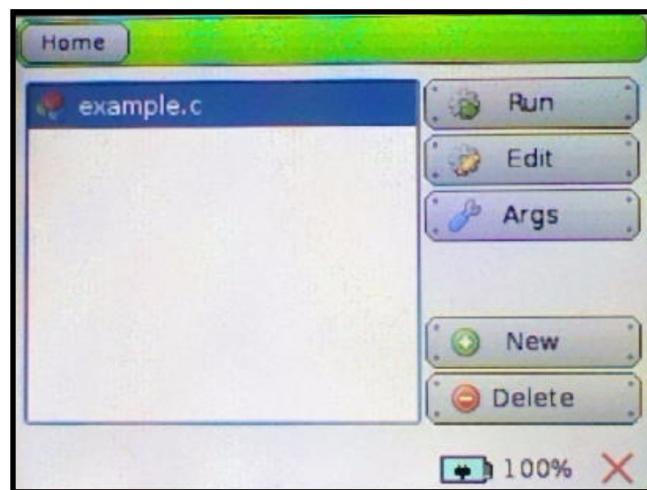
Subsequently, the program will be automatically saved to this file when you compile to check for compiler errors.

Any errors the compiler finds in your program will be displayed in a panel at the bottom of the KISS IDE window, starting with the first error encountered in the program. If you double click on an error, you will be taken to the line listed for the error. What needs to be fixed will either be on that line or on one above it (e.g., you left off a ";" on an earlier line). Since a program error tends to cause errors later in the program, don't be surprised if correcting the first error listed is all that is needed.

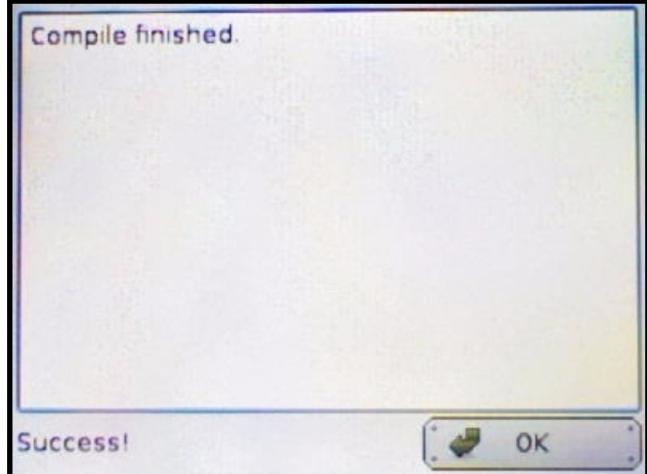
To download your program to the KIPR Link, click the KISS IDE *Download* button. Since your program will need to be compiled on the KIPR Link, this action only copies your program to the Programs directory on the KIPR Link.

The instructions for locating and running your program on the KIPR Link are given in the Quick Start section above, repeated here for your convenience:

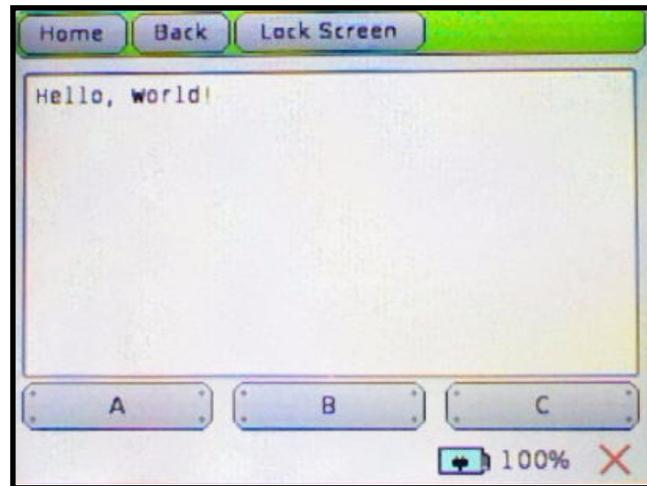
1. To locate your program file, from the *Home* screen on the KIPR Link, press the *Programs* button. Your program will appear under the same file name you selected for it in the IDE. Press on the file name to highlight it.



2. To compile and run your program on the KIPR Link press the *Run* button and the program will compile, bringing up the compiler report screen. If the compilation was successful, the program is ready to run. Otherwise, any errors produced by compilation will be reported here. Press *OK* to run the program, or return to step 6 and *Run* will now execute the program without recompiling it.



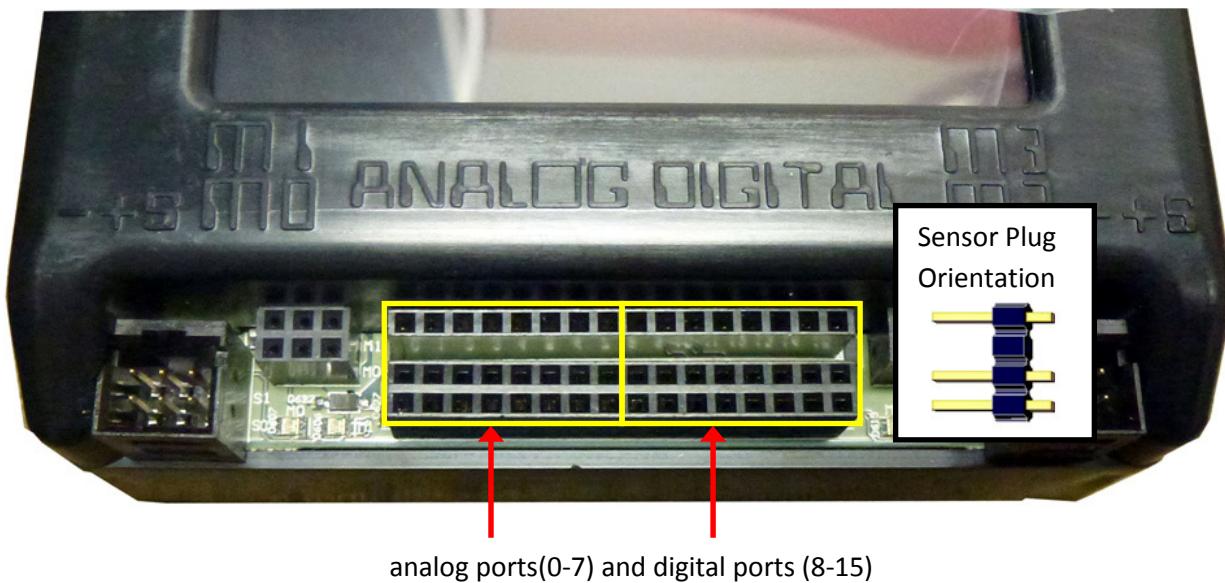
3. To observe programs results (if the program compiled error free) pressing *OK* will bring up the console where program output is displayed. If you go back to step 1 and run the program again, it will run without re-compiling and take you directly to the console.



KIPR Link Function Libraries

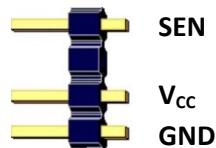
Function libraries are provided in the KIPR Link Firmware which enable programmers to take advantage of the features of the KIPR Link. Some of the more common functions are presented in the following sections, and the complete list can be found in the Appendix.

Using Sensors



analog ports(0-7) and digital ports (8-15)

Any sensors purchased from KIPR will work with the KIPR Link. They are “keyed” so that there is only one orientation for which all of the pins will be in holes.



Digital sensors typically only have two wires and are wired such that when the sensor is triggered the **SEN** and **GND** lines complete a circuit. Analog sensors can have two or three wires. For an analog sensor the resistance between the **SEN** and **GND** lines will vary. The third wire is connected to **V_{cc}** to power the sensor if the sensor requires a power source in order to operate (e.g., a reflectance sensor has an IR emitter which has to be powered to work).

The voltage on the KIPR Link from **V_{cc}** to **GND** is set to +5V for both digital and analog ports. The voltage range between **SEN** and **GND** is also +5V for both digital and analog ports. The +5V setting is established by internal jumper settings, and opening the case to change them will void your warranty. It is suggested that you first contact KIPR Technical Support if you think this is something your application requires. Instructions for accessing the jumpers are in the appendix as are specifications for creating your own sensors.

KIPR Link Library Functions for Sensors

The two most basic library functions for using sensors with the KIPR Link are the **analog10** and **digital** functions.

analog10(<port#>)

Analog sensors produce a varying voltage value as resistance between **SEN** and **GND** varies. **analog10** returns the analog value of the port (a value in the range 0-1023). Analog ports are numbered 0-7. Light sensors and range sensors are examples of sensors you would use in analog ports. The following example is for a light sensor plugged into analog port 3:

```
printf("Light sensor reading is %d\n", analog10(3));
```

digital(<port#>)

Digital sensors operate like a switch, effectively producing a resistance value between **SEN** and **GND** of either none (switch closed) or ∞ (switch open). The **digital** function returns 0 if the switch is open (∞ resistance) and 1 if the switch is closed (no resistance). The digital ports on the KIPR Link are numbered 8-15. Touch sensors are typical digital sensors, commonly used for bumpers or limit switches. The following example is for a button (touch) sensor plugged into port 8:

```
if (digital(8)==1)
    printf("button is being pressed\n");
else
    printf("button is not being pressed\n");
```

The C language prototypes for these two functions are:

```
int analog10(int port_no);
int digital(int port_no);
```

Sensors usually require a pull up resistor to work properly. Each KIPR Link sensor port has a software selectable pull up resistor, enabled on boot by the KIPR Link Firmware as the default setting. A sensor such as the "ET" distance sensor available from KIPR may already have a built in pull up resistor. In this case, the pull up resistor for the port used by the sensor needs to be disabled, which can be done by using the `set_analog_pullup` function provided in the KIPR Link Library. For example, for an ET sensor plugged into analog port 4

```
set_analog_pullup (4,0);
```

will disable the pull up resistor for port 4. The terminology "floating" is used to describe a port without a pull up resistor and "pull up" if its pull up resistor is enabled. Since "pull up" is the default setting, any data read by the `analog10` function from an ET sensor in port 4 won't be meaningful until the pull up resistor for port 4 is disabled.

There are additional functions for sensors in the KIPR Link library, described in the appendices. These include button functions for the KIPR Link's side button, and for the virtual A, B, C, X, Y, Z buttons on the program input/output console screen.

Using Servo Motors



Servos plug into the servo ports on the front of the KIPR Link. The arrows used above represent the most common coloring for servo cables (ground is black, positive is red, and signal is yellow), but yours may differ. If it does check the pin out to make sure that it is compatible with the KIPR Link before plugging it in (servos sold through the KIPR online store will always be compatible). The servo ports operate at 7.2V. Since a servo motor's function is to move to a position and hold it, the motor will continue to draw significant power to maintain position. As the KIPR Link battery is drained, the

available power may fall below the threshold where the servo will function properly. Servos that mysteriously begin misbehaving are usually symptomatic of a battery in need of recharge.

KIPR Link Library Functions for Servo Motors

Servo motor ports should be disabled when not in use to limit their impact on system power resources. On boot, servo ports are disabled by default. There are 4 servo ports, labeled 0, 1, 2, 3. The KIPR Link Library includes functions for managing servo ports as well as functions for operating servo motors. The basic functions are:

`enable_servo(<servo_port#>)`

Enables power for the specified servo port. When the port is enabled, the servo motor plugged into it will move to the last position set for the port, by default 1024, the center point of servo travel range.

`disable_servo(<servo_port#>)`

Disables power for the specified servo port. This is useful when you want to conserve battery life. Your servo motor will be at the mercy of any external forces and will not hold its position when the port is disabled.

`set_servo_position(<port#>, <position>)`

Moves the servo motor plugged into the specified port to the specified position. Only position values between 0 and 2047 are meaningful, and a value within 200 of an end point of the position range may be unattainable, depending on the servo motor being used. The servo will immediately move to position, unless impeded by external force or the position is unattainable, in which case it will continue to draw its maximum power level in trying to reach the position. If this function is called before the port is enabled, when the port is enabled the servo motor will move to this position.

Example use of these functions for a servo motor plugged into servo port 2:

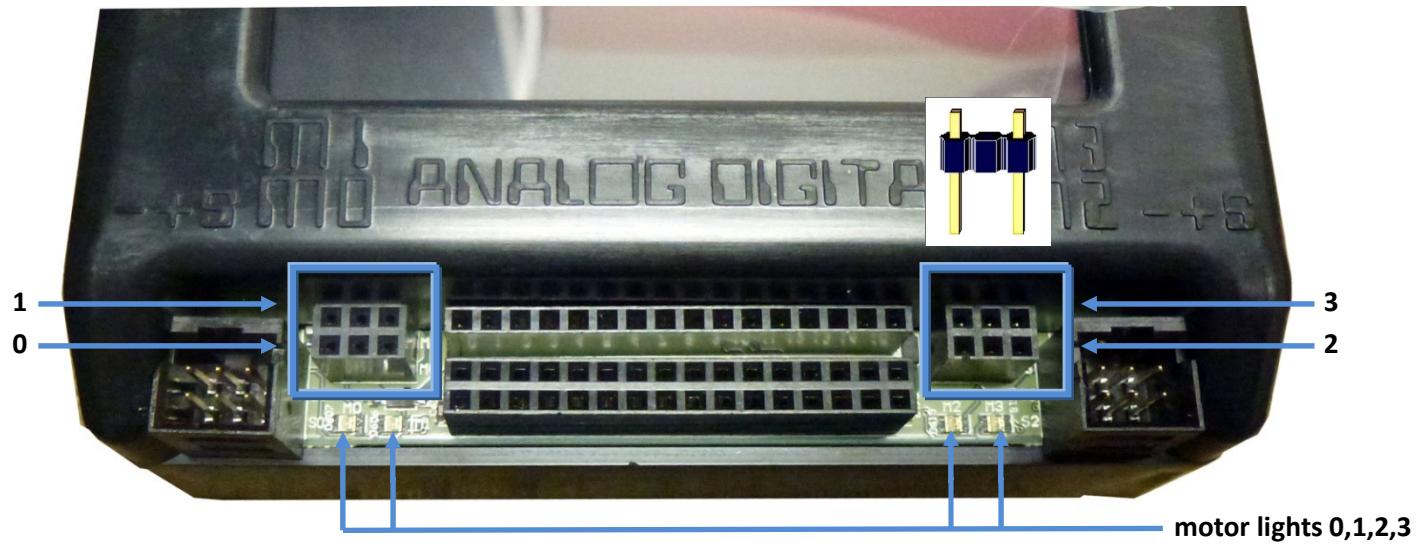
```
set_servo_position(2, 670); // set a position for the servo port
enable_servo(2); // power the port and move the servo
... do something else ...
disable_servo(2); // servo no longer needed
```

The C language prototypes for these three functions are:

```
void enable_servo(int servo);
void disable_servo(int servo);
int digital(int servo, int position);
```

There are a few additional, less commonly used functions for servo motors in the KIPR Link library, described in the appendices.

Using DC Drive Motors



The DC drive motors sold through the KIPR online store use a two pronged plug and can be plugged into the KIPR Link motor ports in either direction. The motor port center hole is not employed. The effect of reversing the plug is to reverse the motor's – + polarity, which simply reverses motor direction.

To check motor polarity manually rotate the motor to produce back EMF to power the LED light for the port. It will glow green for one direction and red for the other. By convention green means forward, so if the light glows red when you rotate the motor in what you want to correspond to forward, reverse the plug.

When your program runs a function that instructs the motor to move the motor will turn. The motor ports operate with a max current draw of 1A per motor port, where motor speed is regulated by using Pulse Width Modulation (PWM) to adjust the average power supplied to the motor. Voltage at full power is 5V.

Each pair of ports (0 and 1, 2 and 3) is controlled by an H-bridge chip, so if you are going to be using close to 1A per motor (full throttle), plug your motors into ports controlled by different H-bridges (e.g., 0 and 2) to limit heat build-up in the chip. This will reduce the incidence of failure for your H-bridge chips over time and will extend their operational life.

Note that only the outside two pin positions for a motor plug connector are used. The KIPR Link Library motor functions set motor port polarity – +, lighting the green LED, if the direction is to be forward. If the motor direct is to be in reverse, motor port polarity is set + –, lighting the red LED. See the appendix for the specifications on creating your own motor plugs.

KIPR Link Library Functions for DC Motors

motor(<motor#>, <power>)

Turns on a motor at a scaled PWM percentage, which will continue until another motor command is issued. Power levels range from 100 (full forward) to -100 (full backward). PWM stands for "pulse-width-modulation" and is a more effective way to control DC motor power than using something like a variable resistor.

mav(<motor#>, <velocity>)

Move At Velocity moves a motor at a specified velocity until another motor command is issued. The value of the velocity is scaled to range from -1000 to 1000 ticks per second. **mav** (and a number of other KIPR Link Library motor commands) uses a more complex motor control scheme than the static PWM percentage employed by the **motor** command. To obtain the specified velocity, **mav** employs PID (proportional-integral-derivative) gain values and a dynamic measure of the BEMF ("back electro-motive force") produced by the motor to dynamically adjust the PWM percentage being applied so that the motor continues moving at the specified velocity. The BEMF value is periodically refreshed by briefly suspending PWM so that the motor's back EMF can be read by the KIPR Link.

mrp(<motor#>, <velocity>, <ticks>)

Move Relative Position moves a motor at a specified velocity from its current position to the current position plus the number of ticks specified. Velocities range from 0 to 1000 ticks per second. Like **mav**, **mrp** uses PID motor control. **mrp** stops executing when the new position is reached or when another motor command is issued. The final motor position may be slightly off due to motor coasting once power is no longer being applied.

bmd(<motor#>)

If the motor is currently executing a positioning command (such as **mrp**), **bmd** (Block Motor Done) does not return until the positioning command finishes. Note that if the motor is stalled by some external cause, the effect of this command will be to hang program execution until the motor is freed.

ao()

All Off turns off power for all motor ports.

off(<motor#>)

Turns off power for the specified motor port.

There are a number of additional motor functions in the KIPR Link library, described in the appendices.

The C language prototypes for these five functions are:

```
void motor(int m,int p);
void mav(int m,int vel);
void mrp(int m,int vel,int ticks);
void bmd(int m);
void ao();
void off(int m);
```

Other Functions Commonly Used With the KIPR Link

msleep(<milliseconds>)

KIPR Link Library function that pauses the currently executing thread, resuming execution of the thread after a time equal to or slightly greater than the number of milliseconds specified.

printf(<string>, ...)

Standard C library function for formatting output directed to standard out (which is the program output console screen for the KIPR Link). If the string contains % codes then the arguments after the initial string (the ... portion of the function's arguments) will be printed in place of the % codes using the format specified by the matching % code. See the Appendix or consult a C language reference for a listing of % codes and how they are interpreted for formatting data.

beep()

KIPR Link Library function that creates a short beep sound.

The C language prototypes for these three functions are:

```
void msleep(int msecs);
int printf(char s[], ...);
    (... represents 0 or more values corresponding to % codes present in s)
void beep();
```

Example use of these functions (along with motor functions, motor in motor port 2):

```
int t=1000, m=2;
mrp(m, 500, 500); // move motor at 500 ticks per second
printf("Running m%d %d mseconds\n",m,p); // use %d for int values
msleep(t); // suspend the current thread for p milliseconds
off(m); // turn off motor
beep(); // beep when thread resumes executing
```

In 1 second the motor should move 500 ticks. In contrast to using **bmd**, **msleep** forces the motor off after 1 second even if it is short of its target position. The **printf** output will be

Running m2 1000 mseconds

For more information on KIPR Link Library functions, consult the on-line documentation under the KISS IDE Help tab as well as the appendices to this Guide.

4. KIPR Link Vision System

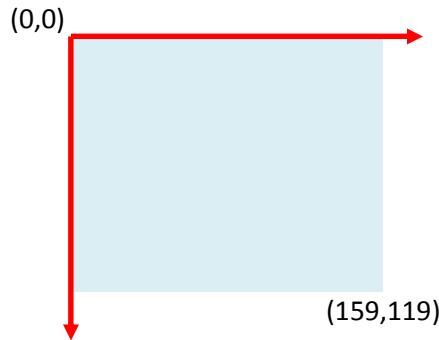
About Color Vision Tracking and QR Codes

The KIPR Link Vision System incorporates color vision tracking and QR code identification. A USB web camera is used to provide images to the KIPR Link at a rate dependent on lighting conditions but exceeding 6 frames per second. Using the KIPR Link interface, an arbitrary number of camera configurations containing channels for color vision tracking and/or QR code identification can be defined.

For color vision tracking, images are processed by the KIPR Link to identify "blobs" matching the color specification for each color channel in a camera configuration. A blob is a set of contiguous pixels in the image matching the color specification for the channel.

For each color channel in a selected configuration, the values to be used to identify which pixels in an image match the desired color for the channel are interactively selected from a color spectrum chart to provide a color specification for the channel. Live feed from the camera simplifies the process of determining how much of the spectrum is needed to produce blobs matching the color (e.g., a particular part of the spectrum might include all pixels that are "reddish" in color for a channel to be used for identifying red objects). The spectrum values for the channel are retained with the configuration until the configuration is deleted.

The camera image size is 160 x 120. The upper left corner has coordinates (0,0) and the lower right has coordinates (159,119). The camera image displayed on the KIPR Link is slightly smaller than the actual image size.



KIPR Link Vision Library functions are used to select a configuration and obtain information about the color blobs being identified by its channels, such as bounding box coordinates and pixel density.

In addition to channels for color tracking, a configuration can have channels for identifying QR (Quick Response) codes. A QR code is essentially a 2-dimensional bar code for compactly representing text data. KIPR Link Vision Library functions are provided for decoding any QR code in the image.

Setting Up KIPR Link Color Tracking Channels

The USB camera plugs into one of the USB (type A) ports on the back of the KIPR Link. Unplugging the camera while it is being accessed will usually freeze the system, requiring a reboot.

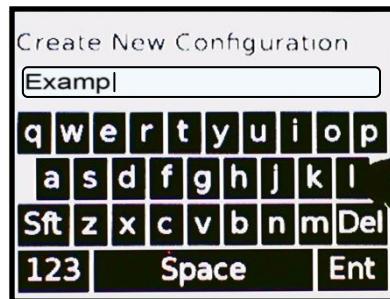
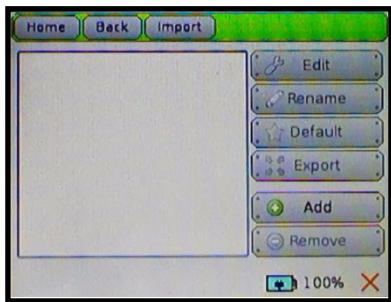


The vision system is accessed from the *Settings* button on the KIPR Link Home screen. Pressing the *Settings* button brings up the KIPR Link Settings screen.

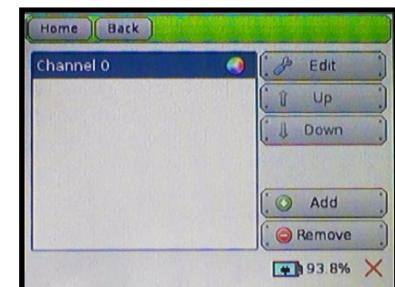
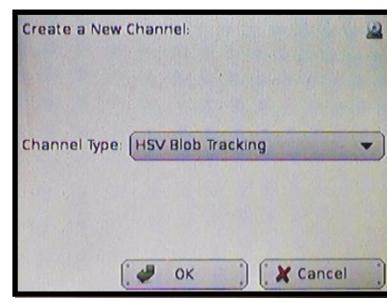
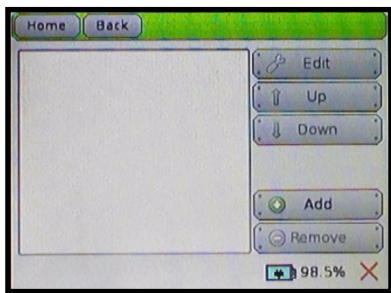


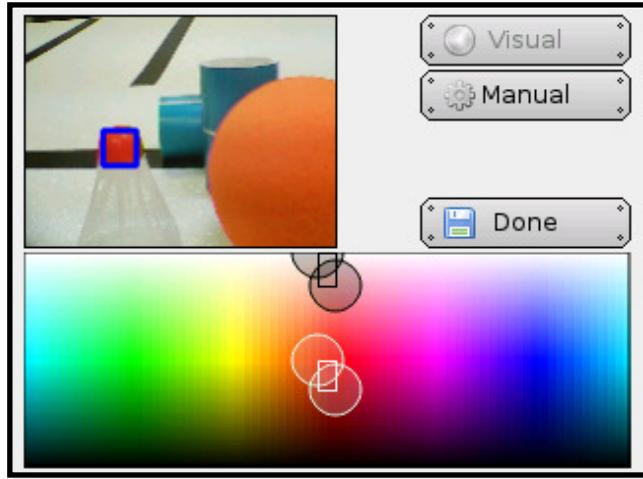
To set up a color tracking channel press the *Channels* button to bring up the channels specification screen.

To create a new camera configuration, press the *Add* button and enter a configuration name. This will bring the channels specification screen back up, with the new configuration added to the list of defined configurations. The configuration marked with a star is the current default configuration. Highlight the camera configuration you want to work on and press the *Edit* button.

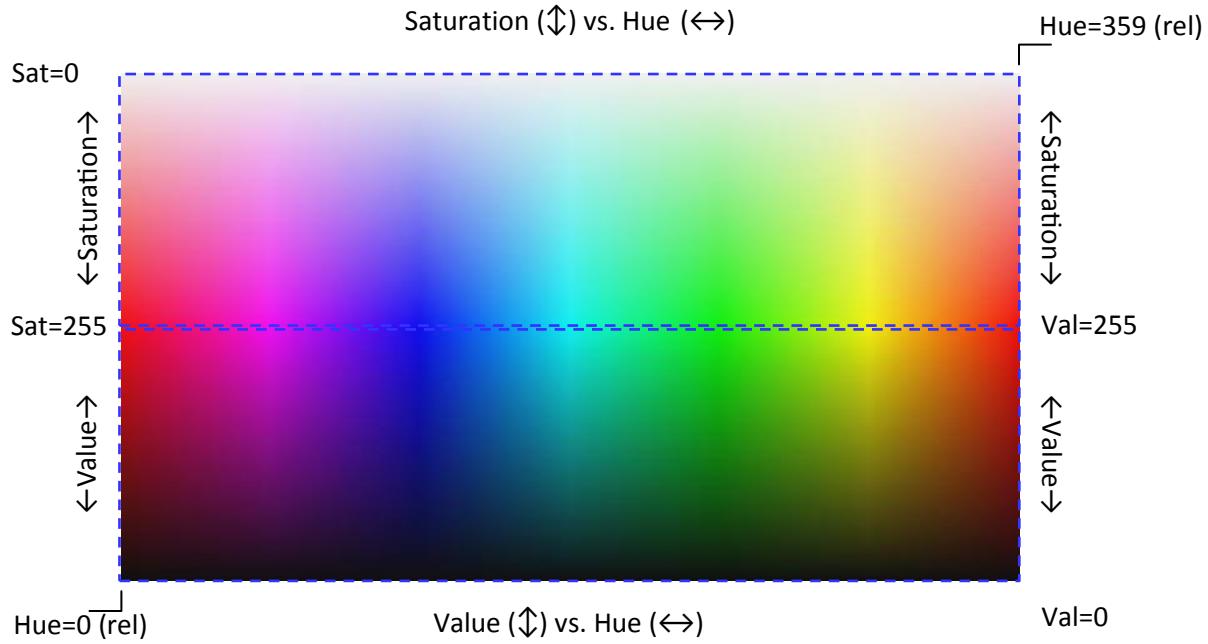


Any channel already defined for the configuration will be listed. Press the *Add* button to add a channel to the configuration and select *HSV Blob Tracking* to make this a color tracking channel. Highlight the channel to be worked on and press *Edit* to bring up its HSV specification screen. After Channel 0, any additional channels added to the configuration will be numbered 1, 2, ...



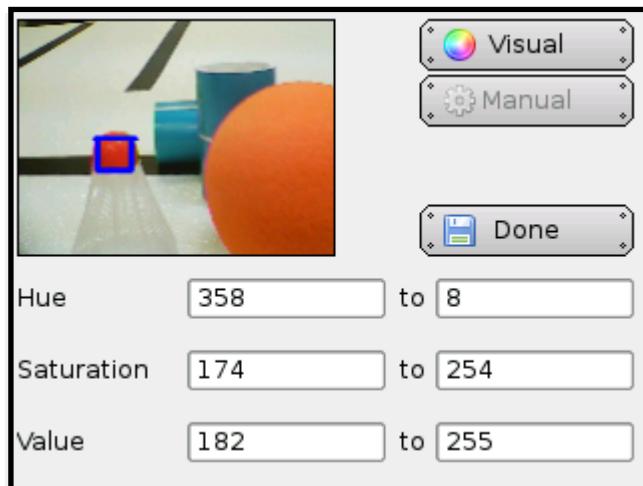


Color definition is based on the HSV color model (Hue-Saturation-Value) where the HSV color selection panel is organized using two stacked panels across the Hue spectrum:



A rough approximation for the desired HSV values can be achieved graphically by dragging the circled corners of the pixel selection boxes. For example, if the channel is to be used to locate red blobs, then dragging the selection corners to the red part of the spectrum will begin producing blue bounding boxes around reddish colored blobs in the image. When the selection corner is released, to facilitate further adjustment, the KIPR Link will shift the spectrum to have the current color spectrum selection in the center (the Hue spectrum cycles back to 0 after 359, so the Hue range may be from higher to lower as well as lower to higher). Every pixel whose value matches the values within the selection is a color match. In the image above, note that there is a blue bounding box in the upper left quadrant of the camera image identifying a red blob.

After obtaining a rough approximation for a color's range of HSV values, the accuracy can be adjusted further by selecting the *Manual* option, which will replace the color spectrum with boxes containing the current HSV values for the channel. These can be adjusted by using the numeric keypad the interface brings up when a value is pressed on the KIPR Link's touch display.

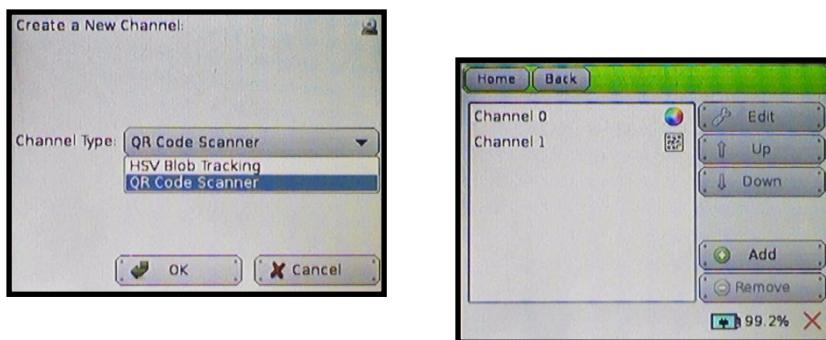


Manual adjustment is particularly useful when the color channel needs to discriminate among objects having related, but distinctive colors, such as the orange and red objects in the above image.

Pressing the *Done* button reverts the display to the channel listing for the configuration, from where more channels can be added.

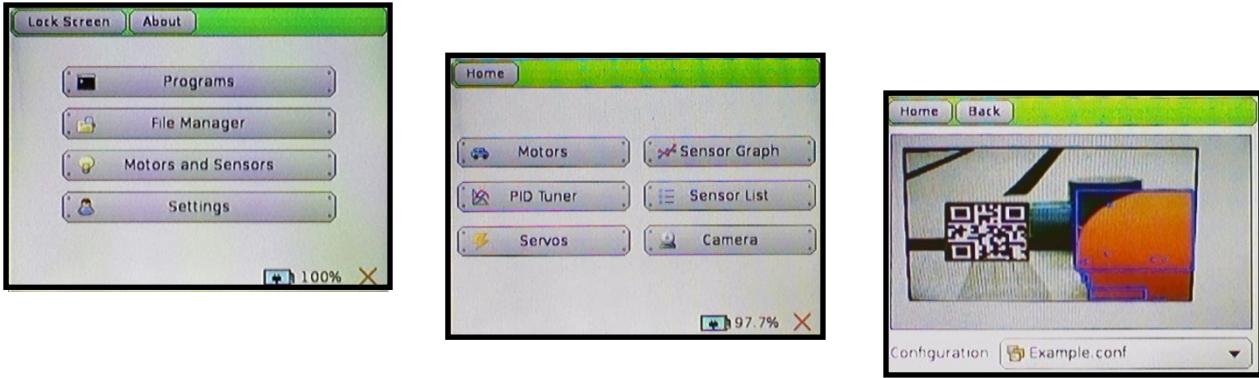
Setting Up a KIPR Link QR Scanning Channel

When a channel is added to a configuration, its type is specified either for *HSV Blob Tracking* or *QR Code Scanner*. There is no additional specification required for QR scanning. The channel type is identified by a different icon for QR codes than for color tracking. A sample program is given below that illustrates how a scanned QR code is decoded.

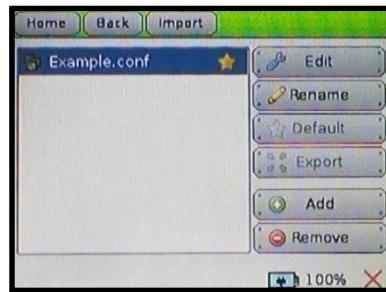


Verifying Channel Behavior

From the main screen, if the *Motors and Sensors* button is pressed, one of the options is for the *Camera*. If pressed this will show the bounding boxes for the channels in the default configuration, with a different bounding box color for each channel (QR codes boxed in black, Channel 0 blobs boxed in blue, etc).



The default configuration is marked with a "star" and can be changed to any other configuration listed by highlighting the configuration and pressing the *Default* button.



KIPR Link Vision Library Functions

These are commonly used vision functions, for a complete list see the appendix.

`camera_open(<resolution>)`

Opens the camera using the currently loaded configuration. For most users, the default configuration (as specified by *settings .. channels .. default*) is used. For another configuration to be employed, it has to be loaded (see `camera_load_config`). A resolution of `LOW_RES`, `MED_RES`, or `HIGH_RES` must be specified. For most purposes, `LOW_RES` will suffice and has the least impact on system performance.

`camera_close()`

Closes the current camera instance and clears its presence from system resources.

camera_update()

Once the camera has been instantiated using camera_open, each call to camera_update will retrieve and process the current camera image. Always call this function before using any other camera functions so they reference the current data.

get_object_count(<channel>)

Returns the number of objects for a channel as determined by the most recent camera_update(). Objects for a channel are ranked numerically with 0 being the largest in area.

get_object_bbox(<channel>,<number>)

Returns a compound data type *rectangle* with component values determined by the most recent camera_update. The rectangle data type has 4 components:

- ulx - the upper left x coordinate of the object's bounding box
- uly - the upper left y coordinate of the object's bounding box
- width - the width of the object's bounding box
- height - the height of the object's bounding box

get_object_center(<channel>,<number>)

Returns a compound data type point2 with component values determined by the most recent camera_update. The point 2 data type has 2 components:

- x - the x-coordinate of the center
- y - the y-coordinate of the center

For a QR code channel the following two functions are used for QR decoding:

get_object_data(<channel>,<number>)

Returns a pointer to the sequence of character data for the QR code. If the channel is invalid, or there is no object, or there is no data, 0 is returned. The data is not guaranteed to be null terminated, but can be accessed using array notation; for example,

get_object_data(0, 0) [0], get_object_data(0, 0) [1], etc.

The pointer returned by get_object_data will be valid until camera_update is called again, at which point get_object_data will return a new pointer.

get_object_data_length(<channel>,<number>)

Returns the number of characters associated with the QR code on a QR channel. If the channel is invalid, or there is no object, or there is no data, 0 is returned.

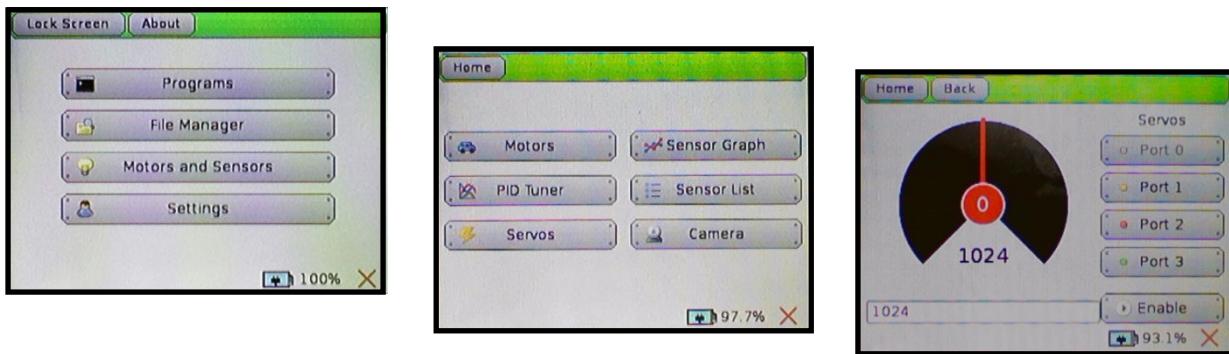
Sample color tracking program controlling a servo motor

This sample program is a demo for using the camera on the KIPR Link to control a servo motor. If you don't have a servo motor, skip to the next example. Servo motors come with "horns" designed to fit on the motor shaft. Choose one that will serve as a pointing device and attach it to your servo. The action of this program is to turn the servo in response to an object on color Channel 0, in effect rotating the servo to continue to point at the object as it is moved back and forth in front of the camera.

Set Up

Attach the camera to your KIPR Link and plug your servo into servo port 0. You need to set the color model on Channel 0 to track an object you can move in front of the camera (usually the more saturated with color the better). The servo needs to be pointed so that when at its midpoint position (1024) it points at the center of the camera's field of view.

Motors can be directly manipulated from the KIPR Link interface, which for servos provides a means for determining position settings to use in a program. In this case, we only need to orient the servo. From the Home Screen press the *Motors and Sensors* button to bring up the device selection screen from which if you select *Servos* you will get the servo test screen. Press *Port 0*, enter 1024, and press *Enable* to position the servo.



Press *Back* to return to the device selection screen (first press if you want to turn the servo off). This time press *Camera* to get the camera view on the screen so you can align the servo to be pointing at the center of the camera's field of view. For a better effect, you might want to attach a pointer to the servo horn (of a different color than for your color model).



You are now ready to run the program below. Reminder: Use the KISS IDE *Compile* button to test the program before downloading and running it on your KIPR Link.

Code

Note: For improved readability as the program executes, the Link specific function **display_printf** is used in place of **printf**, since **printf** will scroll the screen once it has reached the last row of the display.

display_printf(<col>, <row>, <string>, ...)

Performs a standard **printf** starting at screen location **col**, **row**, limited to columns 0 through 41 and rows 0 through 9 (fewer rows if extra buttons are turned on). The effect is a print in place to the screen. Use of '\n' in the string will distort the outcome. Excess text for any row is truncated.

```
/* For a servo plugged into port 0 and initially centered on the
camera's field of vision, this program rotates the servo to keep it
pointing towards the largest object for color channel 0 as the object is
moved about */
int main() {
    int offset, x, y;
    enable_servo(0);          // enable servo
    camera_open(LOW_RES);     // activate camera
    camera_update();          // get most recent camera image and process it
    while(side_button() == 0) {
        x = get_object_center(0,0).x; // get image center x data
        y = get_object_center(0,0).y; // and y data
        if(get_object_count(0) > 0) { // there is a blob
            display_printf(0,1,"Center of largest blob: (%d,%d)",x,y);
            offset=5*(x-80); // amount to deviate servo from center
            set_servo_position(0,1024+offset);
        }
        else
            display_printf(0,1,"No object in sight");
        msleep(200);           // don't rush print statement update
        camera_update();       // get new image data before repeating
    }
    disable_servos();
    camera_close();
    printf("All done\n");
}
```

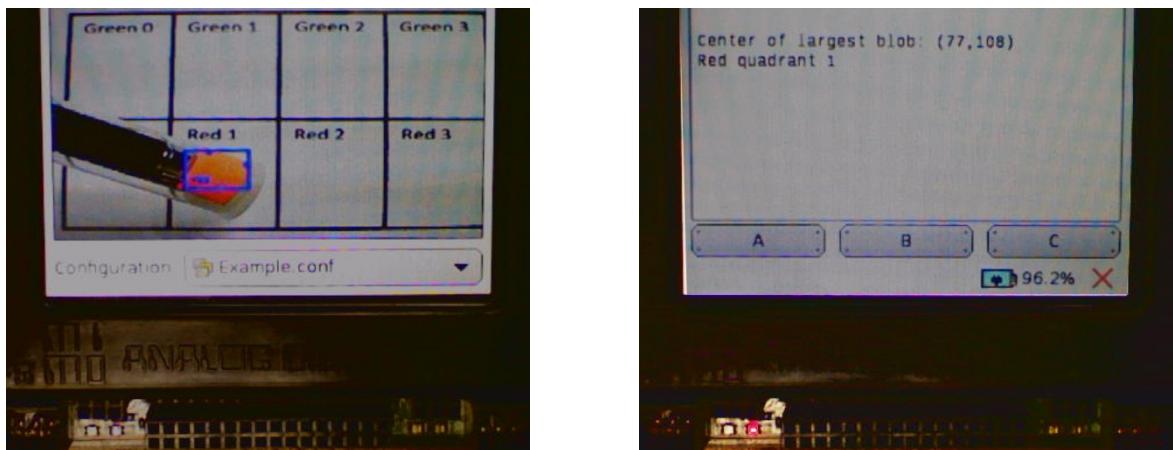
Sample color tracking program controlling motor lights

This sample program is a demo that tracks an object on color channel 0 and lights up the motor ports that correspond to the object's location. Each motor port corresponds to one-fourth of the screen. If the object is on the lower half of the screen a red motor light will come on. In on the upper half a green motor light will come on.

Set Up

Attach the camera to the KIPR Link. You need to set the color model on channel 0 to track an object you can move in front of the camera (the more saturated the color, the better). For best results point the camera horizontally at a grid marked off on a piece of paper so you can see the accuracy of the behavior. The center of the camera's field of vision needs to be aligned with the center of the grid. Finally, test and download the program to the KIPR Link.

| | | | |
|---------|---------|---------|---------|
| Green 0 | Green 1 | Green 2 | Green 3 |
| Red 0 | Red 1 | Red 2 | Red 3 |



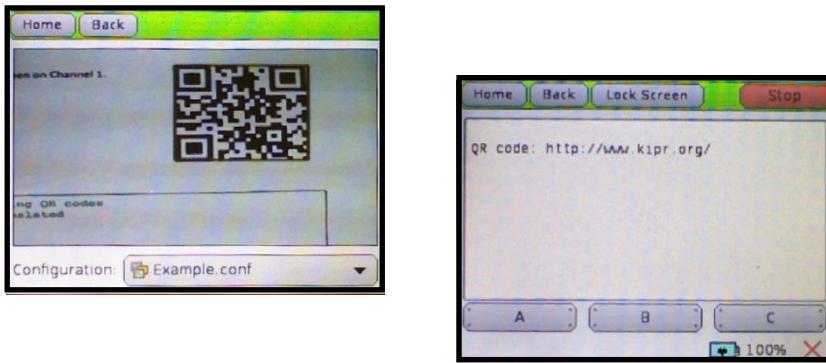
Code

```
/* For this program, point the camera at an 8-quadrant grid that fills
the KIPR Link screen. Each motor light corresponds to an area of the grid
in the pattern
    G G G G
    R R R R
i.e. if the center of the largest object on channel 0 is over a particular
grid location, that motor light will be turned on */
int main()
{
    int x, y, quad, xMax = 160, yMax = 120;
    camera_open(LOW_RES); // activate camera
    camera_update(); // get most recent camera image and process it
    while (side_button() == 0) {
        x = get_object_center(0,0).x; // get image center x data
        y = get_object_center(0,0).y; // and y data
        if (get_object_count(0) > 0) { // there is a blob in view
            display_printf(0,1,"Center of largest blob: (%d,%d) ",x,y);
            ao(); // turn off all motor lights
            // determine the horizontal quadrant to be lighted
            if (x < xMax/4) {
                quad = 0;
            } else if (x >= xMax/4 && x < xMax/2) {
                quad = 1;
            } else if (x >= xMax/2 && x < 3*xMax/4) {
                quad = 2;
            } else if (x >= 3*xMax/4) {
                quad = 3;
            }
            if (y < yMax/2) { // green row
                fd(quad); // quad's green motor light on
                display_printf(0,2,"Green quadrant %d ", quad);
            } else { // red row
                bk(quad); // quad's red motor light on
                display_printf(0,2,"Red quadrant %d ", quad);
            }
        } else {
            display_printf(0,1,"No object in sight ");
            ao(); // lights off
        }
        msleep(200); // don't rush print statement update
        camera_update(); // get new image data before repeating
    }
    ao(); // clean up
    camera_close();
}
```

Sample program for decoding a QR code

This sample program is a demo that uses QR scanning Channel 1 for detecting and decoding a QR code when one is present in the camera field of view.

From the *Motors and Sensors .. Camera* screen,
aim your camera at this code and verify it is being seen on Channel 1.



Now compile and run the sample program to find out what the QR data is.

Code

```
// Assume channel 1 is for identifying QR codes
// If a QR code is found, it is translated
int main() {
    int i, length;
    camera_open(LOW_RES); // activate camera
    camera_update(); // get most recent camera image and process it
    while(side_button()==0) {
        if (get_object_count(1) > 0) { // there is a QR code in view
            display_printf(0,1,"QR code: ");
            length = get_object_data_length(1,0);
            // print QR code letter by letter until end of data
            for(i=0; i < length; i++) {
                display_printf(9+i,1,"%c", get_object_data(1,0)[i]);
            }
        } else {
            display_printf(0,1,"No QR code detected ");
        }
        msleep(200); // don't rush print statement update
        camera_update(); // get new image data before repeating
    }
}
```

5. Troubleshooting

If at any point you need additional help, are uncomfortable completing a troubleshooting step, or there is a problem you cannot resolve, call KIPR Technical Support at **(405) 579-4609** between 9AM and 5PM Central Standard Time, or email support@kipr.org.

| Problem | Solution |
|---|--|
| My Link does not turn on. | Plug your KIPR Link into the charger. At the plug point a red LED should be visible indicating the charger is correctly plugged in. A green charge status LED should also come on indicating the KIPR Link battery is charging. It will go off when the KIPR Link battery is fully charged, which should take no more than 90 minutes. When on charge your KIPR Link should boot into the home screen. |
| All of the motor ports have red lights on when I boot my Link. | This indicates your KIPR Link Firmware has been corrupted. Reload the firmware. See Appendix. |
| My Link comes on but only loads to the splash screen and keeps rebooting continuously. | This behavior indicates the KIPR Link Firmware has been corrupted. Reload the most current version of the Firmware. See Appendix. |
| My Link's touch screen is unresponsive, or it is difficult to press buttons in the user interface. | You need to recalibrate the touch screen. From the home screen of the KIPR Link. To fix this, press the <i>Settings</i> button and then press <i>Calibrate</i> . Press each of the four corners (a stylus may be helpful). If the <i>Home</i> button is still hard to get to respond, recalibrate again and don't press so far into the corners. |
| My Link will not compile code that successfully compiled on my computer. | This indicates your KIPR Link Firmware is not in sync with the KISS IDE. Reload the firmware. You may also need to get the most current version of the IDE. See Appendix. |
| My camera is displaying a frozen, black, or garbled image in the vision menu. | This happens when the KIPR Link loses the connection to the camera (such as when you unplug it and plug it back in). Reboot the KIPR Link. |

| Problem | Solution |
|--|---|
| My KIPR Link is not recognized by my Mac running OS 10.4, but is recognized by my other computer running OS 10.5 (or newer) or Windows. | Your KIPR Link Firmware is out of date. Reload the firmware. See Appendix. See Appendix. |
| My KIPR Link is not recognized by my Mac running OS 10.5 (or newer). | Turn on your KIPR Link. Check to make sure that the KIPR Link is listed in the System Profiler. If the KIPR Link is not listed there, check to make sure your USB cable is plugged in and functioning. If you are still having issues try a different USB port. |
| My KIPR Link is not recognized by my PC running Windows XP. | Turn on your KIPR Link. Under <i>Control Panel .. Device Manager</i> check to see if the KIPR Link is listed in the COM ports. It may appear as "Gadget Serial v2.4". If there is a device warning, the system had issues installing the driver, in which case try reinstalling the driver by clicking on the port, or try reinstalling from the KISS IDE installer. If the device isn't present in the COM port listing, check to make sure your USB cable is plugged in and functioning since powering on the KIPR Link should cause your PC to establish the USB connection. If you are still having issues try a different USB port. |
| My KIPR Link is not recognized by my PC running Windows 7 x64. | Turn on your KIPR Link and use <i>Device Manager</i> as outlined above to see if there is a COM port listing for "KIPR Link" or for "Gadget Serial". If present with a COM port number greater than 12, you need to purge excess COM ports*, otherwise, make sure you are giving the KISS IDE enough time to find the COM port, since as largest it is the last checked. If there is no such COM port listing, Gadget Serial should be listed under <i>Other Devices</i> , and if not, try reinstalling the driver from the KISS IDE installer. Once detected in Device Manager right click <i>Gadget Serial</i> and select <i>Update Driver Software</i> . If you have internet access, "Search automatically" should fix the issue. If you opt to "Browse my computer", point update to the c:Windows/KIPRLinkDriver folder. If a <i>KIPRLinkDriver</i> folder is not visible, under the Control Panel select <i>Folder Options .. View</i> and activate <i>Show hidden files, folders, and drives</i> . Now you should be able to see a c:Windows/KIPRLinkDriver folder so you can go back to Device Manager to try the "Browse" option. |

* Under Windows, excess, hidden COM ports may be present, either from serial devices not currently in use or from serial devices present during system power off without an operating system shut down. Device Manager can be set up to show hidden COM ports so they can be selectively removed:

Step 1: Under *Programs..Accessories* right click *Command Prompt* and select *Run as Administrator*.

Step 2: Enter

set devmgr_show_nonpresent_devices=1

Step 3: Enter

start devmgmt.msc

to bring up Device Manager.

Step 4: In Device Manager, under *View* select *Show Hidden Devices*.

Step 5: Expand the COM port listing, right click each excess COM port and select *Uninstall* to purge it.

The KISS IDE has been verified to run on all of the listed systems, but variations among system configurations may in some instances prevent a communications link from being established until minor adjustments are made. If you find yourself unable to get any of the above suggestions to work, please call KIPR Technical Support for assistance.

6. Appendices

Updating the KIPR Link Firmware

The Firmware for the KIPR Link is an image file which on boot provides a user interface and other system services for the KIPR Link and its Linux-based operating system. Firmware releases are obtained from <http://files.kipr.org/link>. Installation instructions and the current "official" release may be obtained via the web site <http://www.kipr.org/kiss-platform-link-firmware>.

Equipment Needed: KIPR Link, KIPR Link AC Adapter, USB Flash Drive (preferably, a high quality one)

Step 1: Download the current release of the KIPR Link Firmware. Regardless of release number, the downloaded firmware image is in the file named **kovan_update.img.gz**. Do not unzip this file, the KIPR Link will take care of that in installing it.

Step 2: Copy the **kovan_update.img.gz** file to the root (top level) of your USB flash drive.

Step 3: Eject (unmount) the USB drive from your computer and wait a second or so for the system to give you the go ahead before removing it. To eject a USB flash drive on a Mac (OSX) right click on the USB drive icon and choose Eject. For Windows machines, under *Computer* right click on the USB drive icon and choose Eject. This procedure protects the USB drive from being damaged or the image file from being corrupted if the copy action has not yet finished.

Step 4: Unplug any motors, servos, or sensors from your KIPR Link and plug in the AC adapter. Power off the KIPR Link, then plug the USB drive into one of the two USB ports on the back of the controller.

Step 5: While holding down the side button on the opposite side from the KIPR Link power switch, turn on the KIPR Link.

Step 6: Continue holding down the side button until the update bar appears, which signals that update has started. Once the update starts, you can let go of the side button.

Step 7: Wait for the update to complete, remove the USB drive, and reboot the device by switching the power off and back on again. Verify the firmware version. If not updated, the update procedure probably failed to start and the KIPR Link simply booted using the older firmware, in which case carefully repeat the process.

Step 8: From the home screen go into *settings..calibrate* to recalibrate the touch screen.

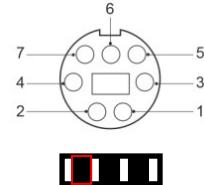
Controlling an iRobot Create with the KIPR Link

The KIPR Link can control an iRobot Create via its TTL serial connection. You will need to get a KIPR Link Create cable from the KIPR store (<https://botballstore.org>) or manufacture your own:

The iRobot Create uses a 7 pin Mini-Din connector where pin 3 is Create TTL serial in, pin 4 is Create TTL serial out, and either pin 6 or 7 is GND.

The KIPR Link uses a 3 port female header plug connected to the Mini-Din plug in the order Create serial out (red), GND, Create serial in.

Mini-Din pins 1 and 2 are for Create battery + (inactive except when the Create is on).



Once you have a KIPR Link/Create cable follow the steps below to communicate with your iRobot Create.

Step 1: Connect the Mini-Din end of the KIPR Link/Create cable to the Mini-Din connection port of the iRobot Create (it is hidden under a removable cover above the charge port on the Create).

Step 2: Connect the other end of the cable to the KIPR Link TTL serial port (the "official" cable has a keyed plug, but a header plug will also work if red orientation is observed). The power plug is optional, and if used, when the iRobot Create is turned on will "trickle" charge the KIPR Link battery. If you aren't using an "official" cable, note the orientation of the wiring (Create serial out, GND, Create serial in) has to be correct to establish a connection to the iRobot Create.



Step 3: Write a program for the KIPR Link that communicates with the iRobot Create and download it to the KIPR Link.

Step 4: Set the iRobot Create on the ground.

Step 5: Power on the iRobot Create (this is the step most commonly left off!)

Step 6: Run your program on the KIPR Link.

The KIPR Link serial interface is normally used for downloading programs from the KISS IDE to the KIPR Link via USB. This same interface is also used for communicating with the iRobot Create. The **create_connect** library function redirects the serial interface to the TTL serial connection instead of the USB connection. It must be run before any of the functions in the KIPR Link Create Library will work. The function **create_disconnect** is used to revert the serial connection to USB.

iRobot Create KIPR Link Library Functions

There are a large number of functions in the KIPR Link Library for controlling an iRobot Create module. These functions provide an interface between a program on the KIPR Link and the Create Open Interface (http://www.irobot.com/filelibrary/pdfs/hrd/create/Create%20Open%20Interface_v2.pdf). The functions which update sensor data, and the connection functions return the requested information if they are successful and return a number greater than 100,000 if there is some error. If an error is returned, the error message is 100,000 + <Create-Serial-Interface-Packet-Number>. For example, a code of 100,007 indicates an error when requesting bumper or wheel drop sensor status.

The functions for moving the iRobot Create are non-blocking, excepting scripts which once started play to completion. Movement commands (with the exception of **create_stop**) are sent to the iRobot Create only if they represent a change from the previous movement command. For this reason, movement commands may be placed in tight loops without concern of overwhelming the serial connection. The iRobot Create's trajectory will continue until a different movement command is given.

The iRobot Create may also be used to play MIDI music. Up to sixteen 16 note songs may be loaded into the iRobot Create. See the Create Open Interface manual for details on note and duration codes.

These are the commonly used functions, for a complete list, see the appendices.

create_connect()

Connects the KIPR Link serial interface to the iRobot Create. Call this function first. By default the iRobot Create will be in “safe mode”.

create_disconnect()

Disconnects the KIPR Link from the iRobot Create by reverting the serial interface to USB. Call this function at the end of the program, otherwise the KIPR Link will remain connected to the iRobot Create and the KISS IDE will not be able to communicate with your KIPR Link. This function also shuts off the Create motors. Power cycling your KIPR link will also revert the serial interface to USB.

create_stop()

Stops the drive wheels.

create_drive(<speed>,<radius>)

Drives in an arc at a set speed. Speed range is 20-500mm/s, radius is in mm.

create_drive_direct(<left motor speed>,<right motor speed>)

Specifies individual left and right motor speeds from 20-500 mm/sec (plus or minus)

As the iRobot Create operates, it accumulates data on distance traveled and angle turned through. There are also "set_create" and "get_create" functions in the KIPR Link Create Library for initializing and obtaining these measures. The complete listing is in the appendix. Two commonly used set_create and get_create pairings are :

set_create_distance(value)

Initializes the distance accumulation (in mm) to the specified value.

get_create_distance()

Returns the distance (in mm) that the center of the iRobot Create has traveled since the distance accumulation was last initialized.

set_create_total_angle(value)

Initializes the total turn angle accumulation (in degrees) to the specified value.

get_create_distance()

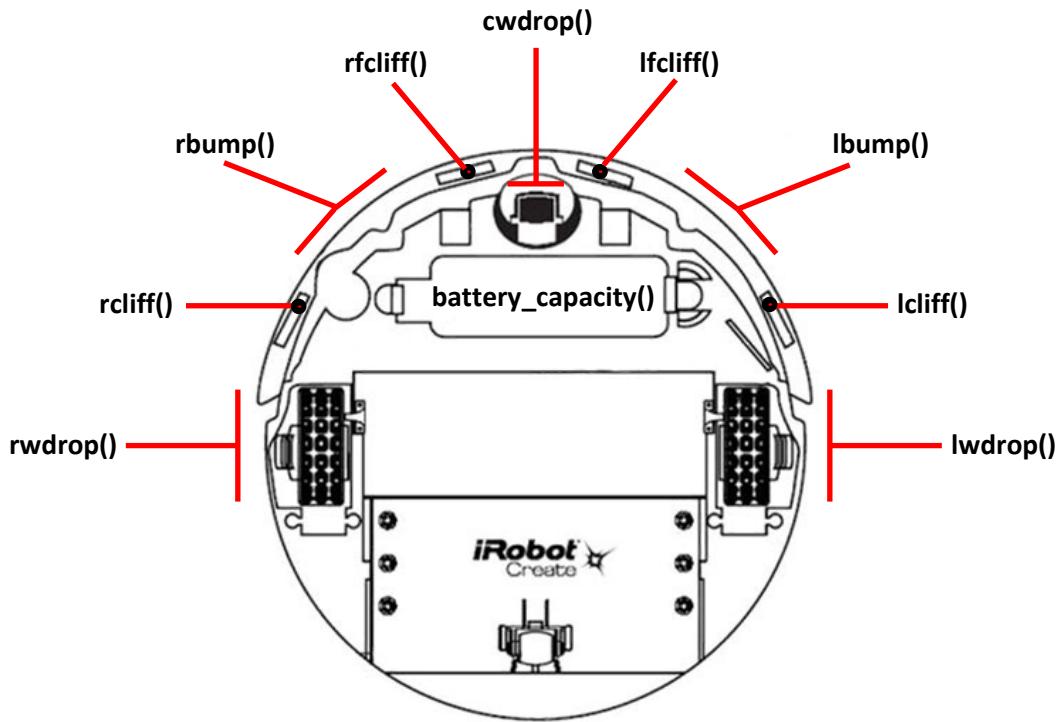
Returns the total angle (in degrees) that the iRobot Create has turned through since the total angle accumulation was last initialized.

"**get_create**" functions in the KIPR Link Create Library are also used for querying the iRobot Create regarding the status of its sensors; for example,

get_create_lbump()

Returns the value of the left bump sensor (pressed = 1, not pressed = 0)

The following diagram shows additional **get_create** suffixes for sensors whose status can be accessed via the KIPR Link Create Library.



The complete listing of **get_create** functions is in the KISS IDE documentation and in the appendices.

In using an iRobot Create module, keep in mind that the Create and the KIPR Link are independent entities communicating via a serial connection. If the connection is lost, the iRobot Create will continue on its own according to the last command it received. The relatively slow nature of the communications between the KIPR Link and iRobot Create may also cause sensor data to be missed unless your program is constructed to recognize that possibility (e.g, collect 3 readings at 0.1 second intervals and go with best 2).

Sample Program for Controlling an iRobot Create with the KIPR Link

This sample program is a demo to make the iRobot Create drive in a circle with a radius of 0.25 meters at a speed of 200 mm/sec for 10 seconds, then displaying the distance traveled and the angle covered.

Set Up

Fully charge your KIPR Link and iRobot Create. Connect the KIPR Link to the iRobot Create with the KIPR Link Communication cable. Place the KIPR Link into the cargo bay of the iRobot Create. Set the iRobot Create on the floor with an adequate area cleared in front of the iRobot Create. Note that if the iRobot Create detects a ledge with the cliff or wheel drop sensors (like being picked up), the program will stop and the iRobot Create will play a sad tone.

Code

```
/* This is a program to make the iRobot Create drive in a circle
with a radius of 0.25 meters at a speed of 200 mm/sec for 10 seconds,
the displaying the distance traveled around the circle and the angle
that the turn covered*/
int main()
{
    printf("connecting to Create\n");
    create_connect();
    set_create_distance(0);
    set_create_total_angle(0);
    create_drive(200, 250);
    msleep(10000);
    create_stop();
    printf("\nResults:\n");
    printf("  distance = %d mm\n", get_create_distance());
    printf("  angle = %d degrees\n", get_create_total_angle());
    printf("\ndisconnecting from Create\n");
    create_disconnect();
}
```

Writing an iRobot Create Script

The iRobot Create Open Interface is described in a guide made available via the web (http://www.irobot.com/filelibrary/pdfs/hrd/create/Create%20Open%20Interface_v2.pdf). Among other things, the guide describes the serial byte codes used to govern the behavior of the iRobot Create module.

The KIPR Link communicates with the iRobot Create via a (TTL) serial connection. Functions are included in the KIPR Link Library for the iRobot Create, and send serial byte code sequences to the Create over the serial connection, making it possible to operate the Create without having to reference the Open Interface guide. These sequences cover the large majority of actions users typically want to have performed by a Create module (e.g., drive forward at a given speed, determine how far the Create has travelled, etc). They also provide a means for the KIPR Link to directly control an iRobot Create.

The Open Interface also provides for scripts, where a script is a (limited) sequence of iRobot Create byte code commands ordered to perform some set of actions independent of external control. In contrast to other commands, a command to start a script disables serial communications until the script has finished.

The iRobot Create has several built in scripts, mostly to serve the needs of its cousin, the iRobot Roomba. The Open Interface provides byte code commands for running these. It also has a byte code command for loading a user defined script onto the iRobot Create along with a byte code command to start it running. The user defined script remains available until the iRobot Create is power cycled.

Unlike high level languages, scripts for the iRobot Create have no provision for flow of control commands such as if and while, but can use commands to wait for an elapsed time, or for a specified distance or angle to be reached, or for an event such as a bump (wait commands are not available except within scripts).

Memory for storing a user defined script is limited to 100 bytes.

The **create_write_byte** function in the KIPR Link Library is used for defining and sending a script (byte by byte) to the iRobot Create. The KIPR Link Library provides 3 commands for serial communications between the KIPR Link and an iRobot Create.

create_read_block(<data string>, <count>)

The Create sends the number of bytes specified into the data string.

create_write_byte (<byte>)

The KIPR Link send the specified byte to the iRobot Create.

create_clear_serial_buffer()

The internal serial buffer is emptied of any unaccessed send/receive data.

Example

The following program includes a function that uses `create_write_byte` to define and send a script to the iRobot Create byte by byte. The script is designed to move the Create a specified distance at a specified speed and illustrates the low level coding involved in defining a script.

Code

```
#define RUN_SCRIPT create_write_byte(153)
void make_drive_script(int speed, int dist) {
    create_write_byte(152); // specifies start of script definition
    create_write_byte(13); // remaining number of bytes in script
    create_write_byte(137); // drive command, speed & radius follow
    create_write_byte(speed >> 8); // send bits 8-15 of speed
    create_write_byte(speed); // and then send bits 0-7
    create_write_byte(128); // send hex 80 (X8000 specs drive straight)
    create_write_byte(0); // and then send hex 00 (no turn radius)
    create_write_byte(156); // wait for distance command, dist follows
    create_write_byte(dist >> 8); // send the 2 bytes
    create_write_byte(dist); // for distance in mm
    create_write_byte(137); // stop by dropping speed and radius to 0
    create_write_byte(0); // speed = 0
    create_write_byte(0);
    create_write_byte(0); // turn radius = 0
    create_write_byte(0);
    // end of script (15 bytes)
}
int main() {
    create_connect();
    set_create_distance(0);
    set_create_total_angle(0);
    make_drive_script(500, 500); // script to move 0.5m at 500 mm/sec
    msleep(500); // give serial connection some time
    RUN_SCRIPT;
    msleep(1500); // allow time for the script to finish (+ some extra)
    printf(" distance traveled = %d mm\n", get_create_distance());
    printf(" angle turned = %d degrees\n", get_create_total_angle());
    create_disconnect();
}
```

When an `int` in the range -32,768-32,767 is provided as the argument for `create_write_byte`, the value assigned to the byte is the low order byte (bits 0-7). To change this to bits 8-15, first right shift (`>>`) the `int` by 8 bits.

There are three major byte code commands used in the script:

1. Byte code 152

Specifies start of script definition, and must be immediately followed by a byte that gives the remaining number of bytes in the script (0-98).

2. Byte code 137

Byte codes 137 is a drive command for the iRobot Create, and must be immediately followed by 4 bytes representing two 16-bit two's complement integers representing speed and turn radius in mm, respectively. The high order byte (bits 8-15) of each number is sent first. For a positive speed and positive turn radius, the iRobot Create arcs to the left while driving forward. A turn radius of hex 8000 or 7FFF (32,767) is a special case to drive straight. There are also special cases for turning in place CW and CCW.

3. Byte code 156

Command to wait until the iRobot Create has traveled a specified distance, immediately followed by 2 bytes representing a 16-bit two's complement integer for distance in mm.

Byte code 153 is the command sent by the program that directs the iRobot Create to run the current user defined script.

The byte code 137 drive command does not provide for individual control of drive motors. Byte code 145 is a drive command for providing direct control, where the two 16-bit two's complement integers that immediately follow it specify right and left motor speeds, respectively.

The wait command used in the example is for distance traveled. There are 4 wait commands that can be used in scripts (but not otherwise):

1. wait for elapsed time (155)

The byte code is followed by 1 byte giving time from 0-255 tenths of a second, resolution of 15ms.

2. wait for distance traveled (156)

The byte code is followed by 2 bytes giving the distance (mm) as a 16-bit 2's complement integer.

3. wait for total angle rotated through (157)

The byte code is followed by 2 bytes giving the angle (mm) as a 16-bit 2's complement integer.

4. wait for event to occur (158)

The byte code is followed by a 1 byte event number (+ or minus, where + is for event occurrence, and - is for event non-occurrence). For example, event -9 is "no wall detected" (e.g., the event doesn't occur until a robot being sensed by the wall sensor has moved out of the way). To obtain a complete listing of the 22 wait events recognized by this command, consult the Create Open Interface.

The most typical use for scripts is to move a Create, so the above example can be used as a template for script development.

Here are few other ideas to consider regarding the use of scripts:

1. A script can be run indefinitely by concluding it with byte code 153 (the run script command).
2. A command sequence constructed as "drive, wait for event, stop" will continue immediately to the stop sequence if the event occurrence is already present, which has the effect of preventing the script from moving the Create.
3. A script can control the Create whether or not the KIPR Link that provided the script remains attached. For example, if a user script is started that immediately waits for a "trigger" event (such as a bump), the iRobot Create will remain idle until the event occurs, during which time the KIPR Link can be detached. In this manner, a script can be used as a means for operating an iRobot Create as an independent, if somewhat limited, robot entity.

Sample Program for Using KIPR Link Digital Output to Light an LED

By default the digital ports for the KIPR Link are configured for input. The KIPR Link Library function `set_digital_output` is used to configure digital port direction; for example,

`set_digital_output(9, 1);` configures port 9 for output and
`set_digital_output(9, 0);` configures it for input.

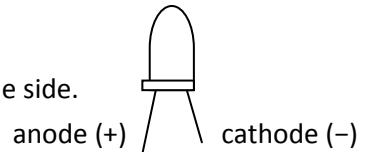
For a digital port configured for output, the KIPR Link Library function `set_digital_value` is used to set the port's output value to either 0 (low) or 1 (high); for example,

`set_digital_value(9, 1);` sets the output value for port 9 high.

If you have 5mm LED on hand, you can use the following program to operate it. An LED will "turn on" when voltage applied to its anode lead rises above a prescribed level (typically between 1.9 and 3.2V, depending on color). If too much current is passed through the LED, it will burn out (the typical spec is 20-30mA). For the KIPR Link, digital outputs on the **SEN** rail are sufficiently current limited to operate an LED without burning it out.

The LED's anode is normally identified by having the longer of the two leads.

Additionally, the flange at the base of the LED is normally flattened on the cathode side.



Setup

Insert the LED's anode lead into port 9's **SEN** line and the cathode lead into the a slot on the **GND** rail.

Code

```
/* This is a program to blink an LED plugged into digital port 9 */
int main()
{
    printf("LED in port 9\n");
    printf("Press side button to quit\n");
    set_digital_output(9, 1);
    while (side_button() == 0) {
        set_digital_value(9, 1);
        msleep(500);
        set_digital_value(9, 0);
        msleep(500);
    }
    set_digital_output(9, 0);
    printf("\ndone\n");
}
```

Sample Program Using a Thread for Monitoring a Sensor

In computer usage, the term thread is short for the phrase "thread of execution", and represents a sequence of instructions to be managed by the system as it schedules processing time among running processes. On a single processor machine, like the KIPR Link, the instructions running in separate threads appear to be operating in parallel. Each thread, once started, will continue until its process finishes or until it is forcibly terminated by another process using the **thread_destroy** function. Each active thread gets a small amount of time in turn until all of its statements have been executed or it is forcibly terminated. If a thread's process cannot complete all of its statements before its turn is over, it is paused temporarily for the next thread to get its share of time. This continues until all the active threads have gotten their slice of time and then it all repeats. Assuming the processor is fast enough, from the user's viewpoint it appears that all of the active processes are running in parallel.

Functions running in threads can communicate with one another by reading and modifying global variables. The global variables can be used as semaphores so that one process can signal another. Process IDs may also be stored in global variables so that one process can destroy another one's thread if that is necessary program logic (think in terms of a process that is in an indefinite loop monitoring sensors, so it would never finish otherwise).

There are four KIPR Link Library functions for controlling threads, **thread_create**, **thread_destroy**, **thread_start**, and **thread_wait**. A variable used for retaining a thread's system id must have the special data type, **thread**.

In the following example a thread is employed for running a function that monitors the side_button, raising a flag if it has been pressed. This way, a button press that occurs while the main function is in sleep mode is still captured. The flag is implemented using a global variable.

Code

```
int flag = 0; // global flag to signal when side button pressed
void chksens() {
    while (1) {
        if (side_button()) flag = 1; // if button pressed, flag it
        msleep(100); // keep checking side button (every 1/10th second)
    }
}
int main() {
    int cnt = 0;
    thread tid; // thread variable for holding thread id
    tid = thread_create(chksens); // create a thread for chksens
    thread_start(tid); // start chksens running in its thread
    while (flag == 0) { // thread is still running while main sleeps
        display_printf(1,1,"elapsed time %d      ",++cnt);
        msleep(1000);
    }
    thread_destroy(tid); // remove the thread
    printf("\ndone\n");
}
```

File I/O for a USB Flash Drive Plugged into the KIPR Link

Before the Linux operating system can access a file system, it has to "mount" the file system. When a USB flash drive is plugged into the KIPR Link, it is automatically mounted. When the USB flash drive is unplugged it is automatically unmounted. The C Library has a number of functions designed to access files located in mounted file systems. The library functions `fprintf` and `fscanf` respectively provide a straight forward means for writing formatted output to a file on a USB drive plugged into the KIPR Link, and for reading formatted data from a file on the USB drive. There are a number of file processing commands, including ones for accessing files byte by byte. For a full description of the range of functions available consult a standard C reference book.

To access a file, in addition to the file name, the directory "path" leading to the file has to be known. For the KIPR Link, the directory path to a mounted Flash drive in a USB port is

```
/kovan/media/sdal/
```

Files are accessed in C via a pointer of type FILE, which is defined in the system header file `<stdio.h>`. The pointer for a file is established when the file is "opened" for access. If the `fopen` function returns a NULL pointer, it indicates that either the file doesn't exist for the specified file path, or its file system hasn't been mounted (e.g., the USB drive has not been plugged in). Both cases are illustrated in the following program for a USB drive plugged into a KIPR Link. The example otherwise is a program designed to send data to a file, close the file, then reopen the file and retrieve the data to verify a successful write operation. If the file doesn't exist it is created. If it does exist, it is appended to. A user defined preprocessor macro (USB) is constructed to set the file path for the USB drive, illustrating how the preprocessor can be used to potentially simplify program code.

Example program using `fprintf` and `fscanf`:

```
#include <stdio.h> // make sure file I/O is defined
/* USB is a Macro defined to preface a file name with the directory path for
a mounted USB drive, turning the result into a character string */
#ifndef USB
/* An auxiliary macro is employed that converts its argument into a string
(by surrounding it with double quote marks) */
#define _STRINGIFY_(x) #x
/* the USB macro appends x to the path for the USB drive, then uses
_STRINGIFY_ to turn it into a string */
#define USB(x) _STRINGIFY_(/kovan/media/sdal/x)
#endif
int main() {
    FILE *f; // file pointer f (the FILE data type is in <stdio.h>
    // A file name "myUSBfile" for the USB drive is set up using macro USB
    char s[81], chkf[] = USB(myUSBfile); // file path as a string variable
    int x, data = 2;
    // try opening for read ("r") to see if the file exists
    if ((f = fopen(chkf, "r")) != NULL) {
        fclose(f); // file chkf already exists
        printf("Will be appending to USB %s\n", chkf);
    }
    // (file given by chkf is not open at this point)
```

```

// open to append ("a"), which also tests if the USB stick is plugged in
if ((f = fopen(chkf, "a")) == NULL) {
    printf("No USB stick detected\n");
    return -1; // exit the program
}
// file is now open for append; if it didn't exist it has been created
printf("Sending %s, %d\n", "Field ", data);
fprintf(f, "Field "); // use fprintf to send a text string to chkf
fprintf(f, "%d", data); // now send formatted numeric data using fprintf
fclose(f); // close the file to make sure the output is sent
// now read it back
f = fopen(chkf, "r"); // it exists since we just created it
fscanf(f, "%s %d", s, &x); // read the two data items from the file
fclose(f); // done with file, so close it
printf("Data read is %s: %d\n", s, x);
}

```

The USB drive can now be removed from the KIPR Link. A file named `myUSBfile` will now be present on the USB drive and can be accessed using a text editor to verify file contents.

Creating your own sensor

Tools Needed

- Soldering Iron
- Wire clipper
- Wire stripper
- Hot melt glue gun or heat source for heat shrink tubing
- Razor knife or sharp scissors

Supplies needed

- 1 x 4 male header, 0.1" (2.54mm) spacing (if all you have is a longer header row, you can use your wire clipper to clip off a 1 x 4)
- Small gauge stranded wire (28 AWG ribbon cable is preferred)
- Solder
- Sensor (3.3V is preferred, but 5V will work)
- Insulating material (like hot glue or heat shrink)

Method

The KIPR Link sensor interface employs standard 0.1" (2.54mm) female headers. There are three female header rows that make up the analog and digital sensor ports. The gap between Row 1 and Row 2 (as shown below) is 0.1" (2.54mm). Row 1 is the sensor input **SEN**, Row 2 is **V_{cc}** which is set at +5V, and Row 3 ground is **GND**. Adjusting jumper settings to alter rail voltage requires opening the case, voiding your warranty - if this is something you have questions about, call KIPR Technical Support.



The KIPR Link normally will work with either 3.3V or 5V sensors. For a 3.3V sensor a max value reading for **SEN** occurs when voltage reaches 3.3V or more (Row 1). You will need to read the data sheet for the sensor you are creating to determine how the sensor needs to be wired.

KIPR sensors use wire peeled from standard 28AWG ribbon cable, which because it is stranded is much more flexible than solid wire. Your sensor connection will require either a two wire cable or a three wire cable, depending on its specification.

KIPR sensor plugs are based on 1 x 4 male headers clipped from longer header rows, with one of the two middle pins pulled clear to provide a gap between Row 1 and Row 2.

To make the connection between sensor and plug, form leads by separating the wires a small amount on each end of the cable, then strip off enough insulation to be able to solder the leads to your sensor on one end and to the plug on the other. Be careful that you match up each connection from the sensor to its corresponding pin on the plug. Use a colored marker to identify the same outer wire on each end of the cable as an easy way to avoid a cross connection.

Once your cable is attached to your plug and your sensor, you need to insulate the leads to prevent any cross contact. KIPR sensors use hot melt glue for this purpose, which also stabilizes the connection and provides strain relief (to flatten the glue before it hardens, sandwich between two refrigerated pieces of smooth metal, then trim off excess glue using a razor knife or sharp scissors).

Before plugging your new sensor into your KIPR Link, use a multi meter to check for shorts and for continuity between each sensor lead and its corresponding plug lead.

Navigate to the sensors screen on the KIPR Link (*Motors and Sensors .. Sensor List*), and plug in your sensor. For the port plugged into, as the sensor operates, the value should flip between 0 and 1 for a digital sensor and vary between 0 and 1023 for an analog sensor.

Creating your own motor

Tools Needed

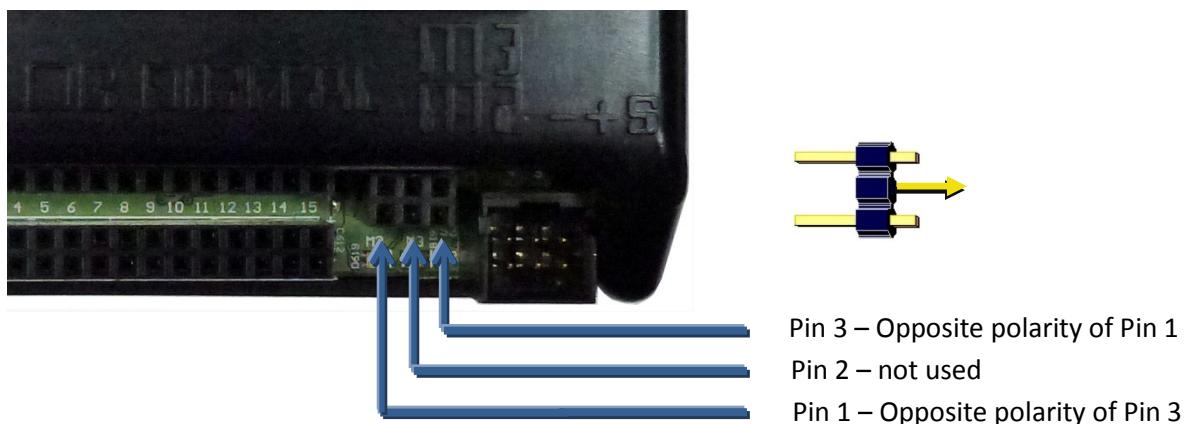
- Soldering Iron
- Wire clipper
- Wire stripper
- Hot melt glue gun or heat source for heat shrink tubing
- Razor knife or sharp scissors

Supplies needed

- 1 x 3 male header, 0.1" (2.54mm) spacing
- Small gauge stranded wire (28 AWG ribbon cable works best)
- Solder
- DC Motor (5V or more, with less than 1A current draw - max motor port power is 5V))
- Insulating material (like hot glue or heat shrink)

Method

As for sensors ports, the KIPR Link DC motor interface employs standard 0.1" (2.54mm) female headers. Each pair of motor ports is a dual 1 x 3 female header strip serviced by a dual H-bridge chip for PWM. The outside two positions (pins 1 and 3) provide the DC poles for driving a motor (analogous to using a battery). For Pin 1 negative (-) and Pin 3 positive (+) the green motor light is lit when the motor is powered. If the polarity is reversed (Pin 1 + and Pin 3 -) then the red motor light is the one lit. By convention, the polarity that lights green is "forward" and the one that lights red is "backward" (remember that if the connection is reversed, the motor simply runs in the opposite direction, so "forward" is a relative term). The middle pin is internally connected to pin 1 but it is best to not use it since the Pin 1/Pin 3 plug can't be plugged in wrong.



When choosing a motor, be aware that supply voltage is regulated by PWM (Pulse Width Modulation), which has a non-linear response curve in providing an effective voltage level that ranges from 0 to 5V. The BEMF PID (Back Electro Motive Force, Proportional Integral Derivative) control system for PWM

provides means for more precise motor control, but requires selecting PID values matching motor characteristics, which can be non-trivial to determine. Also note that the maximum current draw is 1A. A motor which exceeds this will probably cause the KIPR Link to crash.

You will need to read the data sheet for the particular motor you are using to figure out how the motor needs to be wired and whether or not it can be operated using a KIPR Link motor port and PWM to vary motor response.

KIPR DC motors use wire peeled from standard 28AWG ribbon cable, which because it is stranded is much more flexible than solid wire. Your DC motor connection will require a two wire cable (as is the case for connecting a DC motor to a battery).

KIPR motor plugs are based on 1 x 3 male headers clipped from longer header rows, with the middle pin pulled clear to provide a gap between Pin 1 and Pin 3.

To make the connection between motor and plug, form leads by separating the wires a small amount on each end of the cable, then strip off enough insulation to be able to solder the leads to your motor connections on one end and to the plug on the other.

Once your cable is attached to your plug and your motor, you need to insulate the leads to prevent any cross contact. KIPR sensors use hot melt glue and shrink wrap tubing for this purpose, which on the plug end also stabilizes the connection and provides strain relief (to flatten the glue before it hardens, sandwich between two refrigerated pieces of smooth metal, then trim off excess glue using a razor knife or sharp scissors).

Before plugging your new motor into your KIPR Link, use a multi meter to check for shorts and for continuity between motor connections and plug pins.

Navigate to the motors screen on the KIPR Link (*Motors and Sensors .. Motors*), and plug in your motor. Select the port plugged into and press *Power*. Pressing *Forward* or *Backward* should operate your motor and represents the maximum (PWM) power the KIPR Link provides.

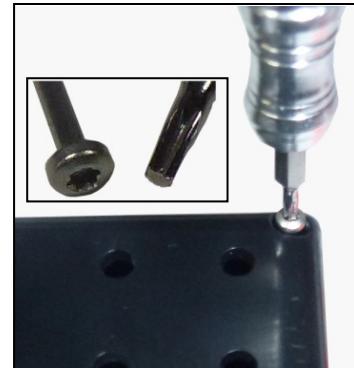
Setting the sensor ports for 5V or 3.3V

Warning! This modification requires opening your KIPR Link case, which will void your warranty. KIPR assumes no liability for the accuracy of these instructions and following them is strictly at your own risk regarding any damage which might occur to either person or equipment employed.

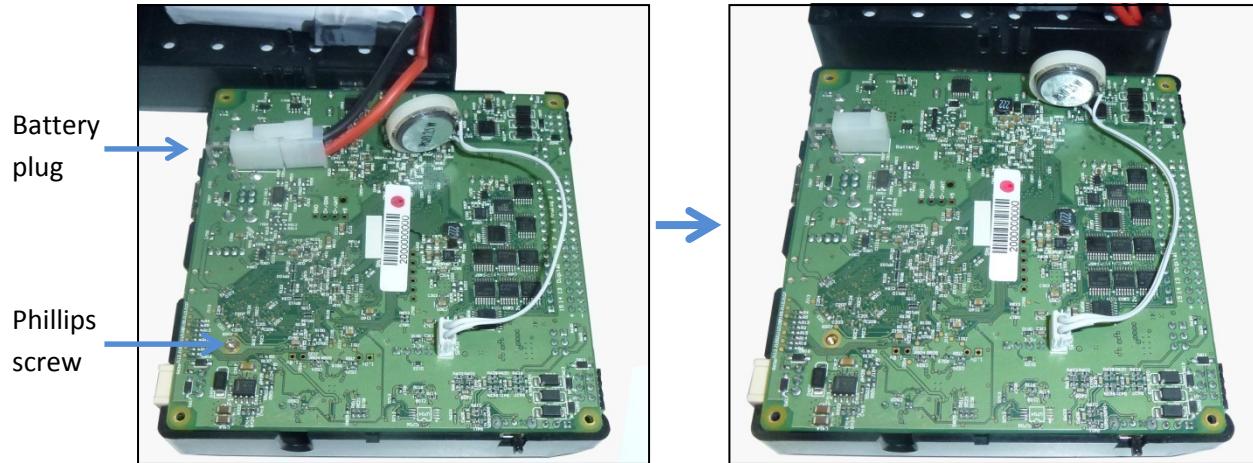
By default the digital and analog sensor ports on the KIPR Link are set to provide +5V on the V_{cc} rail, which is also the nominal voltage for the **SEN** rail. This is internally jumper selectable to be either +5V or +3.3V. There are separate jumpers for the analog rails (0-7) and the digital rails (8-15). The powered sensors offered in the Botball Store (<https://botballstore.com>) will function properly for either setting, but that may not be true for sensors you construct yourself.

Before opening your KIPR Link case, make sure your work area is static free, and keep in mind that opening your KIPR Link case voids your KIPR Link warranty. A grounding strap is strongly recommended for performing this modification. As common sense would dictate, be sure your KIPR Link charger is not plugged into the KIPR Link before you proceed.

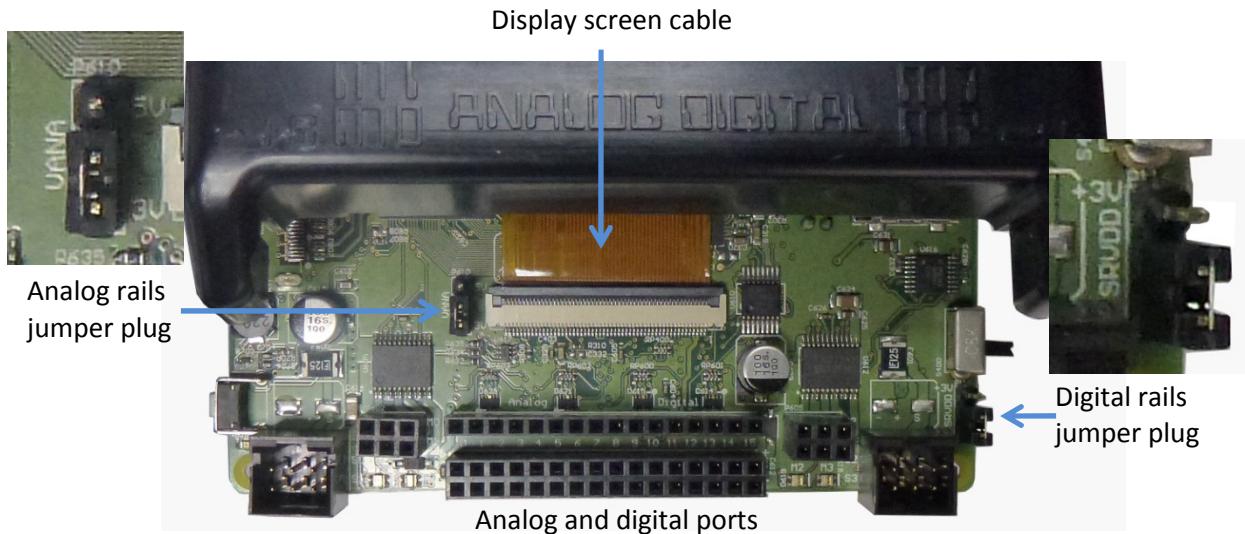
Step 1: Remove the 4 corner screws that hold the bottom half of the KIPR Link case to its top. These are socketed for a Torx #7 screw driver (available from most hardware stores).



Step 2: Lift off the bottom half of the KIPR Link case and disconnect the battery, then remove the (single) Phillips head screw attaching the circuit board to the top half of the KIPR Link case.



Step 3: Carefully loosen the circuit board from the top half of the case, turn the assembly over and move the top half of the case back just enough off of the circuit board to expose the jumper plugs as shown below. Be very careful not to kink or otherwise stress the cable which attaches the display screen to the circuit board. To reset a jumper for the desired voltage level, pull its shunt off of the pins and reinstall it on the pins as indicated by the values printed on the circuit board. In the picture below, the analog ports are set for +3.3V and the digital ports for +5V.



Step 4: Turn the top of the case back over and carefully reinstall the circuit board, then replace the Phillips head screw.

Step 5: Reattach the battery and close up your KIPR Link, paying particular attention that the battery cables are clear and the speaker is back in place, then reinstall the 4 Torx screws that hold the bottom half of the case to the top.

Step 6: Boot your KIPR Link to verify it is still working.

KIPR Link Main Library Functions

(alphabetic order - Math functions are from the C math library and are also automatically included)

a_button [Category: Sensors]

Format: `int a_button();`

Reads the value (0 or 1) of the A button.

a_button_clicked [Category: Sensors]

Format: `int a_button_clicked();`

Gets the A button's state (pressed or not pressed). If pressed, blocks until released. Returns 1 for pressed, 0 for not pressed. The construction

```
while (a_button()==0) {  
    while (a_button()==1); . . . } //debounce A button
```

is equivalent to

```
while (a_button_clicked()==0) { . . . }
```

accel_x [Category: Sensors]

Format: `int accel_x();`

Returns the value of the accelerometer in its x direction relative to the horizontal plane of the KIPR Link.

accel_y [Category: Sensors]

Format: `int accel_y();`

Returns the value of the accelerometer in its y direction relative to the horizontal plane of the KIPR Link.

accel_z [Category: Sensors]

Format: `int accel_z();`

Returns the value of the accelerometer for its vertical, or z direction, relative to the horizontal plane of the KIPR Link. When the Link is horizontal it is calibrated to have a value corresponding to the gravitational constant G (your acceleration to towards the center of the Earth to keep you from flying off of the planet).

alloff [Category: Motors]

Format: `void alloff();`

Turns off all motors. `ao` is a short form for `alloff`.

analog [Category: Sensors]

Format: `int analog(int p);`

Returns the value of the sensor installed at the port numbered `p`. The result is an integer between 0 and 255. The function can be used with analog ports 0 through 7.

analog10 [Category: Sensors]

Format: `int analog10(int p);`

10-bit version of the analog function. The returned value is in the range 0 to 1023 rather than 0 to 255.

any_button [Category: Sensors]

Format: `int any_button();`

Returns 1 if any button is pressed (the Side button or any of the 6 soft buttons A,B,C,X,Y,Z).

ao [Category: Motors]

Format: `void ao();`

Turns off all motors.

atan [Category: Math]
Format: `double atan(double angle);`
Returns the arc tangent of the angle. `angle` is specified in radians.

b_button [Category: Sensors]
Format: `int b_button();`
Reads the value (0 or 1) of the B button.

b_button_clicked [Category: Sensors]
Format: `int b_button_clicked();`
Gets the B button's state (pressed or not pressed). If pressed, blocks until released. Returns 1 for pressed, 0 for not pressed. The construction
`while (b_button()==0) {
 while (b_button()==1); . . . } //debounce B button`
is equivalent to
`while (b_button_clicked()==0) { . . . }`

beep [Category: Output]
Format: `void beep();`
Produces a tone. Returns when the tone is finished.

bk [Category: Motors]
Format: `void bk(int m);`
Turns motor `m` on full speed in the backward direction.
Example:
`bk(1);`

block_motor_done [Category: Motors]
Format: `void block_motor_done(int m);`
Function does not return until specified motor completes any executing speed or position control movement.
Example:
`mrp(0, 500, 20000L);
block_motor_done(1);`

bmd [Category: Motors]
Format: `void bmd(int m);`
This function is the same as `block_motor_done`.

c_button [Category: Sensors]
Format: `int c_button();`
Reads the value (0 or 1) of the C button.

c_button_clicked [Category: Sensors]
Format: `int c_button_clicked();`
Gets the C button's state (pressed or not pressed). If pressed, blocks until released. Returns 1 for pressed, 0 for not pressed. The construction
`while (c_button()==0) {
 while (c_button()==1); . . . } //debounce C button`
is equivalent to
`while (c_button_clicked()==0) { . . . }`

console_clear [Category: Output]
Format: `void console_clear();`
Clear the Link print buffer. See also `display_clear`.

display_clear [Category: Output]
Format: `void display_clear();`
Clear the Link display for `display_printf`. See also `console_clear`.

display_printf [Category: Output]
Format: `void display_printf(int col, int row, char s[], ...);`
Perform a standard `printf` starting at screen location `col`, `row`. (`col` 0 to 41, `row` 0 to 9 - less if extra buttons are turned on)

clear_motor_position_counter [Category: Motors]
Format: `void clear_motor_position_counter(int motor_nbr);`
Reset the position counter for the motor specified to 0.

cos [Category: Math]
Format: `double cos(double angle);`
Returns cosine of angle. `angle` is specified in radians; result is in radians.

digital [Category: Sensors]
Format: `int digital(int p);`
Returns the value of the sensor in sensor port `p`, as a true/false value (1 for true and 0 for false). Sensors are expected to be active low, meaning that they are valued at zero volts in the active, or true, state. Thus the library function returns the inverse of the actual reading from the digital hardware: if the reading is zero volts or logic zero, the `digital` function will return true. Valid for digital ports 8-15 only.

disable_servo [Category: Servos]
Format: `void disable_servo(int p);`
Disables specified servo port.

disable_servos [Category: Servos]
Format: `void disable_servos();`
Disables the servo motor ports (powers down all servo motors).

enable_servo [Category: Servos]
Format: `void enable_servo(int p);`
Enables specified servo port.

enable_servos [Category: Servos]
Format: `void enable_servos();`
Enables all servo motor ports.

exp10 [Category: Math]
Format: `double exp10(double num);`
Returns 10 to the `num` power.

exp [Category: Math]
Format: `double exp(double num);`
Returns e to the `num` power.

extra_buttons_show [Category: Output]
Format: `void extra_buttons_show();`
Shows the X, Y, and Z buttons on the Link display. Note: this reduces the display area for `printf` and `display_printf`. See also `extra_buttons_hide`, `get_extra_buttons_visible`.

extra_buttons_hide [Category: Output]
Format: `void extra_buttons_hide();`
Hides the X, Y, and Z buttons on the Link display. Note: this is the default display configuration. See also `extra_buttons_show`, `get_extra_buttons_visible`.

fd [Category: Motors]
Format: `void fd(int m);`
Turns motor **m** on full in the forward direction.
Example:
`fd(3);`

freeze [Category: Motors]
Format: `void freeze(int m);`
Freezes motor **m** (prevents continued motor rotation, in contrast to **off**, which allows the motor to "coast").

get_analog_pullup [Category: Sensors]
Format: `int get_analog_pullup(int port);`
Returns 1 if the port's pull up resistor is set (the default), and 0 otherwise. See also [set_analog_pullup](#).

get_extra_buttons_visible [Category: Sensors]
Format: `int get_extra_buttons_visible();`
Returns 1 if the X, Y, and Z buttons are visible, 0 if not. See also, [extra_buttons_show](#), [extra_buttons_hide](#).

get_motor_done [Category: Motors]
Format: `int get_motor_done(int m);`
Returns whether the motor has finished a move with specified position.

get_motor_position_counter [Category: Motors]
Format: `int get_motor_position_counter(int m);`
Returns the current motor position value for motor **m** (a value which is continually being updated for each motor using back EMF; a typical discrimination for a given motor is on the order of 1100 position "ticks" per rotation).

get_pid_gains [Category: Motors]
Format: `int get_pid_gains(int motor, int *p, int *i, int *d,
 int *pd, int *id, int *dd);`
This function is used to obtain the weights of the PID control currently set for the motors. The **p**, **i** and **d** parameters are the numerators for the P, I and D coefficients. The **pd**, **id** and **dd** parameters are their respective denominators. Thus all of the parameters are integers, but the actual coefficients can be floats. If a motor is jerky, the P and D terms should be reduced in size. If a motor lags far behind, they should be increased. The default values are set at firmware install. See also [set_pid_gains](#).

get_servo_enabled [Category: Servos]
Format: `int get_servo_enabled(int srv);`
Returns 1 if the specified servo port is enabled and 0 otherwise. See also [enable_servo](#), [disable_servo](#).

get_servo_position [Category: Servos]
Format: `int get_servo_position(int srv);`
Returns the position value of the servo in port **srv**. The value is in the range 0 to 2047. There are 4 servo ports (0, 1, 2, 3).

log10 [Category: Math]
Format: `double log10(double num);`
Returns the logarithm of **num** to the base 10.

log [Category: Math]
Format: `double log(double num);`
Returns the natural logarithm of **num**.

mav [Category: Motors]
Format: `void mav(int m, int vel);`
This function is the same as **move_at_velocity**.

motor [Category: Motors]
Format: `void motor(int m, int p);`
Turns on motor **m** at scaled PWM duty cycle percentage **p**. Power levels range from 100 for full on forward to -100 for full on backward.

move_at_velocity [Category: Motors]
Format: `void move_at_velocity(int m, int vel);`
Moves motor **m** at velocity **vel** indefinitely. The velocity range is -1000 to 1000 ticks per second.

move_relative_position [Category: Motors]
Format: `void move_relative_position(int m, int vel, int ticks);`
Moves motor **m** at velocity **vel** from its current position **curr_pos** to **curr_pos + ticks**. The speed range is 0 to 1000 ticks per second.
Example:
`move_relative_position(1, 275, -1100L);`

move_to_position [Category: Motors]
Format: `void move_to_position(int m, int vel, int pos);`
Moves motor **m** at velocity **vel** from its current position **curr_pos** to **pos**. The speed range is 0 to 1000. Note that if the motor is already at **pos**, the motor doesn't move.

mrp [Category: Motors]
Format: `void mrp(int m, int vel, int pos);`
This function is the same as **move_relative_position**.

mtp [Category: Motors]
Format: `void mtp(int m, int vel, int pos);`
This function is the same as **move_to_position**.

msleep [Category: Time]
Format: `void msleep(int msec);`
Waits for an amount of time equal to or greater than **msec** milliseconds.
Example:
`msleep(1500); //wait for 1.5 seconds`

off [Category: Motors]
Format: `void off(int m);`
Turns off motor **m**.
Example:
`off(1);`

power_level [Category: Sensor]
Format: `double power_level();`
Returns the current power level in volts.

printf [Category: Output]
Format: `void printf(char s[], ...);`
Prints the contents of the string referenced by `s` to the cursor position on the screen.
See the programmers manual embedded with the KISS IDE for more details.

random [Category: Math]
Format: `int random(int m);`
Returns a random integer between 0 and some very large number.

run_for [Category: Processes]
Format: `void run_for(double sec, void <function_name>);`
This function takes a function and runs it for a certain amount of time in seconds. `run_for` will return within 1 second of your function exiting, if it exits before the specified time. The `sec` argument denotes how many seconds to run the given function.

seconds [Category: Time]
Format: `double seconds();`
Returns the count of system time in seconds, as a floating point number. Resolution is one millisecond.

set_a_button_text [Category: Sensors]
Format: `void set_a_button_text(char txt[]);`
This function sets the text displayed on the A button to be the text string specified rather than 'A'.

set_analog_pullup [Category: Sensors]
Format: `void set_analog_pullup(int port, int pullupTF);`
The purpose of this function is to enable or disable the pull up resistor present on each analog port. Without the pull up resistor, the port is said to be floating. For example, `set_analog_pullup(3, 0);` configures analog port 3 to be "floating" (no pull up resistor) whereas `set_analog_pullup(3, 1);` configures port 3 as pull up (enables the pull up resistor). Since many analog sensors lack an integrated pull up resistor (the "ET" sensor being a notable exception), all sensor ports are set to non-floating when the KIPR Link is rebooted or when a program exits.

set_b_button_text [Category: Sensors]
Format: `void set_b_button_text(char txt[]);`
This function sets the text displayed on the B button to be the text string specified rather than 'B'.

set_c_button_text [Category: Sensors]
Format: `void set_c_button_text(char txt[]);`
This function sets the text displayed on the C button to be the text string specified rather than 'C'.

set_digital_output [Category: Output]
Format: `void set_digital_output(int port, int inout);`
Digital ports on the KIPR Link can be configured for either input or output. By default digital ports are set for input. The statement `set_digital_output(9, 1)` will configure digital port 9 for output. The `port` parameter must be in the range of values 8-15.

`set_digital_pullup` [Category: Sensors]

Format: `void set_digital_pullup(int port, int pullupTF);`

Digital ports provides a pull up resistor for sensors that don't have an integrated pull up resistor which can be turned off for sensors that set their own pull up value (there aren't any digital sensors of this type used for Botball). For example, `set_digital_pullup(9, 0);` configures digital port 9 to be "floating" (no pull up resistor) whereas `set_digital_pullup(9, 1);` configures port 9 as pull up (enables the pull up resistor)

`set_digital_value` [Category: Output]

Format: `void set_digital_value(int port, int value);`

Digital ports on the KIPR Link can be configured for either input or output. For a port configured for output, this function is used to set its value to either 0 (low) or 1 (high). The `port` parameter must be in the range of values 8-15.

`set_pid_gains` [Category: Motors]

Format: `int set_pid_gains(int motor, int p, int i, int d, int pd, int id, int dd);`

This function is used to adjust the weights of the PID control for the motors. The `p`, `i` and `d` parameters are the numerators for the P, I and D coefficients. The `pd`, `id` and `dd` parameters are their respective denominators. Thus all of the parameters are integers, but the actual coefficients can be floats. If a motor is jerky, the P and D terms should be reduced in size. If a motor lags far behind, they should be increased. The default values are set at firmware install and may be adjusted using the pid screen.

`set_servo_position` [Category: Servos]

Format: `int set_servo_position(int srv, int pos);`

Sets the position value of the servo in port `srv`. The value of `pos` must be in the range 0 to 2047. There are 4 servo ports (0, 1, 2, 3).

`set_x_button_text` [Category: Sensors]

Format: `void set_x_button_text(char txt[]);`

This function sets the text displayed on the X button to be the text string specified rather than 'X'. See also `extra_buttons_hide`, `get_extra_buttons_visible`.

`set_y_button_text` [Category: Sensors]

Format: `void set_y_button_text(char txt[]);`

This function sets the text displayed on the Y button to be the text string specified rather than 'Y'. See also `extra_buttons_hide`, `get_extra_buttons_visible`.

`set_z_button_text` [Category: Sensors]

Format: `void set_z_button_text(char txt[]);`

This function sets the text displayed on the Z button to be the text string specified rather than 'Z'. See also `extra_buttons_hide`, `get_extra_buttons_visible`.

`setpwm` [Category: Motors]

Format: `int setpwm(int m, int dutycycle);`

Runs motor `m` at duty cycle `dutycycle` (values-100 to 100)

`side_button` (or `black_button`) [Category: Sensors]

Format: `int side_button();`

Reads the value (0 or 1) of the (physical) side button on the KIPR Link.

side_button_clicked [Category: Sensors]
Format: `int side_button_clicked();`
Gets the Side button's state (pressed or not pressed). If pressed, blocks until released. Returns 1 for pressed, 0 for not pressed. The construction
`while (side_button()==0) {
while (side_button()==1); . . . } //debounce Side button`
is equivalent to
`while (side_button_clicked()==0) {. . . }`

sin [Category: Math]
Format: `double sin(double angle);`
Returns the sine of angle. `angle` is specified in radians; result is in radians.

sqrt [Category: Math]
Format: `double sqrt(double num);`
Returns the square root of `num`.

tan [Category: Math]
Format: `double tan(double angle);`
Returns the tangent of angle. `angle` is specified in radians; result is in radians.

thread_create [Category: Threads]
Format: `thread thread_create(<function name>);`
thread_create is used to create a thread for running a function in parallel to **main**, returning a thread ID value of type **thread**. The special data type **thread** is for the thread IDs used by the system to keep track of active threads. Note that the returned value must be assigned to a variable of type **thread** to remain available. When a function is run in a thread (via **thread_start**), the thread will remain active until the function finishes or the thread is destroyed (via **thread_destroy**). If the thread hasn't been destroyed, it can be started again; otherwise, a new thread has to be created for the function.

thread_destroy [Category: Threads]
Format: `void thread_destroy(thread id);`
thread_destroy is used to destroy a thread created for a function. A thread is destroyed by passing its thread ID to **thread_destroy**. The following example shows the **main** process creating a `check_sensor` thread, running it in parallel via **thread_start**, and then destroying it one second later (whether or not the thread is still active):

```
int main() {
    threadtid;
    tid = thread_create(check_sensor);
    thread_start(tid);
    msleep(1000);
    thread_destroy(tid);
}
```

thread_start [Category: Threads]
Format: `void thread_start(thread id);`
thread_start is used to activate a thread, running its associated function in parallel with **main** and any other active threads. The **thread** variable used in the argument must have a thread ID value as returned by **thread_create**. Note that thread IDs generated by **thread_create** must be retained in variables of type **thread** to remain available for later use. The thread is active until its function finishes or until it is terminated by **thread_destroy**.

thread_wait [Category: Threads]

Format: `void thread_wait(thread id);`

The `thread_wait` function is used to wait for a thread that has been started by `thread_start` to finish, where `id` is the thread ID value returned by `thread_create` when the thread was created.

x_button [Category: Sensors]

Format: `int x_button();`

Reads the value (0 or 1) of the X button. This button is an extra button. Use `extra_buttons_show()` to show the X, Y, and Z buttons. See also `extra_buttons_hide`, `get_extra_buttons_visible`.

x_button_clicked [Category: Sensors]

Format: `int x_button_clicked();`

Gets the X button's state (pressed or not pressed). If pressed, blocks until the button is released. Returns 1 for pressed, 0 for not pressed. The "debounce" construction

`while (x_button()==0) { while (x_button()==1); . . . }`

is equivalent to

`while (x_button_clicked()==0) { . . . }`

This button is an extra button. Use `extra_buttons_show` to show the X, Y, and Z buttons. See also `extra_buttons_hide`, `get_extra_buttons_visible`.

y_button [Category: Sensors]

Format: `int y_button();`

Reads the value (0 or 1) of the Y button. This button is an extra button. Use

`extra_buttons_show` to show the X, Y, and Z buttons. See also `extra_buttons_hide`, `get_extra_buttons_visible`.

y_button_clicked [Category: Sensors]

Format: `int y_button_clicked();`

Gets the Y button's state (pressed or not pressed). If pressed, blocks until the button is released. Returns 1 for pressed, 0 for not pressed. The "debounce" construction

`while (y_button()==0) { while (y_button()==1); . . . }`

is equivalent to

`while (y_button_clicked()==0) { . . . }`

This button is an extra button. Use `extra_buttons_show` to show the X, Y, and Z buttons. See also `extra_buttons_hide`, `get_extra_buttons_visible`.

z_button [Category: Sensors]

Format: `int z_button();`

Reads the value (0 or 1) of the Z button. This button is an extra button. Use

`extra_buttons_show` to show the X, Y, and Z buttons. See also `extra_buttons_hide`, `get_extra_buttons_visible`.

z_button_clicked [Category: Sensors]

Format: `int z_button_clicked();`

Gets the Z button's state (pressed or not pressed). If pressed, blocks until the button is released. Returns 1 for pressed, 0 for not pressed. The "debounce" construction

`while (z_button()==0) { while (z_button()==1); . . . }`

is equivalent to

`while (z_button_clicked()==0) { . . . }`

This button is an extra button. Use `extra_buttons_show` to show the X, Y, and Z buttons. See also `extra_buttons_hide`, `get_extra_buttons_visible`.

KIPR Link Vision Library Functions

camera_close [Category: Vision]

Format: `void camera_close();`

Cleanup the current camera instance. See also `camera_open`, `camera_open_device`.

camera_load_config [Category: Vision]

Format: `int camera_load_config(char name[]);`

Loads a config file on the KIPR Link in place of the default config file. You **must** append `.config` to the file name for this function to locate it. Returns 1 on success, 0 on failure. See also `camera_open`, `camera_open_device`.

camera_open [Category: Vision]

Format: `int camera_open(int res_numb);`

Opens the KIPR Link's default channel configuration. The default configuration is selected from among the channel configurations defined on the KIPR Link using its *Settings .. Channels* menu. A resolution of one of `LOW_RES`, `MED_RES`, `HIGH_RES` needs to be specified. Returns 1 on success, 0 on failure. See also `camera_open_device`, `camera_close`.

camera_open_device [Category: Vision]

Format: `int camera_open_device(int number, int res_numb);`

If more than 1 camera is plugged in, 0 is the first camera, 1 is the second camera, etc. Only 1 camera at a time can be used, and the default configuration is selected. A resolution of one of `LOW_RES`, `MED_RES`, `HIGH_RES` needs to be specified. Returns 1 on success, 0 on failure. See also `camera_open`, `camera_close`.

camera_update [Category: Vision]

Format: `int camera_update();`

Pulls a new image from the camera for processing. Returns 1 on success, 0 on failure.

get_channel_count [Category: Vision]

Format: `int get_channel_count();`

Returns the number of channels in the current configuration. See also `get_object_count`.

get_code_num [Category: Vision]

Format: `int get_code_num(int channel, int object);`

Returns the data associated with the given channel and object as an integer. If the given channel or object doesn't exist, -1 is returned. See also `get_object_data`.

get_object_area [Category: Vision]

Format: `int get_object_area(int channel, int object);`

Returns the object's bounding box area. -1 is returned if the channel or object doesn't exist.

get_object_bbox [Category: Vision]

Format: `rectangle get_object_bbox(int channel, int object);`

Returns the bounding box of the given object on the given channel as a `rectangle` data type.

For example,

```
rectangle mybox;  
mybox = get_object_bbox(0, 2);  
printf("x coord %d y coord %d\n", mybox.x, mybox.y);
```

displays the x and y coordinates of bounding box 2 for channel 0.

get_object_center [Category: Vision]

Format: `point2 get_object_center(int channel, int object);`

Returns the (x, y) center of the given object on the given channel as a `point2` data type.

For example,

```
point2 cntr;  
cntr = get_object_center(0, 2);  
printf("Center: x coord %d y coord %d\n", cntr.x, cntr.y);
```

displays the x and y coordinates of center point of box 2 for channel 0.

get_object_centroid [Category: Vision]

Format: `int get_object_centroid(int channel, int object);`

Returns The (x, y) coordinates of the **centroid** of the given object on the given color channel as a `point2` data type (the centroid is the center of mass for the pixels of the specified color). For example,

```
point2 cntd;  
cntd = get_object_centroid(0, 2);  
printf("centroid: x coord %d y coord %d\n", cntd.x, cntd.y);
```

displays the x and y coordinates of centroid of box 2 for color channel 0. The centroid is NOT the same as the center. It is the center of mass for a blob; e.g., for a color arrow pointing right, there are more pixels to right of center, so the centroid is to the right of center.

get_object_confidence [Category: Vision]

Format: `double get_object_confidence(int channel, int object);`

Returns the confidence, between 0.0 and 1.0, that the given object on the given channel is significant. If the channel or object doesn't exist, 0.0 is returned.

get_object_count [Category: Vision]

Format: `int get_object_count(int channel);`

Returns the number of objects being "seen" by the specified channel. Objects are sorted by area, largest first. Returns -1 if channel doesn't exist. See also `get_channel_count`.

get_object_data [Category: Vision]

Format: `char *get_object_data(int channel, int object);`

Returns the sequence of character data associated with a given object on a QR channel. If there is no data associated, 0 is returned. The data is not guaranteed to be null terminated, but can be accessed using array notation; for example,

`get_object_data(0, 0)[0]; get_object_data(0, 0)[1]; etc.`

`camera_update` will invalidate the pointer returned by `get_object_data`. See also `get_object_data_length`.

get_object_data_length [Category: Vision]

Format: `int get_object_data_length(int channel, int object);`

Returns the number of characters associated with the QR code on a QR channel. If there is no data associated, 0 is returned. If the channel or object is invalid, 0 is returned. See also `get_object_data`.

KIPR Link iRobot Create Library Functions

Create serial interface functions

create_clear_serial_buffer [Category: Create Serial Interface]

Format: `void create_clear_serial_buffer () ;`

Clears the internal serial interface buffer of any unaccessed send/receive data.

create_connect [Category: Create Serial Interface]

Format: `int create_connect () ;`

Establishes a USB serial connection between the KIPR Link and a Create module. This statement is normally paired with an `msleep` statement, since it takes more than one second for the communications link to stabilize (`msleep(1500)` is sufficient). If the program is paused and the Create is not turned on, the function will block continued execution until the Create is turned on. This function is always the first step for sending Create Open Interface commands from the KIPR Link to the Create. By default, the Create starts in `create_safe` mode.

create_disconnect [Category: Create Serial Interface]

Format: `void create_disconnect () ;`

Restores the Create to power on configuration (which will also shut off any running motors).

create_read_block [Category: Create Serial Interface]

Format: `int create_read_block(char *data, int count) ;`

Uses the serial interface to have the Create send the number of bytes specified into the character string `data`. 1 is returned on read success, 0 on failure.

create_write_byte [Category: Create Serial Interface]

Format: `void create_write_byte (char byte) ;`

Uses the serial interface to have the KIPR Link send the byte to the iRobot Create.

Create configuration functions

create_full [Category: Create Configuration Function]

Format: `void create_full () ;`

Create will move however you tell it (even if that is a bad thing). In particular, the Create will not stop and disconnect if a wheel drop or cliff sensor goes high.

create_passive [Category: Create Configuration Function]

Format: `void create_passive () ;`

Puts Create into passive mode (motor commands won't work).

create_safe [Category: Create Configuration Function]

Format: `void create_safe () ;`

Create will move however you tell it (even if that is a bad thing). In particular, the Create will not stop and disconnect if a wheel drop or cliff sensor goes high.

create_start [Category: Create Configuration Function]

Format: `void create_start();`

Puts Create back into active mode (all commands will work). Active mode is the default mode at power on.

get_create_mode [Category: Create Configuration Function]

Format: `int get_create_mode();`

Returns the Create's current operating mode (0=off, 1=passive, 2=safe, 3=full). In passive mode, motor commands don't work. All commands work in safe or full mode. In safe mode, the Create will stop all motors and disconnect if any cliff sensors or wheel drop sensors go high. In full mode, the Create will continue any movement commands and remain connected regardless of sensor values.

Create movement functions

create_drive [Category: Create Movement Function]

Format: `void create_drive(int speed, int radius);`

Drives in an arc (see below for point turns and straight). Speed range for all commands is 20-500mm/sec.

create_drive_direct [Category: Create Movement Function]

Format: `void create_drive_direct(int r_speed, int l_speed);`

Specifies individual left and right speeds in mm/sec.

create_drive_straight [Category: Create Movement Function]

Format: `void create_drive_straight(int speed);`

Drives straight at speed in mm/sec.

create_spin_CW [Category: Create Movement Function]

Format: `void create_spin_CW(int speed);`

Spins Clockwise with edge speed of speed in mm/sec.

create_spin_CCW [Category: Create Movement Function]

Format: `void create_spin_CCW(int speed);`

Spins Counterclockwise with edge speed of speed in mm/sec.

create_stop [Category: Create Movement Function]

Format: `void create_stop();`

Stops the drive wheels.

get_create_distance [Category: Create Movement Function]

Format: `int get_create_distance();`

Returns the accumulated distance the Create has traveled since it was turned on or since the distance was reset. Moving backwards reduces this value. The distance is in millimeters.

get_create_normalized_angle [Category: Create Movement Function]

Format: `int get_create_normalized_angle();`

Returns the accumulated angle the Create has turned since it was turned on or the distance was reset, normalized to the range 0 to 359 degrees. Turning CCW increases this value and CW decreases the value.

get_create_overcurrents [Category: Create Movement Function]
Format: `int get_create_overcurrents();`
Returns the overcurrent status byte where the 16's bit indicates overcurrent in the left wheel; 8's bit overcurrent in the right wheel, 4's bit is LD2, 2's bit is LD0 and 1's bit is LD1 (LD is for the Create's 3 low side driver outputs, pins 22 to 24 for the connector in the Create cargo bay).
Seldom used in practice.

get_create_requested_left_velocity [Category: Create Movement Function]
Format: `int get_create_requested_left_velocity();`
Returns the speed the Create is moving (-500 to 500mm/sec) the left wheel according to the most recent movement command executed.

get_create_requested_radius [Category: Create Movement Function]
Format: `int get_create_requested_radius();`
Returns the radius the Create is turning (-2000 to 2000mm) according to the most recent movement command executed.

get_create_requested_right_velocity [Category: Create Movement Function]
Format: `int get_create_requested_right_velocity();`
Returns the speed the Create is moving (-500 to 500mm/sec) the right wheel according to the most recent movement command executed.

get_create_requested_velocity [Category: Create Movement Function]
Format: `int get_create_requested_velocity();`
Returns the speed the Create is moving (-500 to 500mm/s) according to the most recent movement command executed.

get_create_total_angle [Category: Create Movement Function]
Format: `int get_create_total_angle();`
Returns the accumulated angle the Create has turned through since it was turned on or since the distance or angle was reset. Turning CCW increases this value and CW decreases the value.

set_create_distance [Category: Create Movement Function]
Format: `void set_create_distance(int dist);`
Sets the current value that will be returned by get_create_distance to the value dist.

set_create_normalized_angle [Category: Create Movement Function]
Format: `void set_create_normalized_angle(int angle);`
Sets the current value that will be returned by get_create_normalized_angle to the value angle.

set_create_total_angle [Category: Create Movement Function]
Format: `void set_create_total_angle(int angle);`
Sets the current value that will be returned by get_create_total_angle to the value angle.

Create sensor functions

get_create_advance_button [Category: Create Sensor Function]
Format: `int get_create_advance_button();`
Returns 1 if the advance (>>|) button is being pressed, 0 otherwise.

`get_create_bay_AI` [Category: Create Sensor Function]

Format: `int get_create_bay_AI();`

Returns the 10 bit analog value on pin 4 from the cargo bay.

`get_create_bay_DI` [Category: Create Sensor Function]

Format: `int get_create_bay_DI();`

Returns a byte for determining the current digital inputs (0 or 1) being applied to pins 16, 6, 18, 5, and 17 of the connector in the Create cargo bay. The 128, 64, and 32 bits of the byte are not used. The 16 bit is for pin 15, 8 bit for pin 6, 4 bit for pin 18, 2 bit for pin 5 and 1 bit for pin 17.

Pin 15 is used to alter communications baud rate.

`get_create_cwdrop` [Category: Create Sensor Function]

Format: `int get_create_cwdrop();`

Returns 1 if caster wheel has dropped, 0 otherwise.

`get_create_infrared` [Category: Create Sensor Function]

Format: `int get_create_infrared();`

Returns the byte detected from an iRobot remote control, Returns 255 if no byte has been detected.

`get_create_lbump` [Category: Create Sensor Function]

Format: `int get_create_lbump();`

Returns 1 if left bumper is pressed, 0 otherwise.

`get_create_lcliff` [Category: Create Sensor Function]

Format: `int get_create_lcliff();`

Returns 1 if the left cliff sensor is over a surface that doesn't reflect IR (e.g., black) or over a cliff, 0 otherwise.

`get_create_lcliff_amt` [Category: Create Sensor Function]

Format: `int get_create_lcliff_amt();`

Returns the left cliff sensor (analog) reading as a 12 bit value (0 to 4095).

`get_create_lfcliff` [Category: Create Sensor Function]

Format: `int get_create_lfcliff();`

Returns 1 if left front cliff sensor is over a surface that doesn't reflect IR (e.g., black) or over a cliff, 0 otherwise.

`get_create_lfcliff_amt` [Category: Create Sensor Function]

Format: `int get_create_lfcliff_amt();`

Returns the left front cliff sensor (analog) reading as a 12 bit value (0 to 4095).

`get_create_lwdrop` [Category: Create Sensor Function]

Format: `int get_create_lwdrop();`

Returns 1 if the left wheel has dropped, 0 otherwise. Materials supplied with the Create include two wheel clips that when installed will prevent the drive wheels from dropping.

`get_create_number_of_stream_packets` [Category: Create Sensor Function]

Format: `int get_create_number_of_stream_packets();`

If data streaming is being used, it returns the size of the stream.

get_create_play_button [Category: Create Sensor Function]
Format: `int get_create_play_button();`
Returns 1 if the play button (>) is being pressed, 0 otherwise.

get_create_rbump [Category: Create Sensor Function]
Format: `int get_create_rbump();`
Returns 1 if right bumper is pressed, 0 otherwise.

get_create_rcliff [Category: Create Sensor Function]
Format: `int get_create_rcliff();`
Returns 1 if right cliff sensor is over black or a cliff, 0 otherwise.

get_create_rfcliff [Category: Create Sensor Function]
Format: `int get_create_rfcliff();`
Returns 1 if right frontcliff sensor is over a surface that doesn't reflect IR (e.g., black) or over a cliff, 0 otherwise.

get_create_rfcliff_amt [Category: Create Sensor Function]
Format: `int get_create_rfcliff_amt();`
Returns the right front cliff sensor (analog) reading as a 12 bit value (0 to 4095).

get_create_rwdrop [Category: Create Sensor Function]
Format: `int get_create_rwdrop();`
Returns 1 if right wheel has dropped, 0 otherwise.

get_create_vwall [Category: Create Sensor Function]
Format: `int get_create_vwall();`
Returns 1 if a iRobot virtual wall beacon is detected, 0 otherwise.

get_create_wall [Category: Create Sensor Function]
Format: `int get_create_wall();`
Returns 1 if a wall is detected by the right facing wall sensor, 0 otherwise. There is no left facing wall sensor.

get_create_wall_amt [Category: Create Sensor Function]
Format: `int get_create_wall_amt();`
Returns the current wall sensor (analog) reading as a 12 bit value (0 to 4095).

Create battery functions

get_create_battery_capacity [Category: Create Battery Function]
Format: `int get_create_battery_capacity();`
Returns the battery capacity in mAh

get_create_battery_charge [Category: Create Battery Function]
Format: `int get_create_battery_charge();`
Returns the battery charge in mAh.

get_create_battery_charging_state [Category: Create Battery Function]
Format: `int get_create_battery_charging_state();`
0-not charging; 1-recondition charging; 2-full charging; 3-trickle charging; 4-waiting; 5-charge fault. This function is seldom used in practice.

get_create_battery_current [Category: Create Battery Function]
Format: `int get_create_battery_current();`
Returns the current flow in mA.

get_create_battery_voltage [Category: Create Battery Function]
Format: `int get_create_battery_voltage();`
Returns the battery voltage in mV.

get_create_battery_temp [Category: Create Battery Function]
Format: `int get_create_battery_temp();`
Returns the battery temperature in degrees C.

Create built-in script functions

create_spot [Category: Create Built In Script]
Format: `void create_spot();`
Simulates a Roomba doing a spot clean.

create_cover [Category: Create Built In Script]
Format: `void create_cover();`
Simulates a Roomba covering a room.

create_demo [Category: Create Built In Script]
Format: `void create_demo(int d);`
Runs built in demos (see Create Open Interface on web).

create_cover_dock [Category: Create Built In Script]
Format: `void create_cover_dock();`
Create roams around until it sees an IR dock and then attempts to dock

LED and music functions

create_advance_led [Category: Create Music/LED Function]
Format: `void create_advance_led(int on);`
The value 1 causes the Advance LED light (>>|) to turn on, 0 to turn it off.

create_play_led [Category: Create Music/LED Function]
Format: `void create_play_led(int on);`
The value 1 causes the Play LED to turn on, 0 to turn it off.

create_play_song [Category: Create Music/LED Function]
Format: `void create_play_song(int num);`
Plays the specified song that has been loaded onto the Create.

create_power_led [Category: Create Music/LED Function]

Format: `void create_power_led(int color, int brightness);`

The value 0 cases the I/O power LED to turn red, 255 to turn green. Brightness ranges from 0 to 255 with 0 representing off.

get_create_song_number [Category: Create Music/LED Function]

Format: `int get_create_song_number();`

Returns the number of the song currently selected (0 to 15).

get_create_song_playing [Category: Create Music/LED Function]

Format: `int get_create_song_playing();`

Returns 1 if a song is playing, 0 otherwise.

create_load_song [Category: Create Music/LED Function]

Format: `void create_load_song(int num);`

Loads a song from an internal 16 by 33 working array of integers to the Create, where the first column for each song is the number of notes (max is 16). The remaining columns alternate between pitch and duration. See Create Open Interface on the web for details.