KISS IDE User Manual for C

Introduction

KIPR Software Suite includes the KISS IDE, providing an editor and compilers for software development in multiple programming languages, with special purpose function library support for the KIPR Robot Controller. The KIPR Robot Controller is a complete Linux based system with color touch screen for user I/O. Its hardware provides analog and digital I/O ports, DC motor ports, servo ports, USB ports, TTL serial communication, and an HDMI port, plus an integrated accelerometer, gyro, magnetometer, Wifi and Bluetooth, SPI and other program accessible hardware.

This is the on-line manual for the development of C programs using the KIPR Software Suite, which includes an IDE. The KIPR Software Suite implements the full ANSI C specification and can target the host system or a linked KIPR Robot Controller. For information about the C programing language, including history and basic syntax, see the Wikipedia article C (programming language). For a more complete tutorial and guide for C Programming visit CPrograming. The landmark reference book on C, The C Programming Language (2nd Edition) by Kernighan and Ritchie must also be mentioned, since its authors are among the original developers for Unix and C. Their book is also the progenitor for much of the material in this documentation and other programming reference works. The Botball community website also has several articles about programming and a user forum where questions can be posted to the botball community. The KIPR Robot Controller Manual and the KIPR Sensors and Motors Manual available through KIPR have system specific information and program examples not included in this manual.

The primary purpose of this manual is to describe the C functions provided for the KIPR Robot Controller. This manual also includes a basic introduction to programming in C for those who already have experience with some other programming language or for those who need a refresher for C. To learn more about programming in C, consult one of the many books or websites that provide C references and tutorials such as the one suggested above.

KISS IDE Interface

Within the KIPR Software Suite, the KISS IDE is used to manage software projects, especially those produced for the KIPR Robot Controller. The IDE runs inside of your device's web browser. The server may be installed on your host machine or served by a KIPR robot controller. When working on a project it is used to construct new (unsaved) project files and for editing old ones. Projects are organized by the KISS IDE along a right side display panel. Any project name may be clicked on to expose the project files. Project files can be headers, source, or data files, each of which can be edited using the IDE. Basic C programs (such as those created by most Botball teams) will consist of a single source file. Each project must contain exactly one function named **`main`** in one of the source files.

The *Menu* button is used to return to the main menu for the Software Suite. The KISS IDE is located within the Development Tools section of the Software Suite. Clicking on + *Add Project* will bring up a screen for adding a new project to the list. Clicking on an exisitng project name will open up its three categories of files (*Include Files*, *Source Files*, *User Data Files*). Existing files are opened for editing by clicking on them. Clicking on + *Add File* brings up the screen to add a new file for a category. To keep from losing your edits for a file, it is important to press the *Save* button prior to switching files or using a browser navigation button. The *Compile* button will automatically save the file being edited and will compile the project. A window will appear at the bottom of the edit window indicating a successful compile or a list of compiler messages (including any errors). Clicking on a project name in the side panel makes it the active project and clicking on any file name within the project brings it forward into the editing panel. Pressing *Indent*

will auto indent program text in accord with program syntax.

To run a successfully compiled project, press the *Go to Runner* button. This switches to the project run screen which allows the user to run any projects that have been successfully compiled. The project that was being edited will be highlighted in the runner. Pressing *Run* will start the program. The graphics button will bring up the graphics screen. The console button displays the console screen. Pressing *Stop* will kill the currently running program.

Programming in C

Following a Quick C Tutorial introduction, more detailed sections are provided which describe

- C data objects including variables, constants, data types, pointers, arrays, and structures
- C statements and expressions including operators used in expressions and control flow statements used to create program logic
- C Programming style including use of white space, indentation, and comments
- C program debugging to correct syntax and semantic errors, with attention to using `printf` and commenting out sections of code
- Display screen printing using printf including multiple examples using printf % codes
- The C preprocessor including `#include`, `#define`, macros, and conditional compilation
- C math library descriptions of math functions commonly used with the KIPR Robot Controller

The C programming information provided is sufficient for most purposes, but a C programming reference such as those noted above should be consulted where more detail is needed or where elements of C not covered (such as functions with a variable number of parameters or function pointers) are needed to solve some programming issue.

A Quick C Tutorial

A C program consists of one or more function definitions along with user specified data structures, and typically utilizes pre-compiled code from function libraries. To be a C program, one of the functions must be named `main`.

The following is a simple C program which has a single defined function named `main` and which utilizes the library function `printf`. A C function returns a value whose type needs to be specified. As a C function `main` is defined by first declaring its return data type followed by its name (`main`) and argument list in parentheses (empty in this case), after which the program code for the function is given in curly braces (`{}`). Comments are normally included to document the function's purpose and expected behavior, and are ignored in the compilation process.

```
/* C Programmer's Manual
   Simple example */
int main()
{
    printf("Hello, world!\n"); // simple example
}
```

Text between `/*` and `*/` forms what is termed a bracketed comment, which can extend over multiple lines. Text that starts with `//` forms a single line comment which continues only to the end of the line. Commenting program code is a good habit to form, especially considering it has no impact on program performance.

All C functions must have a return data type. While `main` does not return a value to another function, it is expected to return an integer to the operating system (not relevant to this discussion), and so has return type `int`. In C, a function does not have to return a value, which is specified by return type `void`.

Another common numeric data type is `double`, which represents (double precision) floating point numbers used for non-integer calculation. There are additional basic (or *ordinal*) data types, and user defined data types derived from basic data types may also be employed.

The Botball, graphics, opencv, stdio, stdlib, math, strings and time libraries are all included in the header in the template. The system takes care of the needed `#include`.

The function's name immediately follows the function's return type specification (in this case, `main`). Next, in parentheses, are any arguments (or inputs) to the function. `main` has none, signified by an empty set of parentheses.

The open curly-brace (`{`) that follows the argument list signifies the start of the actual function code. Curly-braces are used to structure a program as blocks of code. For C, so-called "white space" consists of one or more of any combination of characters such as spaces, returns, new lines, and tabs, which is collapsed to a single generic white space character in compilation. For this reason, white space can be used for things like indentation that improve program readability. In particular, the KISS IDE editor automatically applies white space readability conventions to enhance files as they are edited.

The body of the function consists of a block consisting of a series of C statements. A C statement specifies a data structure or some specific action to be taken and is terminated with a semicolon (`;`). The simple example program has a single statement,

```
    printf("Hello, world!\n");
```

The action this statement specifies is calling the C library `printf` function to format and print the message `Hello, world!` to the project target. The `'\n'` directs `printf` to "start a new line" (in effect ending the current line and positioning for subsequent printing to start on the next line). When printing reaches the bottom of the KIPR Controller display, any additional lines printed cause the display to scroll up. Since calling the `printf` function is a C statement, it has to be ended with a semicolon (`;`). A common error made by beginning C programmers is omitting the semicolon that is required to end each statement.

The closing curly-brace (`}`) for the `main` function's block structure concludes its definition.

If the compiler issues a warning that `printf` is being used with an implicit declaration, then prefacing the program code with the preprocessor directive

```
    #include <stdio.h>
```

will ensure that the C preprocessor includes the declaration in preparing the code for compilation.

As a second example exhibiting features of C, the following program code defines a function named `square` which returns the mathematical square of an integer. In this case the function does not represent a complete C program, since a function named `main` for using it has not been defined yet.

```
int square(int n)
{
    return n * n;
}
```

The function is declared with return type `int`, which means that it will return an integer value.

The function name (`square`) is followed by its argument list in parentheses, where arguments are separated by commas if there are more than one (there is only one in this case). The argument for `square` is specified to be an integer and is named `n`. The data type for each argument is declared in the same manner as declaring the return data type of the function or the data type for a variable.

When a function declaration specifies arguments, the variable names representing the arguments are local to the function,

or stated another way, they are valid only within the "scope" of the function (they only have meaning within the function's block of code). For this reason, variable names used within the scope of a function will not cause a semantic conflict should their names duplicate those local to some other function.

The "scope" for the function `square` is what takes place within the block structure defining its actions (i.e., the contents within the curly braces surrounding its program statements). In this case, the function's actions consist of a single statement; namely, the `return` statement. The action of the `return` statement is to exit the function, assigning the function's return value to be what is specified by the `return` statement, in this case the result of the computation `n * n`.

Except where grouped by parentheses, expressions are evaluated according to a set of precedence rules associated with the various operators used within the expression. In this case, only the multiplication operator `*` is employed, so operator precedence is not an issue.

For a third example, the following program code defines a function which performs a function call to the `square` program.

```
#include <math.h>
double hypotenuse(int a, int b)
{
    double h;
    h = sqrt(square(a) + square(b));
    return(h);
}
```

The argument list for this function has two arguments, each with integer data type. Additionally, its block structure includes a statement specifying a "working variable" `h`. The data type for `h` is given as floating point, since the C library function `sqrt` returns a floating point argument. If its data type was specified to be integer, the fractional part of any floating point number assigned to it would be lost. In general, as is the case for `h`, any local variables used within a program block (indicated by a set of curly braces) are specified at the beginning of the block.

The value assigned to `h` is the value returned by calling the `sqrt` function from the C math library and which calculates the mathematical square root of its argument. `sqrt` is a built-in C function that takes a floating point number as its argument. For any KISS IDE target, `#include <math.h>` is needed when using functions from the math library.

The `hypotenuse` function uses the `square` function defined earlier. The `sqrt` function is specified to have a floating point argument. So what happens if `sqrt` is called with an integer argument instead of floating point? Answer: C will automatically coerce the integer to a floating point value (which uses a format involving both an exponent and a mantissa). If C's automatic coercion does the "wrong" thing, it can be bypassed by specifying how the value is to be coerced, which is accomplished by preceding it with the data type in parentheses into which it is to be converted; e.g., `(double)(square(a) + square(b))`.

The `hypotenuse` function concludes by returning the value of `h`.

The functions `square` and `hypotenuse` still do not constitute a program since a `main` function that uses them still needs to be defined. A complete program using these two functions follows:

```
/* Extended example: C Programmer's Manual */
#include <stdio.h>
#include <math.h>
int square(int n);
double hypotenuse(int a, int b);
int main()
{
```

```
    printf("Hypotenuse of a 3,4 right triangle is %d\n",(int) hypotenuse(3,4)); // coerce hypotenuse to
int
}
int square(int n)
{
    return(n * n);
}
double hypotenuse(int a, int b)
{
    double h;
    h = sqrt(square(a) + square(b));
    return(h);
}
```

Program behavior is described by `main`, which calls the C library function `printf`, which in turn calls the function `hypotenuse` to obtain the parameter value, and `hypotenuse` in turn calls the function `square` in order to do its calculation. These calls result in the calculation and printing of the length (truncated to integer) of the hypotenuse of a right triangle by supplying the lengths of its two sides to the `hypotenuse` function. Note that the floating point value supplied to `printf` from `hypotenuse` is coerced to integer data type, since otherwise `printf` will print its floating point format as if it were an integer, a semantic error.

The two C statements

```
int square(int n);
double hypotenuse(int a, int b);
```

provide C "prototype" specifications for the `square` and `hypotenuse` functions used in the program. Similar prototypes need to be supplied for the standard C library's `printf` function and the C math library's `sqrt` function. The preprocessor statements

```
#include <stdio.h>
#include <math.h>
```

have the C preprocessor provide the needed prototypes (if not needed for a given KISS IDE target, the preprocessor will skip these statements; i.e., it is usually a good habit to put them in anyway). Examples of other commonly used system header files are `<string.h>`, `<stdlib.h>`, `<stdarg.h>`, and `<time.h>`.

The declaration of a function's prototype or it's actual definition needs to precede the function's first use in program code so that the compiler will be able to correctly translate the statement which calls the function. For this example, since the definitions of the `square` and `hypotenuse` functions follow the definition of the `main` function, their prototype declarations need to be included before the definition of `main`. Absent a declaration for a function, the compiler will issue an implicit declaration warning for where it first encounters the function, the results from which may be a subsequent compilation error or incorrect results when the program is executed.

This concludes the brief C tutorial.

Data Objects

Variables and constants are basic data objects used in a C program. More complex data structures and data types based on basic data objects and types can also be defined. Specification statements (such as the one used for the variable `h` in the `hypotenuse` program above) are used to define the variables the programmer wants to use in a program. A specification statement may set a variable's initial value as well as specifying its data type.

Variables

Variable names are formed from combinations of lower case letters, capital letters, the decimal digits (0-9), and the underscore character (`'_'`). There are two restrictions that must be observed in forming the name:

- a variable name cannot begin with a decimal digit
- a variable name cannot be a keyword such as `if`, `while`, etc.

Variables which are specified as function arguments or which are defined within program blocks are called *local* variables. Their scope is limited to the structure in which they are defined. Variables defined at the same level as the `main` function and other functions (i.e., not inside any function's block structure) apply across all functions used by the program. These are called *global* variables and are used for data structures accessed by more than one function, or for function to function message passing (especially for functions operating in parallel threads).

Since functions and global variables are defined at the same level, they must have unique names. The name of a function or a global variable can be used as the name of a local variable, in which case the local use takes precedence; i.e., the scope of the local variable supercedes that of a function or global variable with the same name.

Declaration

In C, a variable is declared by a specification statement that provides its data type, and optionally its initial value. Declarations for global variable are normally grouped before the definition of `main`. Declarations for local variables are normally grouped at the beginning of the block structure in which they are used. The general form of a variable declaration is one of the following:

- <*data-type*> <*variable-name*>**;**
- <*data-type*> <*variable-name*>=<*initialization-data*>**;**

<*data-type*> can be a basic (ordinal) data type such as `int`, `double`, `char`, or a composite data type such as a pointer data type or a user-defined data type. User-defined data types are based on structures or on enumerated sets. In particular, a `struct` definition creates a user-defined data type `struct` <*struct-name*>, typically simplified by using `typedef` to provide the `struct` <*struct-name*> data type with a more digestible name. Just as `struct` provides a means for defining a data type based on a structure, `enum` provides a means for defining an enumerated data type based on an enumerated set of names.

Local and Global Scopes

If a variable is declared within a function, or as an argument to a function, its binding is local, meaning that the variable is available for use only within the context of that function definition. If a variable is declared outside of a function, it is a global variable and is available for use within any function employed by the program, including functions which are defined in files other than the one in which the global variable is declared. If a global variable represents a complex data structure whose definition would clutter up the main program file, it is sometimes advantageous to define it in a separate file to be included by the C preprocessor in preparing the program code for compilation.

Variable Initialization

The initial value of a local or global variable can optionally be specified in its declaration. If no initialization value is given, the value is indeterminate. When a global variable is initialized, the initialization value must be a constant. In contrast, local variables may be initialized to the value of arbitrary expressions including any global variables, function calls, function arguments, or local variables which have already been initialized.

The compilation process translates a program into machine executable form, initializing global variables within the

resulting execute module as specified in global variable declarations. Local variable initialization doesn't take place until program execution enters the block in which the local variable is defined. The initialization occurs every time program execution enters the block, excepting local variables whose declaration is preceded by the word `static`. For a local variable declared to be `static`, initialization only takes place the first time the block defining it is entered. A `static` local variable retains the last value assigned to it whether or not program execution is in its block.

When applied at the global level, a `static` specification limits the scope of a global variable or subordinate function to the source file in which it is defined. This serves the purpose of allowing a pre-compiled library function to define global variables and subordinate functions whose names can be duplicated in a calling program without incurring error.

A small example illustrating initialization of global and local variables follows:

```
int cnt=50; // global declaration of cnt as integer; initial value 50
double tmp=100.123; // global declaration of tmp as floating point; initial value 100.123
int function_example()
{
    int x; // local declaration of x as integer; initial value indeterminate
    double y=tmp; // local declaration of y as floating point; initial value global tmp
}
```

To recap, local variables are not initialized until the function containing them is executed. The initial value of global variables are part of the compiled version of the program, so global variables revert to their initialized value whenever the program is run.

Constants

Integer (`int`) Constants

For most purposes, integer constants are defined using decimal (base 10) integer format. An integer of type `int` is normally stored in memory as a 32-bit 2's complement integer, which allows values in the range $[-2^{31}, 2^{31}-1]$. Integer constants can also be defined using hexadecimal (base 16) integer format (marked by the prefix "`0x`"), or octal (base 8) integer format (marked by the prefix "`0`"), although both are awkward for representing negative numbers. For 32-bit two's complement integers, the following are true:

$4053_{10} = FD5_{16} = 7725_8$
$-1_{10} = FFFFFFFF_{16} = 37777777777_8$.

In C notation, these comparisons become

```
4053 == 0xFD5 == 07725
-1 == 0xffffffff == 037777777777
```

which also illustrates each of the three formats. Appending a "u" or "U" to the constant changes its value to the range $[0, 2^{32}-1]$ rather than $[-2^{31}, 2^{31}-1]$. An "unsigned" specification is sometimes needed when data is to be copied to a variable of a different type; in particular, 8-bit bytes are treated as unsigned integers.

Present day computer memory is normally organized in 8-bit bytes, so hexadecimal pairs are commonly used to exhibit the value of bytes in memory. Earlier computer systems used 3-bit groupings which were represented using octal digits, explaining why octal representation is included as a means for representing integers in C.

The number of bits used for representing integers is system dependent, but is typically 32 bits for type `int`. The data type `long int` normally specifies a 64-bit integer, and is designated by appending "L" or "l" to the integer's

representation. For most programming needs, integer type (`int`) constants are sufficient.

Floating Point (`double`) constants

Fractions are an extension of the integers, and may be represented using a format termed scientific (mantissa, exponent) notation; i.e.,

$$0.\langle mantissa\rangle \times 10^{\langle exponent\rangle}$$

In addition to providing means for representing nominal fractional values, this format allows reasonable approximation of astronomically large or infinitesimally small numbers, emphasizing magnitude more so than exactness. Since the decimal point for a number such as 123.4567 is floated to the front in setting the mantissa and exponent (0.1234567×10^3), the term floating point representation is used.

Computer arithmetic is binary rather than decimal, so the exponent and mantissa are represented internally as base 2 values. Note that the terminating decimal fraction 0.1_{10} when represented as a binary fraction is non-terminating ($0.00011001100110011\ ..._2$). This illustrates that even in simple cases a floating point value may be an approximation, where precision is improved by allowing more bits for the mantissa (double precision doubles the number of bits used for the representation from 32 to 64, adding extra bits for both mantissa and exponent).

A floating point constant can be defined using either ordinary decimal point representation or a form of (base 10) scientific notation; for example,

```
123.4567 == 12.34567E1 == 1234567e-4 == .1234567e3
```

Since the KIPR Robot Controller does not have floating point operations integrated into its CPU, floating point operations are handled by software, making them significantly slower than integer operations (although still fast in human terms). Hence, floating point should only be used for data that is inherently fractional.

Characters and String Constants

A character constant is given by enclosing the character in single quote marks; e.g., `'K'`. Characters are internally encoded in 8-bit bytes using ASCII representation (e.g., the internal ASCII representation of `'K'` is the (8-bit) hex pair "4B").

A character string constant is a sequence of characters enclosed in quotation marks, e.g., `"This is a character string."`. C processes a character string as (1-dimensional) array delimited by the (unprintable) character constant `'\0'`, which marks the end of the string within the array.

The character constant `'K'` cannot be used interchangeably with the string constant `"K"` since `'K'` is an 8-bit integer and `"K"` is a 16-bit string whose first 8-bits are the 8-bits for `'K'` and the second 8-bits are the 8-bits for `'\0'`.

NULL

The special constant `NULL` (a preprocessor macro included implicitly, or explicitly using `#include <stio.h>`) is provided by the C preprocessor to represent a NULL pointer. In general, a pointer represents the location (or address) of a data structure in memory. A NULL pointer is one which exists, but which points to nothing.

A pointer that hasn't been initialized has no semantic meaning, in contrast to a pointer initialized to `NULL`, which points to nothing (think in terms of the empty set used in mathematics). To check if a pointer variable is pointing to data you compare its value to `NULL`. As an illustration, suppose a linked list data type is defined to have elements with 2 components, the first of which provides a data value and the second of which is a pointer

to the next logical element in the list; e.g., the components for an element in the list might represent a name, and a pointer to the element containing the next name in the list in alphabetical order. If the list is processed to retrieve the name data in alphabetical order, the last element in the list will need to have `NULL` assigned to its pointer since there are no more names! `NULL` then provides a (testable) pointer component value that identifies the last element in the list.

Data Types

C supports the following data types among others:

32-bit Integers

32-bit integers are signified by the data type indicator `int`. They are signed integers in the (decimal) range -2,147,483,648 to +2,147,483,647.

64-bit Floating Point Numbers

Floating point numbers are best specified by the data type indicator `double`. 64-bit floating point numbers have at least 15 decimal digits of precision and range from about 10^{-308} to 10^{308}.

8-bit Characters

Characters are 8-bit (unsigned) integers signified by the data type indicator `char`. A character's value normally represents a standard ASCII character code, most of which are printable.

Pointers

A C pointer is a 32-bit number representing the address of a byte location in memory. A pointer that represents the location in memory where information is stored can be used to manipulate the data by performing calculations on the pointer, passing the pointer, or dereferencing the pointer (deference means to obtain the value stored at the location).

Arrays

An array is a data structure used to store a sequence of homogeneous data elements (each element of the array must be of the same data type). Every array has a length which is determined at the time the array is declared. The location of an element in an array is given by supplying its index in brackets. For example, `myarray[3]` references the fourth element in an array named `myarray` where indexing starts from 0. By providing the index, data may be stored in or retrieved from the array in the same manner as for other variables. By specifying an array whose elements are (same-sized) arrays, an array with more than one dimension can be defined.

Structures

Structures are used to store non-homogenous but related sets of data. Before a structure can be specified, its `struct` data type must be defined. Any available data type can be used in its definition, including those that are user defined. In contrast to arrays, elements of a structure are referenced by name instead of number. For example, if a `struct Triangle` data type is defined as

```
struct Triangle {
    double sideA;
    double sideB;
    double sideC;
};
```

then `struct Triangle x` defines a structure named `x` of data type `struct Triangle`. If side A for triangle `x` is needed, it is referenced by name as `x.sideA`.

Since a function can return a `struct` data type, structures provide a way for a function to return multiple data values. Structures can also be useful for reducing the number of arguments passed to functions. More importantly, structures provide a means for creating complex data structure representations such as directed graphs and linked lists. For these kinds of data structures, the `struct` elements may be dynamically allocated (see `malloc`) or may be taken from an array set up for this purpose that has the same `struct` data type.

Enumerated Sets

Enumerated sets are used to associate names with integer values, by default 0,1,2, ... according to how many entries are in the set. For example,

```
enum letter {F,D,C,B,A};
```

specifies an enumerated data type named `letter` with associated names F,D,C,B,A which by default correspond to the values 0,1,2,3,4.

For a program, the declaration

```
enum letter grade;
```

specifies a variable named `grade` of type `enum letter`, to which can be assigned any of the names F,D,C,B,A (or equivalently the integer values 0,1,2,3,4). For example,

```
grade = C; // same as using grade = 2;
```

Enumerated sets are particularly useful for `switch` constructions since cases can be referred to by names rather than values; e.g.,

```
switch(grade) {
   case A:
          // case action for an A grade
          break;
   case B:
          // case action for an B grade
          break;
          ...
```

and so forth.

While enumeration names must be distinct, the associated values do not need to be. Note that in most instances, `#define` could be used in the same manner intended for enumerated sets, but enumerated sets have the advantage of the actual values being generated automatically by default. It also needs to be noted that no check is made to see if a value assigned to an enumeration variable is valid, which means the program needs to ensure the validity. An invalid value may produce an unpredictable outcome; for example, assigning a negative value to the `grade` variable above may produce an invalid outcome if the variable is being used in a comparison.

Pointers

A pointer is the (numeric) address of the location in memory where a data element is stored. Memory addresses begin at 0 and increase by 1 for each byte of memory. Limited arithmetic operations may be performed on pointers, but the value of the resulting pointer depends on the data type pointed to. For example, adding 1 to the pointer for a `char` data item increases the pointer value by 1 since that advances the pointer to the next character in memory. In contrast, adding 1 to an `int` pointer increases its value by 4 since that advances the pointer to the next integer in memory. Using a pointer to try to access memory not allocated to a program will probably cause a system error or crash, so it is important to insure that pointers used in a program address valid objects in memory.

A pointer data type can be defined for any allowed data type, included user-defined data types. When used as a unary operator (one argument), `*` is the indirection (or "value at address") operator, and is used for defining and dereferencing pointer variables. For example,

```
int *p;
```

defines `p` to be a variable whose value is a pointer to a memory location holding an integer.

When used as a unary operator, `&` is the memory address operator. The address of a variable named `x` is given by `&x` and so if `x` is an integer variable,

```
p=&x;
```

assigns the memory address of variable `x` to the (integer) pointer variable `p`.

Retrieving the value pointed to is known as *dereferencing* the pointer and is given by `*p` for pointer variable `p`. For the example above, both variable `x` and `*p` represent the same value in memory. `*(p+1)` retrieves the (integer) value of the next integer in memory (which is 4 bytes further along than `x`). Furthermore, the contents of the address can be changed by assigning a value to the dereferenced address; e.g.,

```
*(p+1) = *p + 1;
```

It is often useful to deal with pointers to objects, but great care must be taken to insure that the pointers used at any point in your code really do point to valid objects in memory. It is easy to get confused by what a pointer is addressing when it is taken from a complex structure with pointers to pointers or something similar.

Pointers are often used instead of global variables to provide a function with access to data external to the function. If a pointer is passed to a function as an argument, the function then has access to the memory location for the pointer, which could also represent a local variable external to the function. If the function uses the pointer to change the value in the memory location, it will also have changed the value of the external local variable.

For C, the scope of function parameters is local to the function, with values assigned to them only when the function is called. Programming languages typically employ one or more evaluation strategies for function parameters. The ones usually cited are *call by value*, *call by reference*, and *call by name*.

- For *call by value*, the parameter is treated as a local variable initialized using the value supplied for the parameter when the function is called. This is the approach used for C.
- For *call by reference*, no local version of the parameter is created, which means it represents storage for the calling agent, even if the supplied parameter is a constant (i.e., a poorly written function could change the value of a constant passed to it by reference, which was a real problem for the early programming language FORTRAN). Although C does not use call by reference in the strict sense, it accomplishes the same objective by use of pointer variables as parameters. When a pointer variable is used as a parameter for a function in C, by dereferencing the parameter (whose scope is local) the function can access and possibly modify the value of an external variable having the same address.
- For *call by name* the actual name supplied as the parameter is substituted throughout the function, much as the C preprocessors does for `#define` macros. *Call by name* is not supported by C in any meaningful sense.

Arrays

Arrays can be defined in C for any supported data type, including user defined data types. The name of an array is actually a pointer to its first element, the one whose index is 0, so passing an array to a function is a call by reference. Multi-dimensional arrays are defined as arrays of arrays (or arrays of pointers). Arrays are useful for allocating space for many instances of a given data type, arranged sequentially in memory, which provides means for iterating over the set of values in the array.

Declaring and Initializing Arrays

The definition for an array specifies its data type, name, and index structure, which is given inside square brackets. The following statement declares an array of ten integers:

```
int ex_array[10];
```

The elements of the array are numbered from 0 to 9. Elements are accessed by enclosing the index number within square brackets; e.g., `ex_array[4]` denotes the fifth element of the array `ex_array` (since counting begins at zero). Note that `ex_array == &ex_array[0]`.

Arrays not initialized at declaration contain indeterminate values. Arrays may be fully or partially initialized at declaration by specifying the array elements within curly braces, separating the array elements from each other by commas. For example,

```
int ex_array[10]= {3, 4, 5, -8, 17, 301};
```

initializes the first six elements of `ex_array`, with `ex_array[4]` equaling `17`, `ex_array[0]` equaling `3`, etc.

If no size value is specified within the square brackets when the array is declared, but initialization information is given, then the size of the array is determined by the number of initialization elements given in the declaration. If a size is specified and initialization data is given, but the length of the initialization data exceeds the specified length for the array, the excess data will be ignored (and the compiler will issue a warning).

Character Strings

Character strings are implemented as arrays of characters. An array of characters can be initialized character by character, but can also be initialized by using a string constant, for example:

```
char ex_string[]= "Hello there";
```

This initializes the character array `ex_string` with the ASCII values of the characters in "`Hello there`" terminated by the (unprintable) ASCII character `'\0'`. The length of the array is 12, which is the number of characters in "`Hello there`" plus 1 for the `'\0'` string termination character. If the array length had been declared, the initialization would leave the balance of the array indeterminate if more than 12 and truncated (including `'\0'`) if less than 12. When a character array containing a string constant is used as an argument for `printf`, the format specifier %s marks where to insert the string in the formatted output; for example,

```
char ans[10]="no";
int q=3;
printf("The answer to %d is %s.\n",q,ans);
```

produces the (formatted) output "`The answer to 3 is no.`".

The standard C library has string functions for assigning strings (`strcpy`), determining string length (`strlen`), combining strings (`strcat`), and the like (look for string functions in your C reference). For example,

```
strcpy(s,"Example string");
```

is used to copy the character string in the second argument of `strcpy` to the character array in the first.

To make sure that the C preprocessor includes the prototype declarations for the string functions, programs that use them normally have the C preprocessor directive

```
#include <string.h>
```

at the start of the program code. This directive, like the `#include <stdio.h>` directive is ignored if it is issued again elsewhere in the program.

When a character array is initialized using the curly braces syntax, unless `'\0'` is included, there is no string delimiter, and printing the array as a character string using `printf` will produce indeterminate results. When declaring a character array that is intended to hold character strings, the array size needs to be at least 1 larger than the size of the maximum string it will be used for to allow for the `'\0'` string termination character. For example, given

```
char pg_string[81];
```

strings of length up to 80 can be stored in the variable `pg_string`

Passing Arrays as Arguments

When an array is passed to a function as an argument, the address of the array's initial element is actually passed, rather than the elements of the array, a call by reference as discussed earlier. Hence there is not a local copy of the array inside the function, and any modifications the function makes to the array are to its location in memory when declared (what is local to the function is the copy of the address of the array's initial element passed as an argument).

For a function to be able to treat an argument as an array, the argument has to specify the array's data type and index structure.

As an example, the following function has arguments for an index and an array, printing the array element at the index value.

```
void print_element(int indx, int arr[])
{
    printf("Value at index %d is %d\n", indx, arr[indx]);
}
```

The use of the square brackets specifies that the argument is a 1-dimensional array of integers.

Alternatively, since the array name represents a pointer to the first element in the array, the function argument could be specified as pointer variable, in which case the square brackets would be omitted. In this case, in the body of the function the pointer variable name would be used instead of the array name. For example,

```
void print_element(int indx, int *p)
{
    printf("Value at index %d is %d\n", indx, p[indx]);
    // or you could use printf("Value at index %d is %d\n", indx, *(p+indx));
}
```

Either of the following two example calls to the function will work, whichever of the two versions is used:

```
print_element(3, ex_array);
```

```
print_element(4, &ex_array[0]);
```

Multi-dimensional Arrays

A two-dimensional array is just one-dimensional array whose elements are one-dimensional arrays. For example

```
int k[2][3];
```

specifies an array of length 2 whose objects are length 3 arrays integers. `k` can be viewed as a two-dimensional array with 2 rows and 3 columns, where the first row has as elements `k[0][0], k[0][1], k[0][2]` and the second the elements `k[1][0], k[1][1], k[1][2]`. `k[0]` and `k[1]` represent rows of 3 elements each. Hence, in addition to accessing any element of the array using index bracket notation, any row in the array can be accessed similarly. Arrays with any number of dimensions can be generalized from this example by adding more brackets in the array declaration.

The initialization

```
int k[2][3] = {{0,1},{3,4,5}};
```

illustrates initialization of parts of the array, where only `k[0][2]` has not been initialized. Conceptually the array is

```
0 1 ?
3 4 5
```

Structures

As noted above, structures are used to store non-homogenous but related sets of data. In order to specify a structure, a `struct` data type for it must first be defined. Any available data type can be used in the structure definition, including those that are user defined. The elements of a structure are referenced by name to access them.

Since a function can return a `struct` data type, structures also provide a way for a function to return multiple (named) data values. However, the primary purpose of structures is to provide a means for creating complex data structure representations such as directed graphs and linked lists. In the construction of this kind of data structure, structure elements may be dynamically allocated (see `malloc`) or alternatively, be taken from an array having the `struct` data type.

The following example illustrates structure definition, creation of an array of structures, and access to its elements.

```
#include <string.h> // make sure the string function declarations are present
struct ex_stype { // structure definition by struct data type specification
    int i;      // component is an integer named i
    char s[81]; // component is a string named s
};
void set_ds(int x, char nm[], int i); // function to assign values for the s and i components in
structure ds[x]
void show_ds(int x); // function to display the elements of structure ds[x]
struct ex_stype ds[2]; // ds is an array each element of which is an ex_stype structure
int main()
{
    set_ds(0,"one",1); // assign values to the components of structures ds[0] and ds[1]
    set_ds(1,"two",2);
```

```
    show_ds(0);         // display the components of structures ds[0] and ds[1]
    show_ds(1);
}
void set_ds(int x, char nm[], int i)
{
    ds[x].i=i;              // copy argument i to the i component of structure ds[x]
    strcpy (ds[x].s, nm); // use the string library function strcpy to copy argument nm to the s
component of structure ds[x]
}
void show_ds(int x)
{
    printf("name %d is %s\n",ds[x].i,ds[x].s);
}
```

The first part of the example is the specification of the `struct` data type used to define the data structure `ds` used in the program. The `struct ex_stype` data type specifies a structure with an integer component `i` and a string component `s`.

A global variable `ds` is declared as a 2 element array of data type `struct ex_stype` in the same manner that would be used for any other data type.

The two functions, `set_ds` and `show_ds`, illustrate using the dot operator (`.`) to assign values to the components of a structure and to access the components of a structure, respectively. For the declaration of variable `sx`

```
struct ex_stype sx;
```

the components of the structure `sx` are `sx.i` and `sx.s`. Similarly, for the structure in item x of array `ds`, the components are `ds[x].i` and `ds[x].s`, where the index in square brackets has to be added to identify which of the structures in the array is being accessed.

Pointers to `struct` data types can also be used, just like pointers to any other type. When a pointer is for a `struct` data type, an "arrow" (`->`) notation can be used to access elements of the structure addressed by the pointer; for example,

```
struct ex_stype *sptr;
sptr = &dx;
sptr->i = 10;
strcpy(sptr->s, "example text");
```

The dot operator can be equivalently used, but has the clumsier notation `(*sptr).i` as opposed to `sptr->i`.

Just as for arrays, only pointers to structures, not the structures themselves, can be passed to or returned from functions.

Initialization Examples

Complex data structures formed as arrays or as structures may be initialized upon declaration with a sequence of constant values contained within curly braces and separated by commas.

Character arrays may also be initialized as a string from a string of characters enclosed in (double) quote marks.

In initialization of a one dimensional array, the length (inside the square brackets) can be left blank, in which case

the allocated length is determined by what is assigned to the array. The declaration for a multi-dimensional array must specify the size of all dimensions after the first dimension. If a length is specified, and initialization data overflows that length, a warning is issued and the excess data is ignored. If the initialization data falls short of the specified size the rest will be indeterminate.

The following example illustrates a variety of different initializations:

```
/* declare global variables of various types */
int i = 50; // single basic variable
int *ptr = NULL; // single basic pointer variable
int x=1, y, z=3; // multiple basic variables
double farr[3]= {1.2, 3.6, 7.4}; // one dimensional array
int iarr[5]= {1, 3, 7}; // one dimensional array, last 2 indeterminate
int jarr[]= {2, 4, 6, 8}; // one dimensional array, derived length 4
char carr[2][3]={{'a', 'b', 'c'}, {'b', 'd', 'f'}}; // two dimensional array
int xarr[2][5]={{1, 2, 3}, {2, 4, 6}}; // two dimensional array, last 3 rows indeterminate
int yarr[][2]={{1, 2}, {2, 4}, {3,6}}; // two dimensional array, derived size 3x2
char c[]="A B C"; // string, array size 6=5+1
char sarr[5][10]={"a b c","d","e f"}; // array of strings, first 6 characters of row 1 initialized,
first 2 of row 2, first 4 of row 3, last 2 rows indeterminate
struct employee {
    char name[31];
    double weight;
    struct employee *nextrec;  // pointer (self-referential) to the struct for another employee
}
struct employee a_node = {"John Doe", 165.4, NULL}; // perhaps a linked list node
struct elist[2] =
    {{"JFK", 163.1, NULL }, {"LBJ"}}; // not everything has to be initialized
// example function which declares local variables of various types, initialized from global
variables
int f_locals()
{
    int x = i; // local x, global i (value 50)
    int y = yarr[2][1]; // local y, global yarr (value 6)
    int *iptr=&i; // local iptr, address of global i
    struct employee wx={"Jane Doe", 115.2, &a_node}; // local struct wx, pointed to global a_node
    char lc[]=sarr[2]; //  local string lc with initial value "e f" copied from length 4 string
stored in sarr[2]
    . . .
}
```

Statements and Expressions

C has unary operations (one argument), binary operations (2 arguments), and even a ternary operation (3 arguments) for performing actions upon data used in a program. Expressions are combinations using one or more operations. Statements incorporate expressions, assignments, function calls, and control flow constructions to form blocks of code for a C program and are terminated by a semi-colon (;).

Operators

Each of the data types has a set of operators for operations that may be performed on expressions for that data type (e.g., `x + y` is an expression).

Integer Operators

The following operators are provided for integer data:

- **Binary**
  - **Arithmetic**: addition `+`, subtraction `-`, multiplication `*`, division `/`.
  - **Relational (comparison)**: greater-than `>`, less-than `<`, equality `==`, greater-than-equal `>=`, less-than-equal `<=`, modulus `%`.
  - **Logical (Boolean)**: logical-OR `||`, logical-AND `&&`, logical-NOT `!`.
  - **Bitwise**: bitwise-AND `&`, bitwise-OR `|`, bitwise-exclusive-OR `^`, bitwise-NOT (one's complement) `~`, left shift `<<`, right shift `>>`.
- **Unary**
  - integer promotion `+`, additive inverse `-`, increment (prefix or postfix) `++`, decrement (prefix or postfix) `--`

In C, an integer used in a Boolean expression represents FALSE if it is 0 and TRUE if it is non-zero. In determining the result of the logical operation `x && y`, if x is FALSE, the value returned by the operation is FALSE without checking y since logically this is enough to determine the result has to be FALSE. Similarly, for `x || y`, if x is TRUE, the value returned by the operation is TRUE without checking y since logically that is sufficient for determining the result is TRUE. In all other cases, the value for both operands is checked. This can be useful if x is being used as a TRUE/FALSE flag where y may initially be invalid when x is FALSE.

Integer promotion is the action of "promoting" an arithmetic operand to a 32 bit integer, similar to type casting or coercion. Unlike its counterpart, unary `-`, it is seldom used.

Floating Point Operators

The following subset of integer operators are provided for floating point data:

- **Arithmetic**: addition `+`, subtraction `-`, multiplication `*`, division `/`.
- **Relational (comparison)**: greater-than `>`, less-than `<`, equality `==`, greater-than-equal `>=`, less-than-equal `<=`.

For the KIPR Robot Controller, floating point operations are implemented in software, which means they are significantly more resource intensive than integer operations. For this reason, floating point values should not be substituted for integers in iterative processes (such as a loop counter).

There is a large selection of math functions in the C library for performing calculations that are inherently floating point. For example,

```
double a, x,y;
x = sqrt(2); // x is assigned the square of 2
x = log(5);  // x is assigned the base e logarithm of 5
x = log2(8); // x is assigned the base 2 logarithm of 5
x = exp(3);  // x is assigned e cubed
x = exp2(3); // x is assigned 2 cubed
y = pow(x,0.333333); // y is assigned the (approximate) cube root of x
```

```
x = sin(3);  // x is assigned the sine of 3 (radians)
a = asin(0.123); // a is assigned the primary angle (in radians) having sine 0.123
```

There are many more functions in each of these categories, plus functions for other kinds of manipulation of floating point data. The section below for KIPR Robot Controller Library Functions includes many of these under the "Math" category designation.

Characters

For C, a character is an 8-bit unsigned integer whose value is one of the 256 ASCII character codes (ASCII is the acronym for *American Standard Code for Information Interchange*). Integer operations can be performed on character data since if a character variable is used in an integer operation, it is automatically coerced from an 8-bit unsigned integer into a (positive) 32-bit integer for the computation. ASCII encodes 0 ... 9 in order, then later A ... Z in order, and then later a ... z in order, to enable easy alphabetizing using integer comparison. Adding 32 to an upper case character will convert it to lower case.

When a value is stored into a character variable, it is coerced into an 8-bit character (by truncating the upper bits). Since character data occurs in 8-bit bytes, character string data is stored in consecutive byte locations in memory; i.e., treating memory as character data provides a means to address and step through memory one byte at a time.

Compound Assignment Operators

The basic assignment operator is =. The following statement adds 2 to the value of a.

```
a = a + 2;
```

The compound assignment operator += does exactly the same thing; i.e.,

```
a += 2;
```

All of the following binary operators can be used in a compound assignment:

```
+,  -,  *,  /,  %,  <<,  >>,  &,  ^,  |
```

Unary Increment and Decrement

The increment operator ++ increments its operand by 1. When used prefix (`++x`) it has a different semantic interpretation then when used postfix (`x++`). The simple cases

```
x++;
++x;
x = x + 1;
x+=1;
```

are all equivalent.

When used in a more complex expression, the value of `++x` and `x` change *before* the result is used in the expression. For `x++`, the value of `x++` and `x` change *after* the result is used in the expression. For example, for the following constructions using `++x` and `x++`

```
x = 3; printf("%d  %d\n", x, ++x);
```

```
x = 3; printf("%d  %d\n", ++x, x);
x = 3; printf("%d  %d\n", x, x++);
x = 3; printf("%d  %d\n", x++, x);
x = 3; printf("%d  %d\n", x, x+1);
```

the displayed text will be

```
4 4
4 4
4 3
3 3
3 4
```

Confusing? To understand this you also have to know the arguments of are evaluated from right to left in setting values passed. Here's a (long-winded) explanation for how the outcome above is produced by each program line:

Line 1: The `++x` argument is cleared before values are passed to `printf` and by incrementing beforehand both argument values are set to 4.
Line 2: Just as for Line 1, the `++x` argument is cleared before values are passed to `printf`.
Line 3: The `x++` argument is rightmost and gets the current value 3 of `x` because the increment occurs afterwards. The new value of `x` (4) then becomes the value used for the `x` argument.
Line 4: The `x` argument is rightmost and has value 3. The `x++` argument sets the value 3 before it increments. The value of x subsequent to the `printf` is 4.
Line 5: This is the most intuitive case, because it doesn't use ++, but notice it also doesn't increment x.

The difficulty for explaining what appears at first glance to be straight forward code illustrates why the use of these operators in normally limited to the simpler cases.

The interpretation for a combination such as `x+++y` is dependent on the compiler being used so such variations are avoided, although not prohibited. [*So which is it? `(x++)+y` or `x+(++y)`? The answer could be neither!*]

The decrement operator `--` decrements its operation by 1. Its semantics mirror those of `++`.

Data Access Operators

Data access operators are ones used to manipulate components of arrays and data structures, or to directly manipulate data via memory addresses.

- The unary reference operator (`&`) returns the memory address pointing to a variable, array component, structure component, or function regardless of its data type.
- The unary dereferencing (or indirection) operator (`*`) applied to a pointer accesses the referenced address as an object for the data type of the pointer, either for assigning a value to it, or for utilizing its value.
- The array subscript (`[ ]`) brackets operator accesses the component of the array given by the index enclosed within the brackets.
- The structure component (`.`) dot operator accesses, by name, the specified component.
- The structure dereference (`->`) arrow operator applied to a pointer to the structure accesses the referenced component (by name) of a structure (for structure `ds` with component named component `i`, `ds.i` and `(&ds)->i` are equivalent.

Precedence and Order of Evaluation

As per Kernighan and Ritchie's book, the following table summarizes the rules for precedence and associativity for the C operators. Operators listed earlier in the table have higher precedence; operators on the same line of the table have equal precedence and when used without parentheses are cleared left to right.

Operator	Associativity
`()  []  ->  .`	left to right
`!  ~  ++  --  -  *  &  (`*<data-type>*`)  sizeof`	right to left
`*  /  %`	left to right
`+  -`	left to right
`<<  >>`	left to right
`<  <=  >  >=`	left to right
`==  !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`=  +=  -=  *=  /=  %=  &=  ^=  |=  <<=  >>=`	right to left
`,`	left to right

Unary `+, -,` and `*` have higher precedence than their binary forms.

Control Flow

The order in which a computer executes machine-level operations stored in memory is normally sequential, with the exception of those operations which redirect the flow of executrion to continue elsewhere in computer memory. This capability is reflected in C programs by control flow statements which serve to redirect the sequential step by step flow the program would follow otherwise. There are two classes of control flow commands, *selection* and *iteration*. Selection commands use a condition test to determine what statement or block of statements to execute next (`if-else`, `switch`, and the ternary operator `?:`). Iteration commands use a condition test to determine how many times to repeat the execution of a statement or block of statements (`while`, `for`, `do-while`). C also provides means for exiting a control flow program structure independent of the condition test (`break`) and to skip the rest of a block, continuing to the next iteration (`continue`).

Statements and Blocks

Each C statement is terminated by a semicolon. A program block is a sequence of statements grouped together inside curly braces. Variables may be defined inside a block and are local to the block (their scope does not extend outside of the block). The object for a control flow statement may be a single statement or a block of statements, except for `?:`, which as an operator applies to expressions rather than statements.

If-Else

The `if` statement is a selection statement for making yes/no and either/or decisions, For an either/or decision, the `if` statement is paired with an `else` statement (which can only be used in a pairing with `if`).

The syntax for `if` is

```
if (<expression>)
    <statement or block>
```

When paired with `else` the syntax is

```
if (<expression>)
    <statement or block>
else
    <statement or block>
```

When the <*expression*> evaluates as TRUE (i.e., is not equal to zero), then the <*statement or block*> that follows `if` is executed.

When the <*expression*> evaluates as FALSE (i.e., is equal to 0), then the <*statement or block*> that follows `if` is not executed and if there is an `else` paired with `if`, then the <*statement or block*>; that follows `else` is executed.

In effect, these constructions use syntax that corresponds to how similar logic is expressed in English, such as

- "If the fever is over 100F then schedule a doctor's appointment" (yes/no or do/don't do)
- "If the manuscript is acceptable then pay for it, otherwise return it to the author" (either/or)

Ternary Operator `?:`

The ternary operator `?:` provides a compact means for expressing if-else logic where assignment statements are employed as the if-else actions. For example, the following `if-else` construction assigns the larger of variables `x` and `y` to variable `z`:

```
if (x > y)
    z = x;
else
    z = y;
```

This same construction can be implemented by using the ternary operator `?:`

```
z = (x > y) ? x : y;
```

The ternary expression is formed from a condition test (`x < y`) followed by a question mark (`?`), then the expression (`x`) to assign to `z` if the condition evaluates as TRUE, separated by a colon (`:`) and then the expression (`y`) to assign to `z` if the condition evaluates as FALSE. Strictly speaking, given the low precedence level of the operator, the parentheses around the condition could be omitted, but are advisable to facilitate identifying the operator's three operands.

ANSI C requires that only the expression whose value is to be assigned to z be evaluated, just as in the `if-else` equivalent. The ternary operator is frequently used for obtaining absolute value; e.g.,

```
absx = (x < 0) ? -x : x;
```

While

The `while` statement in C is the iteration statement most commonly used for managing program loops. A *loop* is a sequence of program logic that is repeated multiple times (0 or more); for example,, a loop might be used to print out the elapsed time second by second while waiting for a button to be pressed.

The syntax for a `while` statement loop is

```
while (<expression>)
    <statement or block>
```

When the <*expression*> evaluates as TRUE (i.e., is not equal to zero), then the <*statement or block*> that follows `while` is executed, after which the process repeats, starting with evaluation of the <*expression*> again.

When the <*expression*> evaluates as FALSE (i.e., is equal to 0), then the <*statement or block*> that follows `while` is not executed and the program moves on to the statement following the <*statement or block*> for the `while`.

Note that the loop continues until something happens that causes the <*expression*> to evaluate as FALSE (e.g., a button is pressed or a counting variable has exceeded its limit). It is up to the programmer to ensure that within the body of the loop (or otherwise) the condition will eventually be FALSE (otherwise the loop will continue until the program is halted by external intervention).

A loop that doesn't terminate is termed an *infinite loop*. An infinite loop is one that will continue its iteration until the program is halted. An *indefinite loop* is one whose iteration will continue until some external action occurs (such as a button press). It is easy to create an infinite loop using `while`. For example,

```
while(1);
```

forms an infinite loop because the <*expression*> for the `while` is 1 which is always TRUE, and the <*statement or block*> is empty (`;`), providing no means other than a forced program halt to end the ongoing loop iteration (which is repeatedly doing nothing).

A common error to avoid when programming a `while` statement loop is exemplified by the following:

```
while (i < 10); // misplaced semi-colon
    x[i++] = i;
```

In this case, the misplaced semi-colon for the `while` has made its <*statement or block*> component empty. Assuming `i` is less than 10 when the loop starts, this semantic error makes the loop infinite and the program will "hang" at this point, requiring a forced halt to end it. For ideas on how to locate this kind of semantic error for a program, see the section below on program debugging.

Break

The `break` statement in C provides a means for a program to break out of executing statements within a loop structure or a multi-way selection to proceed with execution of the program statements that come next. A condition test for `break` is sometimes used within what would otherwise be an infinite loop to break out of it. The `break` statement applies to `while`, `for`, `do-while`, and `switch`.

For example,

```
int i; // counter will need some initial value
while (i < 100) { // has the counter reached its limit?
    printf("cnt = %d\n", i++); // show the counter and increment it
    if (side_button() != 0) break; // exit if the side_button is pressed
    msleep(1000); // pause for one second
}
```

uses a `break` to exit the `while` loop.

Continue

The `continue` statement in C provides a means for a program in a `while`, `for`, or `do-while` loop to skip the balance of the current iteration, continuing to the next iteration if the loop condition test doesn't end the loop.

For example,

```
int i; // counter will need some initial value
while (i < 100) { // has the counter reached its limit?
    if (i <= 0) { // don't start cnt until i reaches 1
       i+=1;
       continue;
    }
    printf("cnt = %d\n", i++); // show the counter and increment it
    if (side_button() != 0) break; // exit if the side_button is pressed
    msleep(1000); // pause for one second
}
```

Return

The `return` statement in C both ends a function and provides a means for the function to return a value to the calling function. A function also ends once its terminating brace is reached or if a `return` statement having no return value is executed. The calling function can ignore a returned value in any case.

For

The `for` statement in C is normally used for managing loops that employ an iteration counter. A `while` statement loop could also be used for this purpose, but in many cases the program logic is clearer if a `for` statement is employed to control the loop.

The syntax for a `for` statement loop is

```
for (<expr-1>;<expr-2>;<expr-3>)
    <statement or block>
```

The behavior of the `for` statement loop is equivalent to that of the following `while` statement loop:

```
<expr-1>; // initialize the counter (prime the pump)
while (<expr-2>) { // has the counter reached its limit? (have we pumped enough in?)
    <statement or block>
    <expr-3>; // increment the counter (pump some more in)
}
```

For example, the following code counts assigns values from 0 to 99 to an integer array

```
int i, arr[100];
for (i = 0; i < 100; i++)
    arr[i] = i;
```

Switch-Case

The `switch` selection statement for C is used to select one of a series of `case` targets. The `case` targets for a `switch` statement are grouped together inside curly braces as a program block. Each `case` target is followed by a series of C statements which represent that case. The `switch` selection in effect skips past cases until it reaches the one selected, and program execution continues from there. If a `break` statement is in the statements following a `case` target, then the program exits the `switch` statement's block at that point, in effect skipping any remaining cases.

`if else` selection is used to select one of two cases. `switch` selection is used to select 0 or more cases.

The syntax for a `switch` statement is as follows:

```
switch (<expression>) { // switch to the case target whose <constant> matches the value of the
<expression>
case <constant>:
    <0 or more C statements>
    break; // optional - otherwise, program execution continues with the next case

<additional case targets>

default: // optional target used when selection didn't match any case target
    <0 or more C statements>
    break; // optional - otherwise, program execution continues with the next case
}
```

The first `case` target whose <*constant*> matches the value of the `expression` for the `switch` statement is the one selected. Program execution continues from the selected `case` target until either a `break` statement is encountered, or the end of the `switch` program block is reached. If there is no `case` match, then the `default` target is selected. Absent a `default` target and a `case` match, the `switch` program block is skipped. Putting in a `default` case with a `break` statement is considered to be good form, and recognizes the likelihood of more cases being added under program maintenance.

Normally, each `case` is ended with a `break` statement since the primary purpose of the `switch` statement is to make the choice of one of several possible cases to execute; for example,

```
char answer[81];
switch(answer[0])
{
    case 'y':
        printf("answer was yes\n");
        break;
    case 'n':
        printf("answer was no\n");
        break;
    default:
        printf("answer was not understood\n");
}
```

If user input "yes sir" is captured in the character string `answer`, then the text "answer was yes" will be printed to the screen. If if was "no sir", then the "answer was no" would have been printed, and if it was "No sir", the

text "answer was not understood" would have been printed (since C is case sensitive, 'n' ≠ 'N').

Do-While

The `do-while` statement in C is less commonly used for controlling loop iteration than the `while` and `for` statements.

The syntax of a `do-while` statement is as follows:

```
do
    <statement or block>
while (<expression>);
```

In contrast to the `while` and `for` statements, the body of the loop for a `do-while` statement will be executed at least once since the <*expression*> controlling loop iteration is evaluated at the end of the loop rather than at its beginning. Sometimes program logic for a loop is best expressed by having the condition test occur at the bottom of the loop. For example,

```
do
    printf(".");
while (seconds() < 5);
```

The logic insures that "." will be printed at least once (so something will always be printed), in contrast to

```
while (seconds() < 5)
    printf(".");
```

for which nothing will be printed if the value returned by the function `seconds` already exceeds 5.

Programming Style

Many authors have expressed opinions on programming style, which for C generally consists of how key elements of a program are exhibited, how comments are handled, and how white space is employed, most notably in how block structures are offset by placement and indentation. The manner in which a program is written affects its readability, ease of understanding, and effort for debugging, enhancement, and maintenance.

The most commonly used style, K&R Style, reflects how programs are presented in Kernighan and Ritchie's book on C (The C Programming Language (2nd Edition)). This style, or a variant, is advocated by most C programming experts and is the style observed in this manual.

White Space

As noted earlier, the term "white space" references combinations of characters that when printed produce a blank area on the display. For C, white space is formed by using characters such as spaces, new lines, and tabs. The white space characters in C are

Character	Print Action
' '	space

'\n'	new line
'\t'	horizontal tab
'\v'	vertical tab
'\r'	carriage return
'\f'	formfeed

The print result may differ from one display device to another (for example, while '\n' has the same outcome for almost every device, '\r' may behave the same as '\n' for some devices but on others simply reposition printing at the start of the current line).

The C compiler either ignores white space that is not imbedded inside double quote marks, or collapses it to a single generic white space character for compilation purposes. This means that other than for character strings, white space can be used for things like indentation that improve program readability. White space is required where it is needed to separate adjacent objects, but if another separator such as a parenthesis, comma, curly brace, square bracket, semi-colon, etc is present, white space is not required.

Indentation

Good use of indentation offsets program blocks, not only making a program more easily understood, but making it easier to determine if the curly braces are matched and block structures will produce the desired semantic results. From the C compiler's point of view,

```
/* Simple example: C Programmer's Manual */
int main()
{
    printf("Hello, world!\n"); /* simple example */
}
```

and

```
/* Simple example: C Programmer's Manual */int main(){printf("Hello, world!\n");} /* simple example
*/
```

are equivalent, but it is pretty clear the use of white space to set off the elements of the program in the first version makes it more understandable than for the second. This distinction becomes even more important when nested selection and iteration statements are being employed. For an admittedly extreme example, the following construction as presented will compile, but from a human readability point of view is almost incomprehensible:

```
switch(i){case 0:while(j++<100){k++;if(k>10)break;}break;case 1:if(j>50)k--;else while(j--
>=0)k++;break;}
```

In addition to better presenting overall program logic, use of a consistent programming style facilitates debugging programs. Just as writers develop styles intended to make the text they produce more easily read and understood, programmers develop styles for making their programs easier to read and understand, but with the added objective of making them easier to debug. The KISS IDE program editor provides facilities for automatic indentation that reflects common practice for C programmers.

Commenting

Comments are the means for programmers to integrate documentation into a program. The lead comment typically

specifies the purpose of the program. For large projects, it is a common practice to also include a (bracketed) comment on program history, in particular reflecting the changes that have been made to the program, including by whom, when, and where in the program.

Multi-line comments are typically used to offset description and purpose of major program components. Single-line comments are useful for provide an explanation of logic or reasoning that might be subsequently useful if that part of the program needs to be revisited for debugging or other purposes.

Program Debugging

Debugging is the process of correcting syntactic and semantic errors found in a program.

Syntax Errors

The C compiler cannot compile a program that has syntax errors such as a missing semi-colon or other program construction error. Syntax errors are the ones most easily corrected since the compiler can identify both where they occur and what the problem is. The KISS IDE lists any syntax errors found during program compilation in an error panel below the program code. The interface provides the line number and position within the line identifying the location of the error, with a description of the cause of the error. Since the mistake causing the first error listed will typically cause additional syntax errors within the program, it is often the case that correcting the syntax causing the first error will fix the rest.

Semantic Errors

Just because a program compiles does not mean it will do what is intended, which is termed program semantics. A semantic error manifests itself during program execution, either because the program crashes or it produces erroneous results. Semantic errors are caused by errors in program logic or programmer oversights (such as failure to initialize a variable). They can be difficult to track down because it usually is not clear where in program execution the error occurred.

For a program which relies on data inputs, determining the presence of semantic errors requires testing the program for a representative set of possible data inputs, including so-called *boundary conditions*. For example, if a data input has a range from 0 to 1000, then 0 and 1000 are the boundaries for the data, and one of these, 0, will cause the program to crash if used in the denominator of a division operation. That means that once the part of the program where the error occurs has been located, program logic has to be adjusted to check the data input to see if it is 0 before using it in the division. For complex systems, program testing is an ongoing part of the software development cycle.

Semantic errors that are rare events are difficult to correct, since being able to repeat what causes the error is key to being able to resolve it. Once a semantic error can be reliably repeated, various techniques are employed to determine the cause and to locate where in the program it occurred. Only the basic techniques will be covered here, but operating systems provide debugging tools designed to determine the cause of an error and to facilitate locating where in the program code it occurred.

Using printf and Commenting for Diagnostic Purposes

For the smaller C programs constructed for environments like the KIPR Robot Controller, most semantic errors that occur can be corrected quickly using one or both of the following two approaches.

- For a C program that runs, but produces erroneous output, it is often useful to insert `printf` statements at various points in the program code to display the value of selected variables to see if their intermediate values are as expected. For example, in an expression containing the sub-expression `(x * y-z)`, displaying the

values of `x, y` and `z` might help discover the sub-expression should have been `(x * (y-z))`.
- For a C program that crashes, "commenting out" sections of code can be used to locate the part of the program triggering the crash. Commenting out is relatively straight forward process to implement, because inserting `//` in front of a line of code turns it into a comment (so it will no longer be executed) and whole blocks of code can be commented out by surrounding them using `/* */` comment bracketing.

  What is commented out has to be selected so that the program will still compile. After commenting out sections of the program, if it still crashes when executed, then what was commented out is probably not where the culprit is located. Including/excluding parts of the code in this manner will usually identify where the problem is occurring so it can be diagnosed and corrected. Sometimes the best tactic is to comment out a large enough section to eliminate the crash, then reduce the scope of commenting out in small steps until the program crash recurs.

For larger programs, a system debugging tool may be called for. Unix systems include the debugging tool `sdb`, which has its own command structure that has to be learned to be used effectively.

Display Screen Printing

The C function `printf` is used for printing to the KIPR Robot Controller display screen, where print output is wrapped to subsequent lines if too long for the display, and is scrolled upward as the capacity of the screen is exceeded. For controlled printing to the display, the KIPR Robot Controller Library additionally includes the `display_printf` version of `printf`, which prints starting at a specified (`column, row`) position on the KIPR Robot Controller display and which doesn't wrap text which exceeds screen width. The number of rows available for controlled printing is different for the 3 button (A,B,C) version of the KIPR Robot Controller program console screen than for the 6 button (X,Y,Z,A,B,C) version.

The syntax of `printf` is the following:

```
printf(<format-string>, ... );
```

The <*format-string*> is a character string which includes 0 or more "%" codes. For each % code, a corresponding argument is supplied to `printf` after the <*format-string*> to provide the value to be formatted and inserted into the print output in place of its % code.

It is important to note that in resolving % codes `printf` ignores the data type of the corresponding argument. In other words, since `printf` does not require a data type match between an argument and its corresponding % code, the argument will not be automatically coerced as it would be for an assignment statement. This is particularly important to keep in mind when both integer and floating point values are being used.

The use of % code formatting is best illustrated by some examples.

Printing Examples

Example 1: Printing a message

A `printf` statement employing no % codes simply prints out the <*format-string*> as a message; for example,

```
printf("Hello, world!\n");
```

The character `\n` at the end of the string signifies an advance to the next (or new) line for any further printing by the program. When the bottom of the display is reached, the display is scrolled (up) for each line subsequently printed.

Example 2: Printing an integer

In most environments, integers in C are stored in memory as 32-bit 2's complement integers. The % code used to format the corresponding argument as a ± decimal integer is "%d" (or %i"). For the following example, the value of the variable `x` is displayed as a decimal integer, with a leading minus sign if the integer is negative:

```
printf("Value is %d\n", x);
```

The code "%d" specifies that the first argument after the initial character string in the argument list for `printf` (the variable `x`) is to be formatted as a ± decimal integer and inserted in place of "%d" in the printed output. The length of the formatted output will vary depending on the number.

Example 3: Other integer print formats

As already seen, the "%d" code is used to format an argument in ± decimal integer form. The "%x" and "%X" codes are used to format an argument in (32 bit) 2's complement form using hexadecimal (hex) digits 0 ..9,A,B,C,D,E,F (each of which represents 4 bits in binary, 0000, 0001, 0010, ..., 1111). Negative 2's complement numbers have a leading 1, so negative 32-bit integers written in hex require 8 hex digits beginning with one of the hex digits 8-F; e.g., -28 when formatted using "%X" is FFFFFFE4. For positive numbers, the space required will vary unless a % code modifier for length is employed; e.g., using the % code "%8X" for the integer 28 will yield " 1C" (8 spaces are consumed).

It needs to be emphasized that a length modifier represents the minimum amount of space that will be used when the formatted result is inserted in the print line. If there isn't enough space, `printf` will use more.

If the code "%08X" is used instead of "%8X", leading 0's will replace any leading spaces (so using %08X" for the integer 28 will yield "0000001C" as the formatted result). This applies to "%d" as well; e.g., formatting the numbers -28 and 28 using "%04d" yields as formatted results "-028" and "0028", respectively. A length specifier is employed when numbers are being printed to line up uniformly in columns. If you always want the sign of the number printed, not just when it is negative, the code "%+d" forces the sign to be printed as + or -.

For example, the `printf` statement

```
printf("Values are %d, %X, %04x\n", -28, -28, 28);
```

displays the text string

```
Values are -28, FFFFFFE4, 001c
```

The third % code in this `printf` ("%04x") has a length specifier (plus a leading 0's specifier) and lower case is used for the hex digits. If the value to be formatted requires more space, `printf` will override the length specifier. It is up to the programmer when using a length specifier to anticipate number size and make the length specifier sufficiently large.

Examples 2 and 3 are representative of output formats for integers using `printf`. There are additional integer output formats (including ones for representing integers using octal digits and for unsigned integers) described in most C references.

Example 4: Printing a floating point number

The % code used to format the corresponding argument as a floating point number is "%f". In formatting for floating point, `printf` rounds the fractional part of the number up according to the number of decimal places used (its precision). The default precision is 6 decimal places. A precision specifier is used to limit this. For example, "%.2f" limits the precision to 2 decimal places. For example,

```
printf("Values are %f, %.2f\n", 1.266, -1.266);
```

displays the text string

```
Values are 1.266000, -1.27
```

The code "%f" specifies that the first argument after the initial character string in the argument list for `printf` (1.266) is to be formatted as a floating point number rounded to 6 decimal places and inserted in place of "%f" in the printed output (since 6 decimal places is sufficient, no rounding occurs). In contrast, the code "%.2f" for the second argument rounds its number (-1.266) to 2 decimal places to yield -1.27.

There are additional floating point % codes described in C references that are used for formatting very large (or very small) floating point numbers in scientific (exponential) notation (±<*mantissa*> E ±<*exponent*>).

Example 5: Printing character strings

The % code "%c" is used to format character data. The % code "%s" is used for character strings, since they are frequently needed for print display.

```
char header[] = "Data: ", cs='a';
int x=28;
printf("%s%c = %d\n", header, cs, x);
```

displays the text string

```
Data: a = 28
```

where for the printed output "%s" is replaced by the character string at the memory location corresponding to `header`, "%c" is replaced by the character given by `cs`, and "%d" is replaced by the ± decimal representation of the data given by `x`.

Example 6: Printing using display_printf

The function `display_printf` is only valide for the KIPR Robot Controller and the Simulator. It is like the standard `printf` function except its first two arguments specify the column and row (zero indexed) on the display where printing starts. The remaining arguments are the same as for `printf`. The following example prints

```
2. KIPR store
```

starting at column 5 on the 3rd row of the display. Note that the first row/first column of the display has column index 0 and row index 0.

```
int i = 2;
char nm[] = "KIPR store";
display_printf(4,2,"%d. %s ",i,nm);
```

The column range for `display_printf` is 0 - 41. The row range for the 3 button (A,B,C) case is 0 - 9 and for the 6 button (X,Y,Z,A,B,C) case is 0 - 7.

`display_printf` does not wrap, truncating strings that go beyond the end of a row on the display.

The special character '`\n`' should be avoided when using `display_printf` since it will have unpredictable effects on what is displayed.

When repeatedly printing variable data to the same place on the display, care must be taken to add enough spaces to what is printed to clear artifacts from the previous print; e.g., if one of the following was used,

```
display_printf(4,2,"%d. %s",i,nm); // NO
display_printf(4,2,"%d. %s ",i,nm); // YES
```

then if we had already printed "2. KIPR store" and changed `nm` to "Staffing" then the first of these would print

```
2. Staffingre
```

whereas the second would add enough spaces on the end to clear the artifacts, printing

```
2. Staffing
```

printf % Codes Summary

% Code	Associated Data Type	Format Outcome
`%d` or `%i`	`int`	± decimal integer
`%x` or `%X`	`int`	2's complement hexadecimal representation
`%f`	`double`	± number with decimal point
`%c`	`char`	ASCII character (low byte for `int`)
`%s`	`char *`	ASCII characters until '\0' is reached

There are additional % codes for `printf` and more complex % code modifiers that can be used in `printf` statements. For information on these, see a standard C reference.

C Preprocessor

Before the C compiler receives a file, it is first passed through the C preprocessor to prepare the file for compilation. The preprocessor clears away comments, shrinks white space to single characters, and processes any preprocessor directives present in the program.

Preprocessor Directives

Preprocessor directives are identified by the "#" symbol, which must be the first entry on a line for the directive to be recognized by the C preprocessor.

The two primary preprocessor directive are `#include` for inserting files into the program code and `#define` for defining macros that are expanded wherever their names appear in program code.

For a macro call to be recognized by the preprocessor, it's definition has to occur at some point earlier in the program. Macro definitions begin with the macro directive `#define` and are usually grouped together at the beginning of the file in which they appear. The preprocessor will flag a macro definition as being a duplicate if its name is reused for another `#define` in the same file. If more than one file is employed for a program, a preprocessor directive applies only within the file where it is defined.

Since the programmer may or may not know what `#define` commands appear in a `#include` file, commands are provided to check to see if a macro has already been defined (`#ifdef .. #endif` and `#ifndef .. #endif`). For example,

```
#ifndef PI
   #define PI 3.1416159
#endif
```

When the preprocessor encounters a valid macro name in the program code, the macro is called and its name "expanded" to be replaced by whatever text the macro generates; e.g., when the macro name PI is encountered by the preprocessor it replaces it with the text 3.1416159 (for the C compiler to subsequently interpret as a floating point constant).

The directive `#undef` removes a macro name from the list of defined macros (and otherwise does nothing). It is sometimes employed defensively to guard against the possibility of `#include` inserting a macro name that conflicts with the name of a function the programmer has defined. The convention employed to minimize the likelihood of this occurrence is to use upper case letters for the names of macros and for nothing else.

The sequential bypass strategy employed by the C preprocessor precludes iterative loop directives, but a limited if-else selection is provided. The directive for this purpose, `#if`, employs a condition test with computation limited to integer constants, character constants, comparisons, arithmetic and logic operators, and macros names (which are expanded before the condition test is calculated). The section of code selected by a `#if` directive consists of the lines of code that follow it, continuing until one of the directives `#else`, `#endif`, or `#elif` is encountered. The section of code selected by a `#if` directive is processed if the preprocessor calculates the value of the condition test to be non-zero; otherwise, the preprocessor omits the section from the code sent to the compiler. The term used to describe this procedure is *conditional compilation* since the condition test determines whether or not a section of code is sent on to the compiler; for example,

```
#define CFLAG 1
#if CFLAG==1
   display_printf(1,3,"On target ");
#else
   printf("On target\n");
#endif
```

selects which form of `printf` to use according to how CFLAG is #defined. CFLAG serves as a "configuration variable" to be set according to the ennvironment where the program will be run.

`#elif` has the same interpretation as "else if" and requires a condition test. The section of code selected by `#elif` is terminated by any one of `#else`, `#endif`, or `#elif` also.

`#else` has the same interpretation as "else", with its section of code terminated only by `#endif`.

The sections of code selected by the two directives `#ifdef` and `#ifndef` discussed earlier can be terminated by `#else` or `#elif` as well as `#endif`.

Preprocessor macros can be used to associate a name with a constant that appears in multiple places in code (e.g., PI or LMOTOR), to simplify a C function call, or to provide a debugging capability that can be turned on or off by use of conditional compilation, among many other possibilities.

Preprocessor Include

The preprocessor `#include` directive is used to insert either a system header file or a user defined header file into program code. System header files typically provide the prototypes for the pre-compiled functions in standard C library. They also provide configuration variables in the form of macros (e.g., NULL). Some programming environments may include the more common system header files automatically, depending on how much control the programmer is expected to exert over the programming environment. System header files incorporate `#ifndef` statements to avoid introducing duplicated definitions, since a system header file may appear in more than one user header file included by a function.

If the file name for a `#include` directive is enclosed in "pointy brackets" (< >) then the preprocessor searches for the file in the system's directory of header files.

If the file name for `#include` directive is enclosed in (double) quote marks, then the preprocessor searches for the file in the user's file space. Unless the file is located along the user's file path, the path information for

locating the file must also be included (e.g., for a USB stick). The contents of the file can be anything, and will be passed through the C preprocessor while being inserted, which will process any preprocessor directives in the file. For user header files, care needs to be taken with any variable declarations incorporated into the file (to avoid duplicates caused by #includes across multiple program files) and in general function definitions should not be imbedded in header files rather than pre-compiled for a user library. To sum up, when `#include` is used with a carelessly prepared user header file, a compiler error such as a duplicated global variable name may occur, or a preprocessor error such as a duplicated macro name may occur.

The following provides an example of each kind of `#include` directive:

```
#include <stdio.h> // insert function prototypes for system I/O functions
#include "mylib.c" // insert my function library
```

A few of the more commonly used system header files providing definitions for pre-compiled functions in the standard C library follow:

`<stdlib.h>`	[numeric conversion, memory allocation, functions such as `rand`]
`<stdio.h>`	[I/O functions such as `printf`]
`<math.h>`	[math functions such as `sqrt`]
`<string.h>`	[string functions such as `strcpy`]
`<time.h>`	[date and time functions such as `time`]
`<stdarg.h>`	[functions with varying number of arguments (i.e., ones using the `...` argument)]

Preprocessor Macros

The `#define` preprocessor directive specifies a macro definition. The macro definition is limited to one line of code (of indefinite length). Once defined the macro will be expanded wherever it occurs in subsequent program code.

Macros are often used to provide replacement text, where the macro provides a more meaningful name; e.g.,

```
#define RIGHT_MOTOR 0 // equate RIGHT_MOTOR with 0
#define POWER 90 // equate POWER with 90
```

If the motor command

```
motor(RIGHT_MOTOR, POWER);
```

is used in subsequent code, the preprocessor will expand the `RIGHT_MOTOR` and `POWER` macros, replace them with 0 and 90, respectively, so the code as prepared for compilation becomes

```
motor(0, 90);
```

Global variables could also be used to provide meaningful names for quantities, but preprocessor macros produce slightly more efficient code. The primary advantage in either case is that if testing shows that the motor port or power needs to be changed, it only needs to be changed at one place in the program.

The definition of a preprocessor macro can also specify one or more arguments to be used in expanding the macro. For example,

```
#define GO_RIGHT(power) motor(RIGHT_MOTOR,power)
```

defines a macro `GO_RIGHT` that takes an argument (`power`) and uses the macro `RIGHT_MOTOR` in its definition.

If `GO_RIGHT(85)` appears in program code, it will expand to `motor(RIGHT_MOTOR(85)` which will then expand to `motor(0,85)` as the code prepared for compilation.

Superficially, the use of a macro that doesn't have arguments looks like a global variable reference. Likewise,

the use of a macro with arguments looks like a call to a function. However, macro expansion is simply one-time text replacement which occurs during preprocessing Compilation resolves global references as memory locations subject to dynamic change during program execution. When called, a function evaluates and interprets its arguments dynamically.

Appropriate use of macros can make it easier to follow the C program logic and can be used to facilitate program testing and modification.

Conditional Compilation

The C preprocessor can be used to select code to be compiled based on logical conditions in preparing a program for compilation. This is called *conditional compilation*. Conditional compilation is used to select the code that is to be incorporated into a program based on a condition test. For example, unless a macro named `DEBUG` has been defined (usually empty; i.e., `#define DEBUG`), the precompiler can omit code whose only purpose is for debugging. In particular,

```
#ifdef DEBUG
  printf("Going Left\n");
  beep();
#endif
```

generates the debugging `printf` and `beep` only if `DEBUG` has been defined, in which case the message "Going Left" will be printed and the KIPR Robot Controller will beep when program execution reaches this part of its code. If `DEBUG` is not defined, the preprocessor will leave the code out and the debugging alert will not occur as the program executes.

Macros can also be conditionally defined; for example, the somewhat more sophisticated debugging macro definition

```
#ifdef DEBUG // if DEBUG is defined, SHOW(printf("%d",i) generates DEBUG code
  #define SHOW(x) printf("DEBUG: "); x // if case: SHOW macro is defined to generate
code
#else // if DEBUG is not defined, SHOW(printf("%d",i) generates nothing
  #define SHOW(x) // else case: SHOW macro is defined to generate nothing
#endif
```

defines a `SHOW` macro in one of two ways, either to produce debugging `printf` statements from its argument, or to produce nothing, regardless of argument.

If `DEBUG` has been defined, then a `printf` debugging statement used as the argument for `SHOW` will be incorporated into the program code. Debugging is activated by adding the `#define DEBUG` directive to the program, and deactivated by commenting it out. In particular, when `DEBUG` has been defined the code

```
SHOW(printf("%d\n",i);)
```

inserted at an appropriate point in the program will be expanded by the preprocessor to produce the code

```
printf("DEBUG: "); printf("i=%d\n",i);)
```

Absent a definition for `DEBUG` the macro `SHOW` will expand to produce no code at all.

If the value of `i` is 8 and `DEBUG` has been defined, the expansion of `SHOW` illustrated above will generate code which outputs the debugging display "`DEBUG: i=8`".

The C Library Math Functions

The standard C library has a large number of precompiled math functions. `#include <math.h>` provides the function prototypes for the math functions that operate on floating point numbers, which is most of them. `#include <stdlib.h>` provides the function prototypes for those which operate on integers. Arguments for trigonometric functions use radian measure for angles rather than degrees (1 degree is $2\pi/360$ radians). The following is a representative list for the available math functions. For more information about what math functions are available, consult a C reference.

`acos` [Category: Math]

Format: `double acos(double x);`
Returns the angle between 0 and π whose cosine is x, where $-1 \le x \le 1$. result is in radians.

`asin` [Category: Math]

Format: `double asin(double x);`
Returns the angle between $-\pi/2$ and $\pi/2$ whose sine is x, where $-1 \le x \le 1$. result is in radians.

`atan` [Category: Math]

Format: `double atan(double x);`
Returns the angle between $-\pi/2$ and $\pi/2$ whose tangent is x. result is in radians.

`cos` [Category: Math]

Format: `double cos(double angle);`
Returns cosine of angle. Angle is specified in radians; result is between -1 and 1.

`exp` [Category: Math]

Format: `double exp(double x);`
Returns e^x (e is the Euler constant, the base for the natural logarithm of a number).

`exp10` [Category: Math]

Format: `double exp10(double x);`
Returns 10^x.

`log` [Category: Math]

Format: `double log(double x);`
Returns $\log_e(x)$, the natural logarithm of x.

`log10` [Category: Math]

Format: `double log10(double x);`
Returns $\log_{10}(x)$, the base 10 logarithm of x.

`pow` [Category: Math]

Format: `double pow(double x, double y);`
Returns x^y.

`rand` [Category: Math]

Format: `int rand();`

Returns a (pseudo) random integer between 0 and a very large integer established on system installation, drawn from the (pseudo) random number stream initiated at start of program execution according to a "seed" value. `rand()%m` restricts the range from 0 to m-1. `srand` is used to re-seed random number generation to vary the random number stream used.

`sin` [Category: Math]

Format: `double sin(double angle);`
Returns the sine of angle. angle is specified in radians; result is between -1 and 1.

`sqrt` [Category: Math]

Format: `double sqrt(double x);`
Returns $\sqrt{x}$, the square root of x.

`srand` [Category: Math]

Format: `void srand(int num);`
Re-seeds (pseudo) random number generation (the initial seed at start of program execution is `srand(1)`). Unless it is desirable that a random number stream be reused, the seed value used should be unpredictable from one run to the next. The usual way to accomplish this is to draw a value from the system clock to use as seed; e.g., `int seednum = time(NULL);`. (the time function is declared in system include file `<time.h>` and if used with a NULL operand returns elapsed time in (whole) seconds since system start).

`tan` [Category: Math]

Format: `double tan(double angle);`
Returns the tangent of angle. angle is specified in radians; result is between $-\infty$ and $+\infty$.

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

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 Robot Controller, 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 Robot Controller, 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 Robot Controller, the directory path to a mounted Flash drive in a USB port is

```
/kovan/media/sda1/
```

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 Robot Controller. 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.

```
#include <stdio.h>  // make sure file I/O is defined
// USBFILE is a Macro defined to preface a file name with the directory path for a mounted USB
drive
#ifndef USBFILE
   // _STRINGIFY_ is an auxiliary macro which converts the argument into a string (by
surrounding it with double quote marks)
   #define _STRINGIFY_(x) #x
   // the USBFILE macro appends x to the path for the USB drive, then uses _STRINGIFY_ to make
the text a string
   #define USBFILE(x) _STRINGIFY_(/kovan/media/sda1/x)
#endif
int main()
{
   FILE *f; // file pointer f (the macro defining the FILE data type is in <stdio.h>)
   // A file for the USB drive named "myfile" is set up using macro USBFILE
   char s[81], chkf[] = USBFILE(myfile); // set up the file path for myfile in string variable
chkf
   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);
   }
   // (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 Robot Controller. If not already present, there will now be a file named `myfile` on the USB drive, which can be read using a text processor to confirm the write operation was successful.

The KIPR Robot Controller Library File

The KIPR Robot Controller Library provides pre-compiled functions for employing the features of the KIPR Robot Controller or the KIPR Robot Controller simulator. The prototype declarations for these functions are automatically included by the C preprocessor if the KISS IDE target selection for a program is the KIPR Robot Controller or the simulator.

Refresher for Commonly Used KIPR Robot Controller Library Functions

The most commonly used KIPR Robot Controller Library functions for robot control are ones for accessing sensor ports, operating DC motors, and suspending a program while a motor action is in progress. A quick refresher for the ones typically used follows:

`display_printf(`*<col>*`, `*<row>*`, `*<printf-arguments>*`);`

> The purpose of this function is to display formatted output at a specific (column, row) location on the KIPR Robot Controller program console screen using standard `printf` formatting. The column range for the display is 0 to 41. For the 3 button (A,B,C) console display, the row range is 0 to 9, and for the 6 button (X,Y,Z,A,B,C) console display, the row range is 0 to 7. Excess data printed to a row is truncated (i.e. text does not wrap). Printing "`\n`" using `display_printf` will cause undesired display results.

`digital(`*<port#>*`)`

> For a switch plugged into the port, returns 0 if the switch is open and returns 1 if the switch is closed. Digital ports are numbered 8-15. Typically used with button (switch) sensors employed in bumpers or to detect limit of travel. Will also work with sensors having an analog response by interpreting the voltage reading as 0 or 1.

`analog(`*<port#>*`);`

> For an analog sensor plugged into the port, returns a value scaled to the range 0-1023 representing the analog voltage variation produced by the sensor. Analog ports are numbered 0-7. Light sensors and range finders are examples of sensors used in analog ports. For some sensors, in particular, the "ET" distance sensor used for robotics, the port's pull up resistor (enabled by default) has to be disabled for the analog values to be meaningful.

`analog_et(`*<port#>*`);`

> For an analog sensor plugged into the port, disables the port's pullup resistor and returns a value scaled to the range 0-1023 representing the analog voltage variation produced by the sensor. Analog ports are numbered 0-7. This is the function normally used with the "ET" distance sensor used for robotics, which has a pullup sensor already integrated into its circuitry.

`msleep(`*<int_msecs>*`);`

> Causes the function to pause for the specified number of milliseconds (motor action continues).

`mav(`*<motor_#>*`, `*<vel>*`);`

> Turns on the specified motor port, maintaining the specified motor velocity using PID (proportional, integral, derivative) motor control. The motor number is an integer in the range 0 to 3. Motor

velocity is an integer between -1000 and 1000 where 0 means the motor is off and negative numbers direct the motor to run in the reverse direction for how it is plugged into the motor port.

`motor(<motor_#>, <motor_power>);`

Turns on the specified motor port at the PWM power level specified, where a positive power value operates the motor in the direction as `fd` and a negative value operates it in the reverse direction. The motor number is an integer in the range 0 to 3. The power setting is -100 to 100 representing percentage of full power. Motor response with respect to power is non-linear (i.e., doubling power does not double motor velocity). Moreover, motor velocity for a power setting varies according to load. In contrast, PID motor control varies the PWM power applied to try to establish and maintain a specified motor velocity.

`fd(<motor_#>);`

Turns of the specified motor port at maximum PWM (pulse width modulation) power for the motor's forward direction for how the motor is plugged in. The motor number is an integer between 0 and 3

`bk(<motor_#>);`

Turns of the specified motor port at maximum PWM (pulse width modulation) power for the motor's reverse direction for how the motor is plugged in. The motor number is an integer between 0 and 3

`off(<motor_#>);`

Turns off the specified motor port. Once a motor port is turned on, a motor plugged into the port will continue operating until the motor port is turned off or the program terminates (which turns off all motor ports).

`ao();`

Turns all motor ports off.

KIPR Robot Controller Library Functions

The following is a comprehensive list of the pre-compiled functions provided by the KIPR Robot Controller Library. The library functions for using a USB camera, a graphics window, or a USB depth sensor, and those for controlling a Create module, are presented separately.

`a_button` [Category: Sensors]

Format: `int a_button();`
Returns the state of the A button on the KIPR Robot Controller program console (1 if pressed, 0 otherwise).

`a_button_clicked` [Category: Sensors]

Format: `int a_button_clicked();`
Returns the state of the A button on the KIPR Robot Controller program console (1 of pressed, 0 otherwise). Continued function execution is blocked while the button is 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 Robot Controller (left - right).

`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 Robot Controller (forward - back).

`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 Robot Controller (up - down). When the KIPR Robot Controller is horizontal it is calibrated to have a value corresponding to the gravitational constant G (your acceleration towards the center of the Earth that keeps you from flying off of the planet).

`alloff` [Category: Motors]

Format: `void alloff();`
Turns off all motor ports. `ao` is a short form for `alloff`.

`analog` [Category: Sensors]

Format: `int analog(int p);`
Returns the value of the sensor plugged into analog port `p` scaled as an integer between 0 and 1023. Analog ports on the KIPR Robot Controller are numbered 0 through 7. Each analog port on the KIPR Robot Controller has a selectable pullup resistor, enabled by default since many analog sensors do not have this component built in, although some do (such as the "ET" range finder). For a sensor such as the ET sensor, the values returned by `analog` will be unsatisfactory unless the pullup resistor for its port is disabled by using `analog_et`.

`analog_et` [Category: Sensors]

Format: `int analog_et(int p);`
For an analog sensor plugged into analog port `p`, disables the port's pullup resistor and returns a value scaled to the range 0-1023 representing the analog voltage variation produced by the sensor. Analog ports are numbered 0-7. This is the function normally used with the "ET" distance sensor used for robotics, which has a pullup sensor already integrated into its circuitry.

`analog8` [Category: Sensors]

Format: `int analo8(int p);`
8-bit version of analog. Returns the value of the sensor plugged into analog port `p` scaled as an integer between 0 and 255. Analog ports on the KIPR Robot Controller are numbered 0 through 7.

`analog10` [Category: Sensors]

Format: `int analog10(int p);`
Returns the value of the sensor plugged into analog port `p` scaled as an integer between 0 and 1023. Analog ports on the KIPR Robot Controller are numbered 0 through 7. Each analog port on the KIPR Robot Controller has a selectable pullup resistor, enabled by default since many analog sensors do not have this component built in, although some do (such as the "ET" range finder). For a sensor such as the ET sensor, the values returned by `analog10` will be unsatisfactory unless the pullup resistor for its port is disabled. Same as `analog`

`any_button` [Category: Sensors]

Format: `int any_button()();`
Returns 1 if any button (A,B,C,X,Y,Z,Side) is pressed.

`ao` [Category: Motors]

Format: `void ao();`
Turns off all motor ports (same as `alloff`).

`b_button` [Category: Sensors]

Format: `int b_button();`
Returns the state of the B button on the KIPR Robot Controller program console (1 if pressed, 0 otherwise).

`b_button_clicked` [Category: Sensors]

Format: `int b_button_clicked();`
Returns the state of the B button on the KIPR Robot Controller program console (1 of pressed, 0 otherwise). Continued function execution is blocked while the button is 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 from the KIPR Robot Controller loud speaker. Returns when the tone is finished.

`bk` [Category: Motors]

Format: `void bk(int m);`
Turns on the motor plugged into motor port `m` at full PWM power in the reverse direction (red light comes on for the KIPR Robot Controller motor port). Motor ports are numbered from 0 to 3.
Example:

```
bk(3); // full power reverse for motor 3
```

`block_motor_done` [Category: Motors]

Format: `void block_motor_done(int m);`
If a motor positioning function is in progress for the motor plugged into motor port `m`, continued function execution is blocked until the motor has reached its goal position. Motor positioning functions are `move_to_position` (or `mtp`) and `move_relative_position` (or `mrp`).
Example:

```
mtp(0,500,20000); // turn on motor 0 at 500 ticks/sec and stop when position
20000 is reached
block_motor_done(1); // pause until mtp has reached its goal position of 20000 for
motor 1
```

This function must be used with some care, since if something prevents the motor from reaching its goal position, the program will hang until halted by external means.

`bmd` [Category: Motors]

Format: `void bmd(int m);`
This function is the same as `block_motor_done`, just under a (much) shorter name.

`c_button` [Category: Sensors]

Format: `int c_button();`
Returns the state of the C button on the KIPR Robot Controller program console (1 if pressed, 0 otherwise).

`c_button_clicked` [Category: Sensors]

Format: `int c_button_clicked();`
Returns the state of the C button on the KIPR Robot Controller program console (1 of pressed, 0 otherwise). Continued function execution is blocked while the button is pressed. The construction

```
while (c_button()==0) {
while (c_button()==1); ...} //debounce C button
```

is equivalent to

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

`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.

`console_clear` [Category: Output]

Format: `void console_clear();`
Clear the KIPR Robot Controller program console print buffer. See also `display_clear`.

`digital` [Category: Sensors]

Format: `int digital(int p);`
Returns the value of the sensor in digital 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. Digital ports on the KIPR Robot Controller are numbered 8 through 15.

`disable_servo` [Category: Servos]

Format: `void disable_servo(int p);`
Disables the specified servo port. Servo ports are disabled by default, and if enabled, consume power whenever a motor is plugged into them. Servo ports on the KIPR Robot Controller are numbered 0 to 3.

`disable_servos` [Category: Servos]

Format: `void disable_servos();`
Disables all servo motor ports (powers down all servo motors). Servo ports are disabled by default, and if enabled, consume power whenever a motor is plugged into them.

`display_clear` [Category: Output]

Format: `void display_clear();`
Clear the KIPR Robot Controller program console screen for using `display_printf`. See also `console_clear`.

`display_printf` [Category: Output]

Format: `void display_printf(int col, int row, char s[], ... );`
The purpose of this function is to display formatted output at a specific (column, row) location on the KIPR Robot Controller program console screen using standard `printf` formatting. The column range for the display is 0 to 41. For the 3 button (A,B,C) console display, the row range is 0 to 9, and for the 6 button (X,Y,Z,A,B,C) console display, the row range is 0 to 7. Excess data printed to a row is truncated (i.e. text does not wrap). Printing `"\n"` using `display_printf` will cause undesired display results.

`enable_servo` [Category: Servos]

Format: `void enable_servo(int p);`
Enables specified servo port. Servo ports are disabled by default, and if enabled, consume power whenever a motor is plugged into them. If a servo position for a port hasn't been set when it is enabled, the default position is 1023. Servo ports on the KIPR Robot Controller are numbered 0 to 3.

`enable_servos` [Category: Servos]

Format: `void enable_servos();`
Enables all servo motor ports. Servo ports are disabled by default, and if enabled, consume power whenever a motor is plugged into them.

`extra_buttons_show` [Category: Output]

Format: `void extra_buttons_show();`
Shows the X, Y, and Z buttons on the KIPR Robot Controller program console screen above the A, B, and C buttons. 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 KIPR Robot Controller program console screen. 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 on the motor plugged into motor `m` at full PWM power in the forward direction (green light comes on for the KIPR Robot Controller motor port). Motors are numbered 0 to 3.
Example:

```
fd(3); // full power forward for motor 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_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);`
For a motor moving to a goal position set by a motor positioning command, returns 1 if the motor has reached the goal position and 0 otherwise. Motor positioning functions are `move_to_position` (or `mtp`) and `move_relative_position` (or `mrp`). See also `block_motor_done`.

`get_motor_position_counter` [Category: Motors]

Format: `int get_motor_position_counter(int m);`
Returns the current motor position for motor `m` in "ticks" (a value which is continually being updated for each motor using PID motor control based on 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 PID control values currently set for the KIPR Robot Controller motor ports. The `p`, `i` and `d` arguments are numerators, and the `pd`, `id` and `dd/` arguments are their respective denominators for coefficients used by the PID control algorithm. While all of the parameters are integers, the coefficients used within the PID control algorithm may be integers or floating point. PID stands for proportional, integral, derivative and uses BEMF (back electromotive force) feedback from a motor to determine how much PWM power to apply based on predictors governed by the PID coefficients. If a motor is jerky, the p and d terms should be reduced in size. If a motor lags far behind a specified velocity, they should be increased. The default values are set at firmware install as effective values for motors sold by KIPR. 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. Servo ports on the KIPR Robot Controller are numbered 0 to 3.See also `enable_servo, disable_servo`.

`get_servo_position` [Category: Servos]

Format: `int get_servo_position(int srv);`
Returns the last position value set for the servo in port `srv`. The value will be in the range 0 to 2047. Servo ports on the KIPR Robot Controller are numbered 0 to 3. See also `set_servo_position`.

`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 port `m` at `p`% of full PWM motor power. The range for `p` is -100 to 100, where a negative value runs the motor in its reverse direction. On the KIPR Robot Controller, a motor port supplying forward power lights green, and for reverse lights red, although actual motor direction depends on how it is plugged in.

`move_at_velocity` [Category: Motors]

Format: `void move_at_velocity(int m, int vel);`
Turns on motor port `m`, varying applied PWM power using PID motor control to maintain motor velocity `vel` indefinitely. The velocity range is -1000 to 1000 ticks per second. On the KIPR Robot Controller, a motor port supplying forward power lights green, and for reverse lights red, although actual motor direction depends on how it is plugged in.

`move_relative_position` [Category: Motors]

Format: `void move_relative_position(int m, int absvel, int pos);`
Turns on motor port `m`, varying applied PWM power using PID motor control to maintain motor velocity `absvel` (±) until the motor has moved from its current position cp to the goal position cp + `pos`. The range for the `absvel` argument is 0 to 1000 ticks per second. On the KIPR Robot Controller, a motor port supplying forward power lights green, and for reverse lights red, although actual motor direction depends on how it is plugged in.
Example:

```
move_relative_position(1,275,-1100);
```

`move_to_position` [Category: Motors]

Format: `void move_to_position(int m, int absvel, int pos);`
Turns on motor port `m`, varying applied PWM power using PID motor control to maintain motor velocity `absvel` (±) until the motor has moved from its current position to goal position `pos`. The range for the `absvel` argument is 0 to 1000 ticks per second. On the KIPR Robot Controller, a motor port supplying forward power lights green, and for reverse lights red, although actual motor direction depends on how it is plugged in. If the motor is already at the goal position `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);`
Suspends function execution for the amount of time specified in milliseconds.
Example:

```
msleep(1500); //wait for 1.5 seconds
```

`off` [Category: Motors]

Format: `void off(int m);`
Turns off motor port `m`.
Example:

```
off(1); // turn off motor 1
```

`power_level` [Category: Sensor]

Format: `double power_level();`
Returns the current power level for the KIPR Robot Controller battery.

`run_for` [Category: Threads]

Format: `void run_for(double sec, <function_name>);`
Runs the specified function and tracks its execution until either the function finishes or the specified number of seconds has elapsed. If the specified number of seconds has elapsed, `runfor` terminates execution for the function.

`seconds` [Category: Time]

Format: `double seconds();`
Returns the number of seconds that have elapsed since system start up according to the system's internal clock. Resolution is 0.001 seconds (one millisecond).

`set_a_button_text` [Category: Sensors]

Format: `void set_a_button_text char txt[]);`
This function resets the text displayed on the A button to be the text string specified rather than the default string "A".

`set_b_button_text` [Category: Sensors]

Format: `void set_b_button_textchar txt[]);`
This function resets the text displayed on the B button to be the text string specified rather than the default string "B".

`set_c_button_text` [Category: Sensors]

Format: `void set_c_button_textchar txt[]);`
This function resets the text displayed on the C button to be the text string specified rather than the default string "C".

`set_digital_output` [Category: Output]

Format: `void set_digital_output(int port, int inout);`
Digital ports on the KIPR Robot Controller can be configured for either input or output. By default digital ports are set for input. If the value of `inout` is 1, the port is configured for output, for example, the statement

```
set_digital_output(9,1);
```

when executed configures digital port 9 for output. The port number specified must be in the range from 8 to 15. See also `set_digital_value`. See below for an example program using digital output.

`set_digital_value` [Category: Output]

Format: `void set_digital_value(int port, int value);`
Sets the (SEN) value for the specified port on the KIPR Robot Controller to either 0 (low) or 1 (high). For the sensor ports on the KIPR Robot Controller, the SEN rail is the inner rail having a single row of

sockets. The specified port must be in the range 8 to 15. The library function `set_digital_ouput` is used to configure a digital port on the KIPR Robot Controller for output (or to re-configure it for input).

As an example, if the anode lead for an LED (the longer of its two leads) is plugged into the SEN socket for digital port 9 and its cathode lead into the GND rail (the outer rail for the PWR/GND rails), then if digital port 9 has been set for output

```
set_digital_value(9,1);
```

will turn on the LED when executed and

```
set_digital_value(9,0);
```

will turn it off. See below for an example program using digital output.

`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 change the PID control values currently set for the KIPR Robot Controller motor ports. The `p`, `i` and `d` arguments are numerators, and the `pd`, `id` and `dd` arguments are their respective denominators for coefficients used by the PID control algorithm. While all of the parameters are integers, the coefficients used within the PID control algorithm may be integers or floating point. PID stands for proportional, integral, derivative and uses BEMF (back electromotive force) feedback from a motor to determine how much PWM power to apply based on predictors governed by the PID coefficients. If a motor is jerky, the p and d terms should be reduced in size. If a motor lags far behind a specified velocity, they should be increased. The default values are set at firmware install as effective values for motors sold by KIPR. See also `get_pid_gains`

`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. The KIPR Robot Controller has 4 servo ports numbered 0 to 3. A servo motor consumes maximum power when trying to reach a position, so if the servo cannot reach the position specified, its power consumption will rapidly pull the KIPR Robot Controller battery down. Servo motors tend to reach their limit of travel around 200 short of either end of the 0 to 2047 range, so values close to the limit of travel should be avoided. Additionally, servo travel is not smooth, and positions tend to be about 5 units apart on the 0 to 2047 scale; i.e., position increments should be at least 5. see also `get_servo_position`.

`set_x_button_text` [Category: Sensors]

Format: `void set_x_button_text(char txt[]);`
This function resets the text displayed on the X button to be the text string specified rather than the default string "X". See also `extra_buttons_show, extra_buttons_hide, get_extra_buttons_visible`.

`set_y_button_text` [Category: Sensors]

Format: `void set_y_button_text(char txt[]);`
This function resets the text displayed on the Y button to be the text string specified rather than the default string "Y". See also `extra_buttons_show, extra_buttons_hide, get_extra_buttons_visible`.

`set_z_button_text` [Category: Sensors]

Format: `void set_z_button_text(char txt[]);`
This function resets the text displayed on the Z button to be the text string specified rather than the default string "Z". See also `extra_buttons_show, extra_buttons_hide, get_extra_buttons_visible`.

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

Format: `int side_button();`
Returns the state of the (physical) side button on the KIPR Robot Controller (1 if pressed, 0 otherwise).

`side_button_clicked` [Category: Sensors]

Format: `int side_button_clicked();`
Returns the state of the (physical) side button on the KIPR Robot Controller (1 if pressed, 0 otherwise). Continued function execution is blocked while the button is pressed. The construction returns 1 for pressed, 0 for not pressed. The construction

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

is equivalent to

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

`thread_create` [Category: Threads]

Format: `thread thread_create(`<*function name*>`);`
The `thread_create` function 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 ID's created 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 to the program. 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 to be run in a thread again. Since the system limits how many threads can be created, thread management is important; i.e., threads should be destroyed once they have finished (if threads seem to stop working it is usually because too many have been created). See the section on threads below for additional information and an example.

`thread_destroy` [Category: Threads]

Format: `void thread_destroy(thread tid);`
The `thread_destroy` function is used to destroy a thread created for running a function in parallel to main. The thread is destroyed by passing its thread ID to `thread_destroy`. If the function for the thread is running in the thread when the thread is destroyed, its execution is terminated. See the section on threads below for additional information and an example.

`thread_start` [Category: Threads]

Format: `void thread_start(thread tid);`
The `thread_start` function is used to activate the thread given by `tid`, running its associated function in the thread in parallel with `main` and any other active threads. The value of argument `tid` must have a thread ID value as returned by `thread_create`. Keep in mind that thread IDs generated by `thread_create` must be retained in variables of type `thread` to remain available for later use in a program. The thread is active until its function finishes or until it is terminated by `thread_destroy`. A global variable can be used as a flag for a function running in a thread to signal when it is done.

The following example shows the `main` process creating a thread for the function `check_sensor`, running the function in the thread, and then destroying the thread one second later (whether or not the thread is still active):

```
int main()
```

```
{
    thread tid;  // variable for holding a thread's ID
    tid = thread_create(check_sensor);  // capture thread's ID in tid
    thread_start(tid);  // run check_sensor in the thread
    msleep(1000);  // let it keep running for a second
    thread_destroy(tid);  // stop check_sensor if still running
}
```

See the section on threads below for a complete example using threads.

`thread_wait` [Category: Threads]

Format: `void thread_wait(thread tid);`
If `thread_wait` is executed in a function, it will block continued execution of the function until the function running in the thread given by `tid` has finished (or until the thread given by `tid` is destroyed by some other process). If the thread given by `tid` is inactive, the function does nothing. `thread_wait` is used for synchronization of multiple threads (e.g., thread 1 doesn't continue until thread 2 is done). See the section on threads below for additional information and an example.

`x_button` [Category: Sensors]

Format: `int x_button();`
Returns the state of the X button when it is visible on the KIPR Robot Controller program console (1 if pressed, 0 otherwise). 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();`
Returns the state of the X button when it is visible on the KIPR Robot Controller program console (1 if pressed, 0 otherwise). Continued function execution is blocked while the button is pressed. The construction

```
while (x_button()==0) {
while(x_button()==1); ...} //debounce X button
```

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();`
Returns the state of the Y button when visible on the KIPR Robot Controller program console (1 if pressed, 0 otherwise). 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();`
Returns the state of the Y button when it is visible on the KIPR Robot Controller program console (1 if pressed, 0 otherwise). Continued function execution is blocked while the button is pressed. The

construction

```
while (y_button()==0) {
while(y_button()==1); ...} //debounce Y button
```

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();`
Returns the state of the Z button when visible on the KIPR Robot Controller program console (1 if pressed, 0 otherwise). 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();`
Returns the state of the Z button when it is visible on the KIPR Robot Controller program console (1 if pressed, 0 otherwise). Continued function execution is blocked while the button is pressed. The construction

```
while (z_button()==0) {
while(z_button()==1); ...} //debounce Z button
```

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`.

Example program for lighting an LED plugged into a digital port

By default the digital ports for the KIPR Robot Controller are configured for input. The KIPR Robot Controller 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 Robot Controller Library function `set_digital_value` is used to set the port's output value to either 0 (low) or 1 (high); for example,

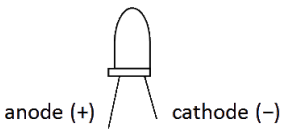
```
set_digital_value(9, 1);
```

sets the output value for port 9 high.

If you have a typical 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 Robot Controller, 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.




```c
/*  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); // set digital port direction to digital output
   while (side_button() == 0) {
      set_digital_value(9, 1); // set digital output to 1 (high)
      msleep(500);
      set_digital_value(9, 0); // set digital output to 0 (low)
      msleep(500);
   }
   set_digital_output(9, 0); // set digital port direction back to digital input
   printf("\ndone\n");
}
```

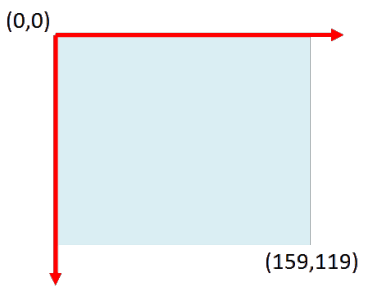
KIPR Robot Controller Vision Library Functions

The KIPR Robot Controller Vision System incorporates color vision tracking and QR code identification. A USB web camera is used to provide images to the KIPR Robot Controller at a rate dependent on lighting conditions but exceeding 6 frames per second. Using the KIPR Robot Controller 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 Robot Controller 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. See the KIPR Robot Controller Manual available from KIPR for information on setting up camera configurations on a KIPR Robot Controller controller.

There are three camera image resolutions supported on the KIPR Robot Controller, 160×120 (`LOW-RES`), 320×240 (`MED-RES`), and 640×480 (`HIGH-RES`). `LOW-RES` is sufficient for most purposes and requires far less processing overhead. For `LOW-RES` the upper left corner for the image has coordinates (0,0) and the lower right has coordinates (159,119). The camera image displayed on the KIPR Robot Controller is slightly smaller than the actual image size.



KIPR Robot Controller 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 Robot Controller Vision Library functions are provided for decoding any QR code in the image. See the QR code vision functions `get_object_data, get_object_data_length` below.

`camera_close` [Category: Vision]

Format: `void camera_close();`
Cleanup the current camera instance. See also `camera_open, camera_open_at_res, camera_open_device.`

`camera_load_config` [Category: Vision]

Format: `int camera_load_config(char name[]);`
Loads a config file on the KIPR Robot Controller other than the default config file. You **must** append `.config` to the name for this function to locate it. Returns 1 on success, 0 on failure. See also `camera_open, camera_open_at_res, camera_open_device.`

`camera_open` [Category: Vision]

Format: `int camera_open();`
Opens the KIPR Robot Controller's default channel configuration. The default configuration is selected from among the channel configurations defined on the KIPR Robot Controller using its *settings..channels* menu. A resolution of `LOW_RES` is used. Returns 1 on success, 0 on failure. See also `camera_open_at_res, camera_open_device, camera_close.`

`camera_open_at_res` [Category: Vision]

Format: `int camera_open_at_res(int res_numb);`
Opens the KIPR Robot Controller's default channel configuration. The default configuration is selected from among the channel configurations defined on the KIPR Robot Controller 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, 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_open_at_res, camera_close.`

`camera_update` [Category: Vision]

Format: `int camera_update();`
Retrieves the current 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_camera_frame` [Category: Vision]

Format: `const unsigned char* get_camera_frame();`
Returns a pointer to the camera frame. The pointer is valid until `camera_update()` is called again. Frame data is in BGR 888 pixel format, 3 bytes per pixel. It is stored as a 1-dimensional character array where each byte is the 8-bit (unsigned) integer value for each BGR color (blue-green-red). For the camera default resolution `LOW_RES` (160×120), the camera frame has length 3×160×120 = 57,600 and has to be processed byte by byte. The following code illustrates processing the camera frame using the graphics function `graphics_pixel` to transfer it to a graphics window for display (see the Graphics Library):

```
int r,c,i;
const unsigned char *img;
camera_update();
img=get_camera_frame();
for(r=0;r<120;r++) {
   for(c=0;c<160;c++) {
      i=3*(160*r + c); // index of pixel to paint into row r, column c
      graphics_pixel(c,r,img[i+2],img[i+1],img[i]); // RGB order by reversing
GBR
   }
}
graphics_update();
```

`get_object_area` [Category: Vision]

Format: `int get_object_area(int channel, int);`
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 mcenter;
mcenter = get_object_center(0,2);
printf("x center coord %d y center coord %d\n", mcenter.x, mcenter.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 mcenter;
mcentroid = get_object_centroid(0,2);
printf("x centroid coord %d y centroid coord %d\n", mcentroid.x, mcentroid.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_code_num` [Category: Vision]

Format: `int get_code_num(int channel, int object);`
Returns the QR data associated with an object for a QR channel as an integer. If the given channel or object doesn't exist, -1 is returned. See also `get_object_data`.

`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 density of color pixels for the 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" for the specified channel. Objects are sorted by bounding box 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, 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, 0 is returned. If the channel or object is invalid, 0 is returned. See also `get_object_data`.

Example program for using a camera to track an object

Camera functions will not provide meaningful data until the camera has been activated using `camera_open` or `camera_open_at_res`. For most purposes, the `LOW_RES` setting used by `camera_open` is sufficient and it serves to reduce frame processing overhead. Once the camera has been activated, the camera function `camera_update` is used to obtain the current frame in the camera's field of view. Note that `camera_update` is called every time through a camera processing loop to provide a current camera frame for processing.

```
/* This program points a servo (that is plugged into port 0 and centered on the camera's
field of vision) towards an object that fits into the color model defined for channel 0 */
int main()
{
   int offset, x, y;
   enable_servo(0);      // enable servo
   camera_open();        // 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");
}
```

Example program to translate a QR code while showing the camera image in a graphics window

This program displays the camera image in a 160×120 graphics window centered on the 320×240 KIPR Robot Controller display, decoding any QR code found and displaying the result in the space above the graphics window.

```
// Assume channel 1 is configured for identifying QR codes
// If a QR code is found, it is translated
int main()
{
    int r, c, ix, i, lngth;
    const unsigned char *img; // variable to hold camera image
    camera_open(); graphics_open(160,120); // activate camera and open a graphics window
    camera_update(); // get most recent camera image and process it
    while(side_button()==0) {
        img=get_camera_frame(); // get a camera frame and display it in graphics window
        for(r=0; r<120; r++) {
             for(c=0; c<160; c++) {
                 ix=3*(160*r + c); // index of pixel to paint into row r, column c
                 graphics_pixel(c,r,img[ix+2],img[ix+1],img[ix]); // RGB order by reversing
```

```
GBR
                }
            }
            graphics_update(); // show the frame
            if (get_object_count(1) > 0) { // there is a QR code in view
                display_printf(0,0,"QR code:                                ");
                lngth = get_object_data_length(1,0); // decode and display above graphics window
                for(i=0; i < lngth; i++) { // print QR code letter by letter until end of data
                    display_printf(9+i,0,"%c", get_object_data(1,0)[i]);
                }
            }
            else {
                display_printf(0,0,"No QR code detected                     ");
            }
            camera_update(); // get new image data before repeating
        }
        camera_close(); graphics_close(); // clean up
}
```

KIPR Robot Controller Graphics Functions

The KIPR Robot Controller (and Simulator) support basic graphical draw operations, which allow a user to create their own user interface if they wish. Functions are provided to draw and color pixels, lines, circles, triangles, and rectangles (non-rotated). The screen and two dimensional objects can also be filled with a single color. Both mouse clicks and cursor location can be detected (on the KIPR Robot Controller, a screen tap corresponds to a left mouse click).

Note that any virtual features of the KIPR Robot Controller user interface (such as the A,B,C buttons) will be unavailable if obscured by the graphics window (it is advisable to use the side button when doing graphical applications for this reason).

`graphics_open` [Category: Graphics]

Format: `int graphics_open(int` width, `int` height);
Opens and centers a graphics window on the display of the specified width and height. The maximum width for the KIPR Robot Controller display is 320, and the maximum height is 240. See also `graphics_close`.

`graphics_close` [Category: Graphics]

Format: `void graphics_close();`
Closes the graphics window on the display, restoring access to any virtual buttons underneath it. See also `graphics_open`.

`graphics_update` [Category: Graphics]

Format: `void graphics_update();`
Repaints the pixels in the graphics window to show any changes that have been made.

`graphics_clear` [Category: Graphics]

Format: `void graphics_clear();`

Erases the graphics window (not shown until `graphics_update`).

`graphics_fill` [Category: Graphics]

Format: `void graphics_fill(int r, int g, int b);`
Colors the pixels in the window using the `r`,`g`,`b` color encoding.

`graphics_pixel` [Category: Graphics]

Format: `void graphics_pixel(int x, int y, int r, int g, int b);`
Colors the specified pixel in the window using the `r`,`g`,`b` color encoding, where columns `x` and rows `y` are indexed starting from the upper left corner of the graphics window.

`graphics_line` [Category: Graphics]

Format: `void graphics_line(int x1, int y1, int x2, int y2, int r, int g, int b);`
Draws a line in the window from the specified (`x1`,`y1`) pixel to the (`x2`,`y2`) pixel using the `r`,`g`,`b` color encoding (where columns x and rows y are indexed starting from the upper left corner of the graphics window).

`graphics_circle` [Category: Graphics]

Format: `void graphics_circle(int cx, int cy, int radius, int r, int g, int b);`
Draws a circle in the window of the specified radius centered at (`cx`,`cy`) using the `r`,`g`,`b` color encoding (where columns x and rows y are indexed starting from the upper left corner of the graphics window).

`graphics_circle_fill` [Category: Graphics]

Format: `void graphics_circle_fill(int cx, int cy, int radius, int r, int g, int b);`
Draws a circle in the window of the specified radius centered at (`cx`,`cy`) and fills it using the `r`,`g`,`b` color encoding (where columns x and rows y are indexed starting from the upper left corner of the graphics window).

`graphics_rectangle` [Category: Graphics]

Format: `void graphics_rectangle(int x1, int y1, int x2, int y2, int r, int g, int b);`
Draws a rectangle in the window with upper left corner (`x1`,`y1`) and lower right corner (`x2`,`y2`) using the `r`,`g`,`b` color encoding (where columns x and rows y are indexed starting from the upper left corner of the graphics window).

`graphics_rectangle_fill` [Category: Graphics]

Format: `void graphics_rectangle_fill(int x1, int y1, int x2, int y2, int r, int g, int b);`
Draws a rectangle in the window with upper left corner (`x1`,`y1`) and lower right corner (`x2`,`y2`) and fills it using the `r`,`g`,`b` color encoding (where columns x and rows y are indexed starting from the upper left corner of the graphics window).

`graphics_triangle` [Category: Graphics]

Format: `void graphics_triangle(int x1, int y1, int x2, int y2, int x3, int y3, int r, int g, int b);`
Draws a triangle in the window with corners (`x1`,`y1`), (`x2`,`y2`), (`x3`,`y3`) using the `r`,`g`,`b` color encoding (where columns x and rows y are indexed starting from the upper left corner of the graphics window).

`graphics_triangle_fill` [Category: Graphics]

Format: `void graphics_triangle_fill(int x1, int y1, int x2, int y2, int x3, int y3, int r, int g, int b);`
Draws a triangle in the window with corners (`x1`,`y1`), (`x2`,`y2`), (`x3`,`y3`) and fills it using the `r`,`g`,`b` color encoding (where columns x and rows y are indexed starting from the upper left corner of the graphics window).

`get_mouse_position` [Category: Graphics]

Format: `void get_mouse_position(int *x, int *y);`
Assigns the column,row position of the cursor in the window to the two specified address parameters. Note that the typical call for this function will look like

```
get_mouse_position(&col, &row);
```

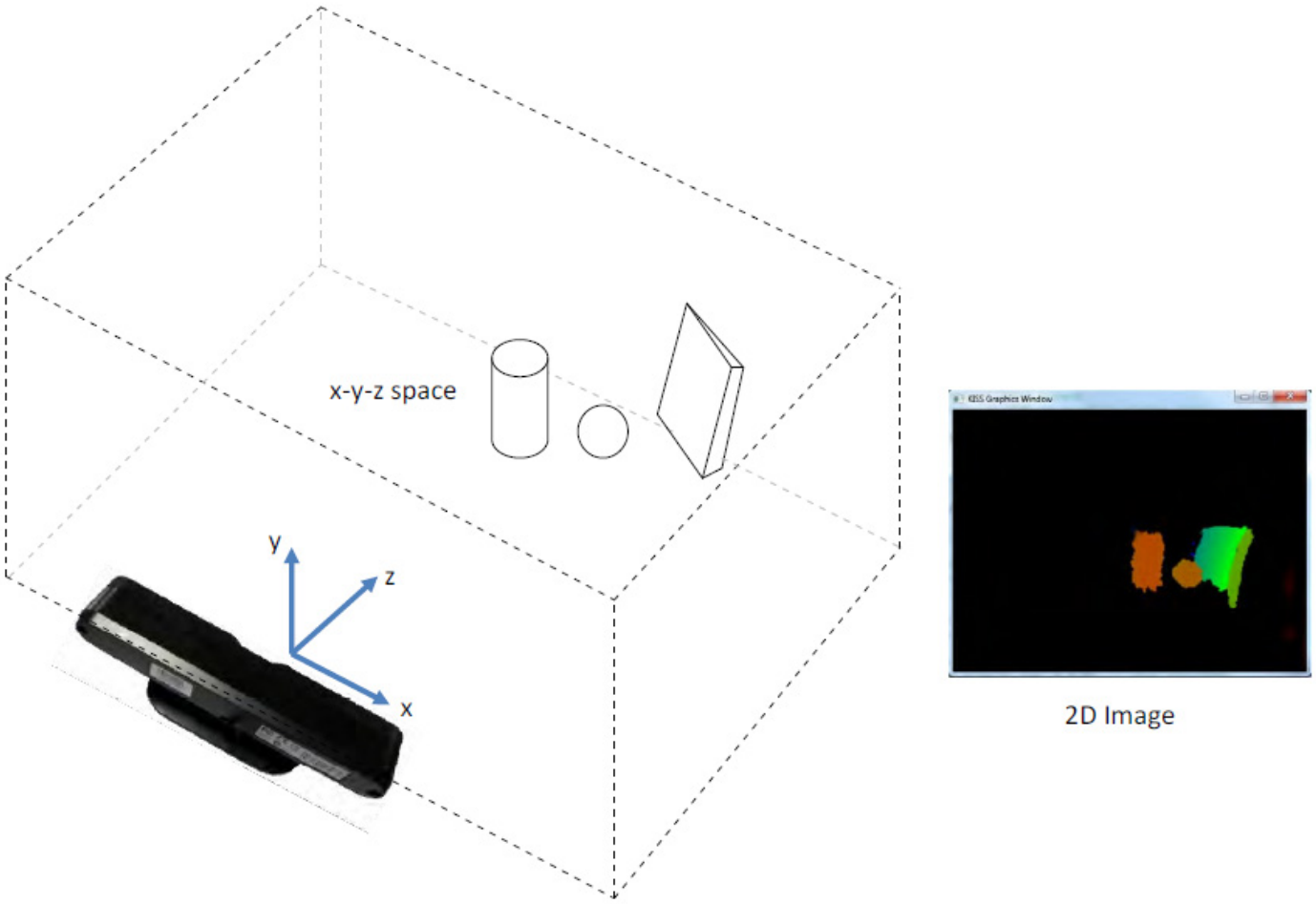
`get_mouse_left_button` [Category: Graphics]

Format: `int get_mouse_left_button();`
Returns 1 if the left mouse button is clicked or if the KIPR Robot Controller display is tapped. The two additional functions `get_mouse_middle_button` and `get_mouse_right_button` are also available but have no meaning on the KIPR Robot Controller (unless a mouse is attached)

KIPR Robot Controller Depth Functions

The KIPR Robot Controller has functions supporting the use of an ASUS Xtion as a USB depth sensor.

The USB depth sensor provides the KIPR Robot Controller with a 240×320 row,column (2-dimensional) image of the space in front of it (rows measured down from the upper left corner of the image and columns to the right). The depth functions provide information about the three dimensional space in front of the Xtion, where the space is organized in (x,y,z) coordinates with the x-axis directed right, the y-axis is directed up, and the z-axis directed straight out from the Xtion. Coordinate (0,0,0) is the center of the Xtion face. Coordinate values are given in millimeters (mm). The depth sensor's effective range is from about 1/2 meter to 5 meters.



`depth_open` [Category: Depth]

Format: `int depth_open();`
Turns on the depth sensor. Returns 1 on success, 0 on failure. See also `depth_close`.

`depth_close` [Category: Depth]

Format: `void depth_close();`
Clean up the depth instance and power down the sensor. See also `depth_open`.

`depth_update` [Category: Depth]

Format: `int depth_update();`
Generates a new image from the depth sensor for use by the depth functions. Returns 1 on success, 0 on failure.

`depth_scanline_update` [Category: Depth]

Format: `int depth_scanline_update(int row);`
Replaces the current scanline information with the scanline information for the specified row of the image. Returns 1 on success, 0 on failure.

`get_depth_value` [Category: Depth]

Format: `int get_depth_value(int row, int col);`
Returns the value of the depth coordinate z in mm for row and column positions of the image. See also `get_depth_world_point`.

`get_depth_world_point` [Category: Depth]

Format: `point3 get_depth_world_point(int row, int col);`
Returns an object of type `point3` giving the (x,y,z) coordinates for the object at position (row,col) in the current image. For

```
point3 p3;
p3 = get_depth_world_point(4,7);
```

the (x,y,z) coordinates for row 4, column 7 in the image are given by `p3.x`, `p3.y`, `p3.z`. The value returned by `get_depth_value` is the same as `p3.z`.

`get_depth_world_point_x` [Category: Depth]

Format: `int get_depth_world_point_x(int row, int col);`
Returns the x coordinate for the object at position (row,col) in the current image. See also `get_depth_world_point`.

`get_depth_world_point_y` [Category: Depth]

Format: `int get_depth_world_point_y(int row, int col);`
Returns the y coordinate for the object at position (row,col) in the current image. See also `get_depth_world_point`.

`get_depth_world_point_z` [Category: Depth]

Format: `int get_depth_world__zpoint(int row, int col);`
Returns the z coordinate for the object at position (row,col) in the current image. The return value is the same as the value returned by `get_depth_value`. See also `get_depth_world_point`.

`get_depth_scanline_object_count` [Category: Depth]

Format: `int get_depth_scanline_object_count();`
Returns the number of objects detected on the scanline, where an object is detected by not having a depth break in scanning across it. This means that if the scanline crosses a cavity, it will report two objects. Objects are numbered starting from 0, ordered from nearest to farthest. By default, the closest pixel for each object is used to determine how near it is.

`get_depth_scanline_object_nearest_x` [Category: Depth]

Format: `int get_depth_object_nearest_x(int obj);`
For the specified object returns the coordinate value for the farthest pixel detected.

`get_depth_scanline_object_nearest_y` [Category: Depth]

Format: `int get_depth_object_nearest_y(int obj);`
For the specified object returns the coordinate value for the farthest pixel detected.

`get_depth_scanline_object_nearest_z` [Category: Depth]

Format: `int get_depth_object_nearest_z(int obj);`
For the specified object returns the coordinate value for the nearest pixel detected.

`get_depth_scanline_object_center_x` [Category: Depth]

Format: `int get_depth_object_center_x(int obj);`
For the specified object returns the coordinate value for the farthest pixel detected.

`get_depth_scanline_object_center_y` [Category: Depth]

Format: `int get_depth_object_center_y(int obj);`
For the specified object returns the coordinate value for the farthest pixel detected.

`get_depth_scanline_object_center_z` [Category: Depth]

Format: `int get_depth_object_center_z(int obj);`
For the specified object returns the coordinate value for the center pixel.

`get_depth_scanline_object_farthest_x` [Category: Depth]

Format: `int get_depth_object_farthest_x(int obj);`
For the specified object returns the coordinate value for the farthest pixel detected.

`get_depth_scanline_object_farthest_y` [Category: Depth]

Format: `int get_depth_object_farthest_y(int obj);`
For the specified object returns the coordinate value for the farthest pixel detected.

`get_depth_scanline_object_farthest_z` [Category: Depth]

Format: `int get_depth_object_farthest_z(int obj);`
For the specified object returns the coordinate value for the farthest pixel detected.

`get_depth_scanline_object_size` [Category: Depth]

Format: `int get_depth_object_size(int obj);`
For the specified object returns the distance on the scanline between the leftmost and rightmost pixel detected.

`get_depth_scanline_object_angle` [Category: Depth]

Format: `int get_depth_object_angle(int obj);`
For the specified object returns the angular measure (in degrees) from the leftmost pixel on the scanline to the right most. Positive is counterclockwise.

Example program for using a depth sensor and a graphical representation to find the distance to an object

```
int main()
{
   int r, g, b, row, col, val;
   depth_open(); // initiate depth sensor
   graphics_open(320,240); // open up graphics window (full screen on KIPR Robot Controller)
   while(!get_mouse_left_button()) { // loop until mouse is clicked (screen is tapped)
      depth_update(); // get a new depth image
      for(row=0; row<240; row++) { // loop through rows
         for(col=0; col<320; col++) { // loop through columns in current row
            val = get_depth_value(row,col) ; // get distance for pixel in mm
            if(val == 0)  // if too close or not a valid value, don't color
               continue;
            else if (val > 1530) { // if more than 1.53m away color in gray scale
               val = (5000-val)/20; // range is from 173 on down (towards darker)
               graphics_pixel(col,row,val,val,val); // paint using r=g=b (gray scale)
            }
            else { // and otherwise ...
               val=val > 510 ? val-510 : 0; // rerange the distance inward by 510mm
```

```
                r=val > 510 ? 0 : 255-val/2;  // increase red for closer
                g=val > 510 ? 255-(val-510)/2 : val/2; // greenish for mid values
                b=val > 510 ? val/2-255 : 0;  // increase blue for farther
                graphics_pixel(col,row, r,g,b); // draw the pixel
            }
         }
      }
      graphics_update(); // paint the screen using the new pixel valuess
   }
   get_mouse_position(&col,&row); // where was screen tapped?
   depth_close(); graphics_close();  //close sensor and graphics window
    // Display the distance (z coordinate) to the pixels's point in space
   printf("Distance to pixel %i,%i is %imm\n\n\n",col,row,get_depth_value(row,col));
}
```

Create Function Library

The functions in the KIPR Robot Controller Library for controlling an iRobot Create module provide an interface between a program on the KIPR Robot Controller and the Create Open Interface (http://www.irobot.com/filelibrary/pdfs/hrd/create/Create%20Open%20Interface_v2.pdf). Each Create function in the KIPR Robot Controller Library is designed to send the Create a series of Open Interface commands that cause the Create to perform some action. A script of Open Interface commands can also be downloaded to the Create to be played later (see the library function `create_write_byte`).

The KIPR Robot Controller communicates with the iRobot Create via a (TTL) serial connection. Functions included in the KIPR Robot Controller Library for the iRobot Create 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 traveled, etc). They also provide a means for the KIPR Robot Controller to directly control an iRobot Create.

A Create script is a (limited) sequence of pre-defined 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. See the example program below for more information on user defined scripts.

The KIPR Robot Controller Library functions for moving the iRobot Create are non-blocking, so when a movement command is sent to the iRobot Create, its trajectory will continue until a different Create movement function is executed on the KIPR Robot Controller. In contrast, once a script is started on the Create it will run to completion, during which time the connection to the KIPR Robot Controller is ignored. The iRobot Create may also be used to play MIDI music. Up to 16 songs may be downloaded to the iRobot Create from a song array on the KIPR Robot Controller. See the Create Open Interface manual for details on note and duration codes.

The library functions for accessing Create sensor data return either the requested data or an error code. A return value greater than 100,000 is used to indicate an error occurred, where the error number is given by 100,000 + <*Create-Serial-Interface-Packet-Number*>. For example, a code of 100,007 indicates an error occurred when requesting bumper or wheel drop sensor status. Sensor packet numbers range from 7 to 42 as described in the Create Open Interface manual.

The Create Open Interface manual has details concerning the physical characteristics of the iRobot Create module and descriptions of how to use the various byte code commands.

Create Serial Interface Functions

`create_clear_serial_buffer` [Category: Create Serial Interface Function]

Format: `void create_clear_serial_buffer();`
Clears the internal serial interface buffer of any unaccessed send/receive data.

`create_connect()` [Category: Create Serial Interface Function]

Format: `int create_connect;`
Establishes a USB serial connection between the KIPR Robot Controller 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 Robot Controller to the Create. By default, the Create starts in `create_safe` mode.

`create_disconnect` [Category: Create Serial Interface Function]

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 Function]

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 pointed to by `data`. 1 is returned on read success, 0 on failure.

`create_write_byte` [Category: Create Serial Interface Function]

Format: `void create_write_byte(char byte);`
Uses the serial interface to have the KIPR Robot Controller send the specified byte to the iRobot Create. The purpose of this function is for writing functions for supplying a series of byte commands to the Create to have it perform some action, or to load a song, or to load a script.

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 execute all commands, but will disconnect and stop 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);`
Moves the Create in an arc (see below for point turns and straight). Speed range for all commands is -500 to 500mm/sec and radius is -2000 to 2000mm.

`create_drive_direct` [Category: Create Movement Function]

Format: `void create_drive_direct(int r_speed, int l_speed);`
Moves the Create by setting the speed for the left side and right side separately, speed range -500 to 500mm/sec.

`create_drive_straight` [Category: Create Movement Function]

Format: `void create_drive_straight(int speed);`
Moves the Create straight at speed in the range -500 to 500mm/sec.

`create_spin_CCW` [Category: Create Movement Function]

Format: `void create_spin_CCW(int speed);`
Spins the Create Counter Clock Wise with edge speed of speed of -500 to 500mm/sec.

`create_spin_CW` [Category: Create Movement Function]

Format: `void create_spin_CW(int speed);`
Spins the Create Clock Wise with edge speed of -500 to 500mm/sec.

`create_stop` [Category: Create Movement Function]

Format: `void create_stop();`
Stops the Create 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);`
Resets the distance accumulation value for `get_create_distance` to the value specified.

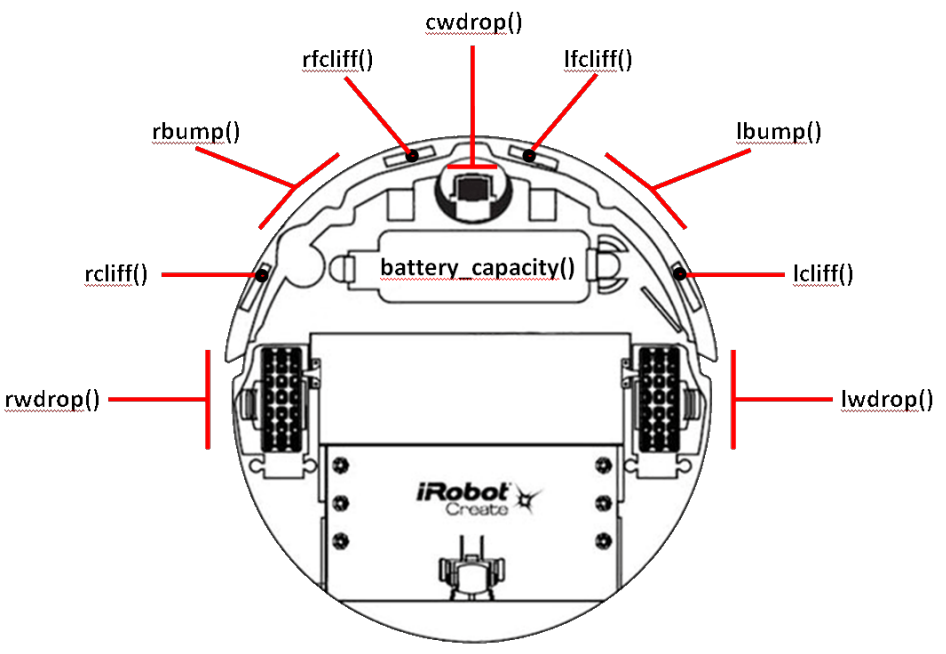
`set_create_normalized_angle` [Category: Create Movement Function]

Format: `void set_create_normalized_angle(int angle);`
Resets the normalized angle accumulation value for `get_create_normalized_angle` to the value specified.

`set_create_total_angle` [Category: Create Movement Function]

Format: `void set_create_total_angle(int angle);`
Resets the total angle accumulation value for `get_create_total_angle` to the value specified.

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 of the connector in the Create 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 the front 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 the 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, 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 the 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 a surface that doesn't reflect IR (e.g., black) or over a cliff, 0 otherwise.

`get_create_rcliff_amt` [Category: Create Sensor Function]

Format: `int get_create_rcliff_amt();`
Returns the right cliff sensor (analog) reading as a 12 bit value (0 to 4095).

`get_create_rfcliff` [Category: Create Sensor Function]

Format: `int get_create_rfcliff();`
Returns 1 if right front cliff 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_rlwdrop();`
Returns 1 if right wheel has dropped, 0 otherwise.

`get_create_vwall` [Category: Create Sensor Function]

Format: `int get_create_vwall();`
Returns 1 if an 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 Sensor Function]

Format: `int get_create_battery_capacity();`
Returns the battery capacity in mAh. This function is seldom used in practice.

`get_create_battery_charge` [Category: Create Sensor Function]

Format: `int get_create_battery_charge();`
Returns the battery charge in mAh.

`get_create_battery_charging_state` [Category: Create Sensor Function]

Format: `int get_create_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 Sensor Function]

Format: `int get_create_battery_current();`
Returns the current flow in mA.

`get_create_battery_temp` [Category: Create Sensor Function]

Format: `int get_create_battery_temp();`
Returns the battery temperature in degrees C.

`get_create_battery_voltage` [Category: Create Sensor Function]

Format: `int get_create_battery_voltage();`
Returns the battery voltage in mV.

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 Function]

Format: `void create_cover();`
Simulates a Roomba covering a room.

`create_demo` [Category: Create Function]

Format: `void create_demo(int d);`
Runs built in demos (see (Create Open Interface Manual).

`create_cover_dock` [Category: Create Function]

Format: `void create_cover_dock();`
Create roams around until its IR detects an iRobot IR docking station and then attempts to dock for recharging.

Create 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 causes 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 Manual for details.

Example program for using the KIPR Robot Controller to control a Create module

```
/* 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, 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); // reset the cumulative distance traveled
    set_create_total_angle(0); // reset the cumulative angle turned through
    create_drive(200, 250); // start move in an arc
    msleep(10000); // sleep for 10 seconds and stop
    create_stop();
    printf("\nResults:\n");
    printf("  distance = %d mm\n", get_create_distance(0.1));
    printf("  angle = %d degrees\n", get_create_total_angle(0.1));
    printf("\ndisconnecting from Create\n");
    create_disconnect();
}
```

Example Create script definition using the KIPR Robot Controller

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.

In this example, a function is used to download a sequence of byte commands to the iRobot Create to load

a script. The example script is designed to cause the Create to move 1/2 meter at 500 mm/sec (uninterruptible). The script definition follows the first two bytes sent to the Create (byte command 152, and a byte whose numeric value gives the number of bytes that follow). Note that the number of bytes for the script is just the count of the `create_write_byte` function calls used after the first two. See the Create Open Interface manual for information on the byte commands.

```
#define RUN_SCRIPT create_write_byte(153) // macro to run the currently loaded script
void make_drive_script(int dist, int speed) {
   create_write_byte(152);         // specifies start of script definition
   create_write_byte(13);          // specifies number of bytes to follow,(defining the
script)
   create_write_byte(137);         // drive command (speed and turn radius in next 4 bytes)
   create_write_byte(speed >> 8); // send speed high byte (bits 8-15 shifted to 0-7)
   create_write_byte(speed);       // send speed low byte
   create_write_byte(128);         // send hex 80
   create_write_byte(0);           // send hex 00 (special case: turn radius hex 8000 or 7FFF
is straight)
   create_write_byte(156);         // wait for distance done (in mm)
   create_write_byte(dist >> 8);  // send dist high byte
   create_write_byte(dist);        // send dist low byte
   create_write_byte(137);         // stop move by changing speed and radius to 0
   create_write_byte(0);           // send high byte (0 speed)
   create_write_byte(0);           // send low byte (0 speed)
   create_write_byte(0);           // null turn radius
   create_write_byte(0);           // null turn radius
   // end of script (13 bytes)
}
int main()
{
   // program to load and test the above script
   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(2000); // 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();
}
```

Categorized Function List (cross-linked)

Each function listed is provided with a cross link to its library description.

Botball

- `void run_for(double delay, <function_name>);` [§]

- void shut_down_in(double delay); [§]
- void wait_for_light(int light_port_); [§]

Create

Serial Interface

- void create_clear_serial_buffer(); [§]
- int create_connect(); [§]
- void create_disconnect(); [§]
- int create_read_block(char *data, int count); [§]
- void create_write_byte(char byte); [§]
- void create_full(); [§]
- void create_start(); [§]
- void create_passive(); [§]
- void create_safe(); [§]
- int get_create_mode(); [§]

Movement

- void create_drive(int speed, int radius); [§]
- void create_drive_direct(int r_speed, int l_speed); [§]
- void create_drive_straight(int speed); [§]
- void create_spin_CCW(int speed); [§]
- void create_spin_CW(int speed); [§]
- void create_stop(); [§]
- int get_create_distance(); [§]
- int get_create_normalized_angle(); [§]
- int get_create_overcurrents(); [§]
- int get_create_requested_left_velocity(); [§]
- int get_create_requested_radius(); [§]
- int get_create_requested_right_velocity(); [§]
- int get_create_requested_velocity(); [§]
- int get_create_total_angle(); [§]
- void set_create_distance(int dist); [§]
- void set_create_normalized_angle(int angle); [§]
- void set_create_total_angle(int angle); [§]

Battery

- int get_create_battery_capacity(); [§]
- int get_create_battery_charge(); [§]
- int get_create_charging_state(); [§]
- int get_create_battery_current(); [§]
- int get_create_battery_temp(); [§]
- int get_create_battery_voltage(); [§]

Sensors

- int get_create_advance_button(); [§]
- int get_create_bay_DI(); [§]
- int get_create_bay_AI(); [§]
- int get_create_cwdrop(); [§]
- int get_create_infrared(); [§]
- int get_create_lbump(); [§]
- int get_create_lcliff(); [§]
- int get_create_lcliff_amt(); [§]
- int get_create_lfcliff(); [§]
- int get_create_lfcliff_amt(); [§]
- int get_create_lwdrop(); [§]
- int get_create_number_of_stream_packets(); [§]
- int get_create_play_button(); [§]
- int get_create_rbump(); [§]
- int get_create_rcliff(); [§]
- int get_create_rcliff_amt(); [§]
- int get_create_rfcliff(); [§]
- int get_create_rfcliff_amt(); [§]
- int get_create_rlwdrop(); [§]
- int get_create_vwall(); [§]
- int get_create_wall(); [§]
- int get_create_wall_amt(); [§]

Built in Scripts

- void create_spot(); [§]
- void create_cover(); [§]
- void create_demo(int d); [§]
- void create_cover_dock(); [§]

LEDs and Music

- void create_advance_led(int on); [§]
- int get_create_song_number(); [§]
- int get_create_song_playing(); [§]
- void create_load_song(int num); [§]
- void create_play_led(int on); [§]
- void create_play_song(int num); [§]
- void create_power_led(int color, int brightness); [§]

Graphics

- int graphics_open(int width, int height); [§]
- void graphics_close(); [§]
- void graphics_update(); [§]
- void graphics_clear(); [§]
- void graphics_fill(int r, int g, int b); [§]

- void graphics_pixel(int x, int y, int r, int g, int b); [§]
- void graphics_line(int x1, int y1, int x2, int y2, int r, int g, int b); [§]
- void graphics_circle(int cx, int cy, int radius, int r, int g, int b); [§]
- void graphics_circle_fill(int cx, int cy, int radius, int r, int g, int b); [§]
- void graphics_rectangle(int x1, int y1, int x2, int y2, int r, int g, int b); [§]
- void graphics_rectangle_fill(int x1, int y1, int x2, int y2, int r, int g, int b); [§]
- void graphics_triangle(int x1, int y1, int x2, int y2, int x3, int y3, int r, int g, int b); [§]
- void graphics_triangle_fill(int x1, int y1, int x2, int y2, int x3, int y3, int r, int g, int b); [§]
- int get_mouse_position(int *x, int *y); [§]
- int get_mouse_left_button(); [§]

Math

- double atan(double angle); [§]
- double cos(double angle); [§]
- double exp(double num); [§]
- double exp10(double num); [§]
- double log(double num); [§]
- int log10(double num); [§]
- double pow(double x, double y); [§]
- int rand(int m); [§]
- double sin(double angle); [§]
- double sqrt(double num); [§]
- void srand(int s); [§]
- double tan(double angle); [§]

Motors

- void alloff(); [§]
- void ao(); [§]
- void bk(int m); [§]
- void block_motor_done(int m); [§]
- void bmd(int m); [§]
- int get_motor_done(int m); [§]
- void clear_motor_position_counter(int motor_nbr); [§]
- void fd(int m); [§]
- void freeze(int m); [§]
- int get_motor_done(int m); [§]
- int get_motor_position_counter(int m); [§]
- int get_pid_gains(int motor, int p, int i, int d, int pd, int id, int dd); [§]
- void mav(int m, int vel); [§]
- void motor(int m, int p); [§]
- void move_at_velocity(int m, int vel); [§]
- void move_relative_position(int m, int speed, int pos); [§]
- void move_to_position(int m, int speed, int pos); [§]
- void mrp(int m, int vel, int pos); [§]
- void mtp(int m, int vel, int pos); [§]
- void off(int m); [§]
- int set_pid_gains(int motor, int p, int i, int d, int pd, int id, int dd); [§]
- int setpwm(int m, int dutycycle); [§]

Output

- void beep(); [§]
- void console_clear(); [§]

- void display_clear(); [§]
- void display_printf(int col, int row, char s[], . . .); [§]
- void extra_buttons_show(); [§]
- void extra_buttons_hide(); [§]
- void set_a_button_textchar txt[]); [§]
- void set_b_button_textchar txt[]); [§]
- void set_c_button_textchar txt[]); [§]
- void set_digital_output(int port, int inout); [§]
- void set_digital_value(int port, int value); [§]
- void set_x_button_textchar txt[]); [§]
- void set_y_button_textchar txt[]); [§]
- void set_z_button_textchar txt[]); [§]

Threads

- thread thread_create(<*function_name*>); [§]
- void thread_destroy(<*thread id*>); [§]
- void thread_start(<*thread id*>); [§]
- void thread_wait(<*thread id*>); [§]
- void run_for(double sec, void <*function_name*>); [§]

Sensors

- int a_button(); [§]
- int a_button_clicked(); [§]
- int accel_x(); [§]
- int accel_y(); [§]
- int accel_z(); [§]
- int analog(int p); [§]
- int analog_et(int p); [§]
- int analog8(int p); [§]
- int analog10(int p); [§]
- int any_button(); [§]
- int b_button(); [§]
- int b_button_clicked(); [§]
- int c_button(); [§]
- int c_button_clicked(); [§]
- int digital(int p); [§]
- int get_analog_pullup(int port); [§]
- int get_digital_pullup(int port); [§]
- double power_level(); [§]
- int get_extra_buttons_visible(); [§]
- void set_analog_pullup(int port, int pullupTF); [§]
- void set_digital_pullup(int port, int pullupTF); [§]
- int side_button(); [§]
- int side_button_clicked(); [§]
- int x_button(); [§]
- int x_button_clicked(); [§]
- int y_button(); [§]
- int y_button_clicked(); [§]
- int z_button(); [§]
- int z_button_clicked(); [§]

Servos

- void disable_servo(int p); [§]
- void disable_servos(); [§]

- `void enable_servo(int p); [§]`
- `void enable_servos(); [§]`
- `int get_servo_enabled(int srv); [§]`
- `int get_servo_position(int srv); [§]`
- `int set_servo_position(int srv, int pos); [§]`

Time

- `void msleep(int msec); [§]`
- `double seconds(); [§]`

Vision

- `void camera_close(); [§]`
- `int camera_load_config(char name[]); [§]`
- `int camera_open(); [§]`
- `int camera_open_at_res(int res_numb); [§]`
- `int camera_open_device(int number, int res_numb); [§]`
- `int camera_update(); [§]`
- `int get_channel_count(); [§]`
- `int get_object_area(int channel, int object); [§]`
- `rectangle get_object_bbox(int channel, int object); [§]`
- `point2 get_object_center(int channel, int object); [§]`
- `point2 get_object_centroid(int channel, int object); [§]`
- `int get_code_num(int channel, int object); [§]`
- `double get_object_confidence(int channel, int object); [§]`
- `int get_object_count(int channel); [§]`
- `char *get_object_data(int channel, int object); [§]`
- `int get_object_data_length(int channel, int object); [§]`

Depth

- `void depth_close(); [§]`
- `int depth_open(); [§]`
- `int depth_update(); [§]`
- `int get_depth_value(int row, int col); [§]`
- `point3 get_depth_world_point(int row, int col); [§]`
- `int get_depth_world_point_x(int row, int col); [§]`
- `int get_depth_world_point_y(int row, int col); [§]`
- `int get_depth_world_point_z(int row, int col); [§]`

Botball

Three functions are included in the KIPR Robot Controller Library to assist programmers in writing programs that meet the basic requirements for a Botball game:

- `wait_for_light(`*<port_num>*`)`, a function for ensuring the robot doesn't begin operating until the starting light comes on.
- `shut_down_in(`*<time>*`)`, a function which ensures the robot stops operating within the time limits for a Botball game.
- `run_for(`*<time>*`, `*<func_name>*`)`, which is used to start a function, but not allow it to run beyond a specified time.

For the typical Botball program, the first executable statement will be

```
wait_for_light(<port_num>);
```

If there are any other set up routines being used, then usually `wait_for_light` will immediately follow them. `wait_for_light` takes the robot operator through a process of using the starting light to set calibration values for the IR light sensor plugged into the specified port. The purpose is to set two values, one to check the sensor reading with light off and the other with light on. The difference between the two values has to be large enough for the sensor to determine if the light is off or on. If the sensor is suitably shielded and is positioned so that a successful calibration is achieved, `wait_for_light` calls for "hands off" and blocks further execution until the light is turned on. When "light on" is detected by the sensor, the program resumes and runs the robot according to program design. If problems occur in the calibration process, the user is notifited and the process repeats after a brief pause.

The first statement in a Botball programm following `wait_for_light` is usually a call to the Botball function

```
shut_down_in(<time>);
```

which will shutdown all motors and the Create module (if being used) once the specified time has expired, then terminate the program.

In contrast to completely killing the program using `shut_down_in`, when program design requires performing some activity after the robot has completed its primary task, the function

```
run_for(<time>, <func_name>);
```

can be used to execute the specified function, halting it once the specified time has elapsed (an action that occurs only if the function is still running). The basic idea is to separate the primary task out of `main` and write it as a function so it can be run using `run_for`. One simple instance where this is useful is at game end when a servo needs to remain on to maintain its position. The necessary command can be executed after using `run_for` to run the primary task. This approach puts the burden back on the programmer to make sure all other servo activity is halted and all drive motors and the Create module are stopped within the game's time limit.

The library functions for Botball are:

`run_for(<time>, <function_name>);`

> Runs the specified function and tracks its execution until either the function finishes or the specified number of seconds has elapsed. If the specified number of seconds has elapsed, `run_for` terminates execution for the function.

`shut_down_in(<time>);`

> Starts a timing routine that will end program execution, turn off all motors, and issue a stop command to the Create when the specified amount of time has elapsed.

`wait_for_light(<port>);`

> Steps the user through a calibration procedure to establish on/off light levels for a sensor plugged in the specified port. If calibration does not provide enough discrimination between light off and light on, the program cycles for another try. If there is enough discrimination, after "light off", the program blocks continued execution of the function until "light on", at which point execution resumes. `wait_for_light` is normally placed at the beginning of a program designed to operate a robot, so that after calibration, when the light comes on the robot will start autonomously.

Example program for the KIPR Robot Controller using a Botball program format

```
int main()
{
   double s;
   int i;
   /* Botball calibration: determine if light sensor can discriminate between light and dark
*/
   wait_for_light(3);  // light sensor in analog port 3
   /* Botball timing: limit is 120 seconds; e.g., shut_down_in(119.5); */
   shut_down_in(10.5); // stop execution once 10.5 seconds have elapsed
   /* Botball program logic would be next ... the stuff below is just to keep the program
running for awhile */
   display_clear(); // clear display for display_printf
   s=seconds(); // system on time at start of run
   for(i=0; i<15; i++) {
      display_printf(1,1,"%d. time elapsed = %.2f     ", i, seconds()-s);
      msleep(1000); // sleep for a second
   }
   display_printf(1,3,"done");
}
```

Threads

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 Robot Controller, 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 gets its share of time. This continues until all the active threads have gotten their slice of time and then thread processing repeats. The KIPR Robot Controller's processor is fast enough so that 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 when it is not in a section of code that might cause a conflict. Process IDs may also be stored in global variables of type `thread` 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 will never finish otherwise).

Since the operating system limits how many threads can be created, it is inadvisable to create threads in a loop. Good thread management is to destroy a thread rather than leaving it hanging around once it is no longer being used. Threads not destroyed by program end may still count against the operating system limit when the program is run again (unless of course the system has been rebooted). This may result in a program which for no observable reason ceases to work properly. It is up to the program to keep track of a thread it creates, and there are no means (short of rebooting) to destroy a thread after the fact.

The library functions for controlling threads are:

`thread_create(`*<function_name>*`);`

Creates a thread for running the specified function and returns a value of type `thread`, which is the thread ID to be used for running the thread as an independent process.

`thread_destroy(`*<thread_id>*`);`

Deactivates the specified thread if it is active (stopping its associated function) and destroys its thread ID.

`thread_start(`*<thread_id>*`);`

Activates the specified thread by running its associated process in the thread.

`thread_wait(`*<thread_id>*`);`

Suspends execution of the function that calls `thread_wait` while the specified thread remains active. It's purpose is to synchronize threads by suspending further execution of a function until the selected active threads have finished.

Example program for the KIPR Robot Controller using threads

This example illustrates how to detect a button press even it occurs while the main process is paused.

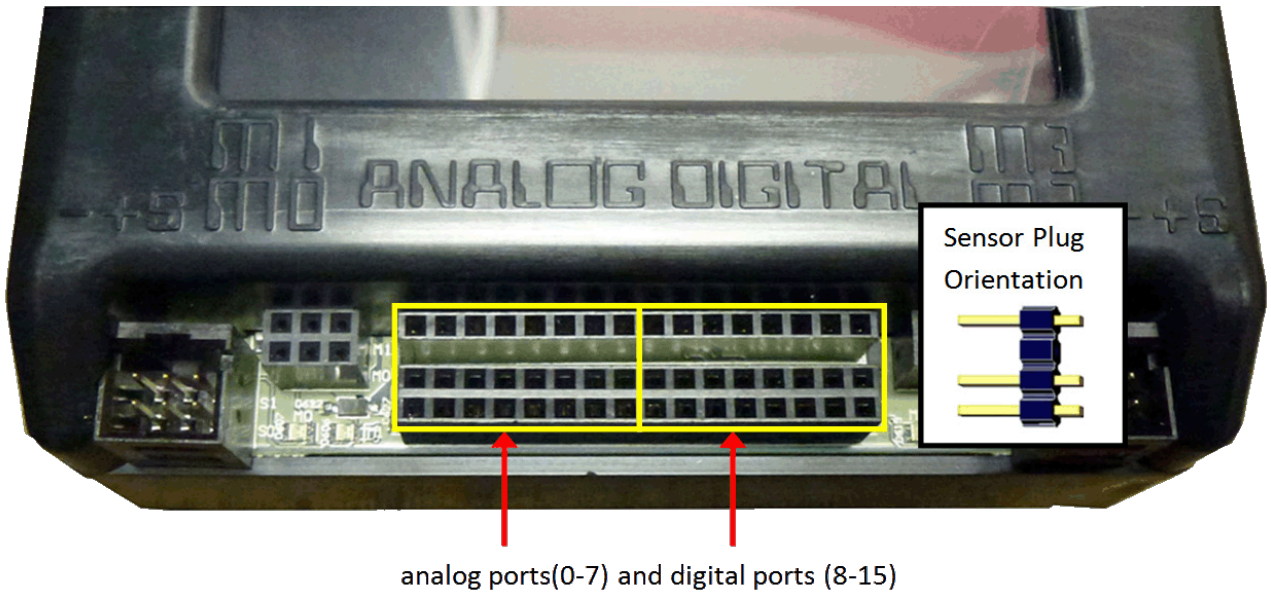
```
int flag=0; // global flag to signal when side button pressed
void chksens() {
   while (1) {
      if (side_button()) flag = 1;
      msleep(100); // check side button every 1/10th second
   }
}
 main() {
   int cnt = 0;
   thread tid; // thread variable for holding thread id
   tid = thread_create(chksens); // create a thread for chksens
   thread_start(tid); // start chksens running in its thread
   while (flag == 0) { // button press during sleep is still caught
      display_printf(1,2,"elapsed time %d   ",++cnt);
      msleep(1000);
   }
   thread_destroy(tid); // remove the thread
   display_printf(0,4,"done");
}
```

KIPR Robot Controller Images

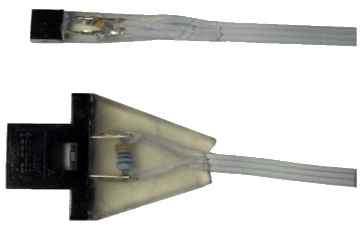
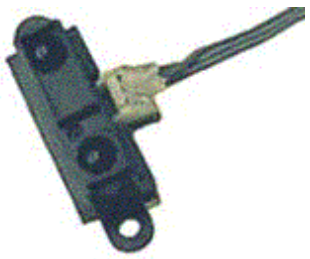
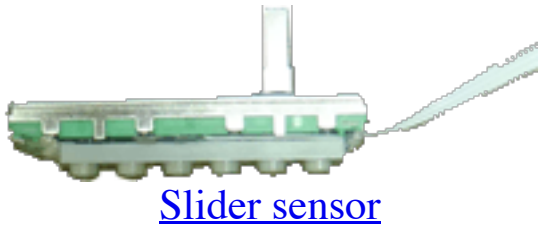
Analog and Digital Sensor Images

The image below is the front of the KIPR Robot Controller showing where the analog and digital sensor

ports are located.



Examples of digital and analog sensors used with the KIPR Robot Controller

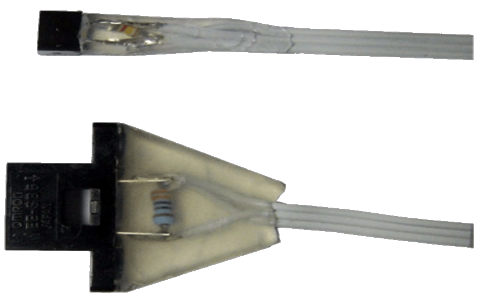
 IR light sensor (analog)	 IR reflectance sensors (analog)	 IR "E.T." Distance Sensor (floating analog)
 Touch Sensors (digital)		 IR "Break Beam" Sensor (digital)
 Slider sensor (analog)	 Potentiometer (Pot) Sensor (analog)	 Ultrasonic rangefinder (SONAR)

Infrared light sensor (analog)

- Analog sensor
- Connect to ports 0-7
- Access with function `analog(`*<port#>*`)`

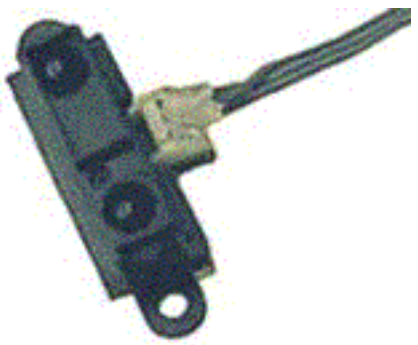
- Low values indicate lots of Infrared
- High values indicate low Infrared
- Sensor is somewhat directional and can be made more so using an opaque tube or Lego to block out extraneous light. Sensor can be attenuated by placing paper in front.

Infrared large and small reflectance sensors (analog)



- Analog sensor
- Connect to ports 0-7
- Access with function `analog(`*<port#>*`);`
- Values measure IR reflection from a surface, allowing a robot to detect surface features such as a black tape line
- Dark surfaces produce high readings (low IR reflectivity) and light surfaces produce low readings (high IR reflectivity)
- Proximity of sensor to surface affects the difference in readings, so can be used to detect drop-offs
- Sensor has a reflectance range of about 3 inches

Infrared "E.T." Distance Sensor (floating analog)



- Floating analog sensor (pullup resistor for port must be disabled for readings to be meaningful)
- Connect to port 0-7
- Access with function `analog_et(`*<port#>*`);`
- Low values indicate large distance
- High values indicate distance approaching ~4 inches
- Range is 4-30 inches. Result is approximately 1/(d)^2. Objects closer than 4 inches will have the same readings as those far away.
- Sharp Electronics part number GP2D12
- Sensor shines a narrow infrared beam, and measures the angle of the beam return using a position-sensitive detector (PSD):

Touch sensors (digital)

- Digital switch sensors
- Connect to ports 8-15
- Access with function `digital(<port#>);`
- 1 indicates switch is closed
- 0 indicates switch is open
- These make good bumpers and can be used for limit switches on an actuator

Infrared "Break Beam" Sensor (digital)

- Digital sensor
- Connect to ports 8-15
- Access with function `digital(<port#>)`
- 1 indicates slot is empty
- 0 indicates slot is blocked
- Can be used for alignment; e.g., a winch with a pin which when blocking the slot indicates alignment is reached
- Can also be used for encoders, where the edge of a wheel with holes around its perimeter passes through the slot, alternately breaking the beam

Analog slider

- Analog sensor
- Connect to ports 0-7
- Access with function `analog(<port#>);`
- Values range from 0 to 2047

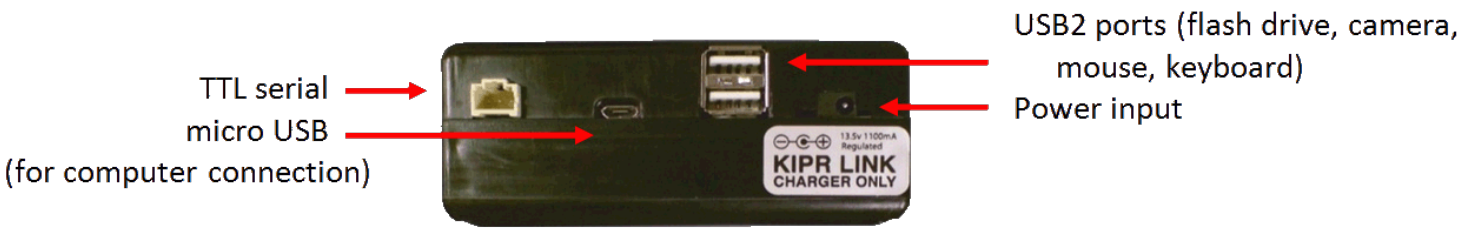
Analog pot (potentiometer)

- Analog sensor
- Connect to ports 0-7

- Access with function `analog(<port#>);`
- Values range from 0 to 2047

USB Sensor Images

USB sensors plug into one of the two USB2 ports on the back of the KIPR Robot Controller



Specialized function libraries are provided for

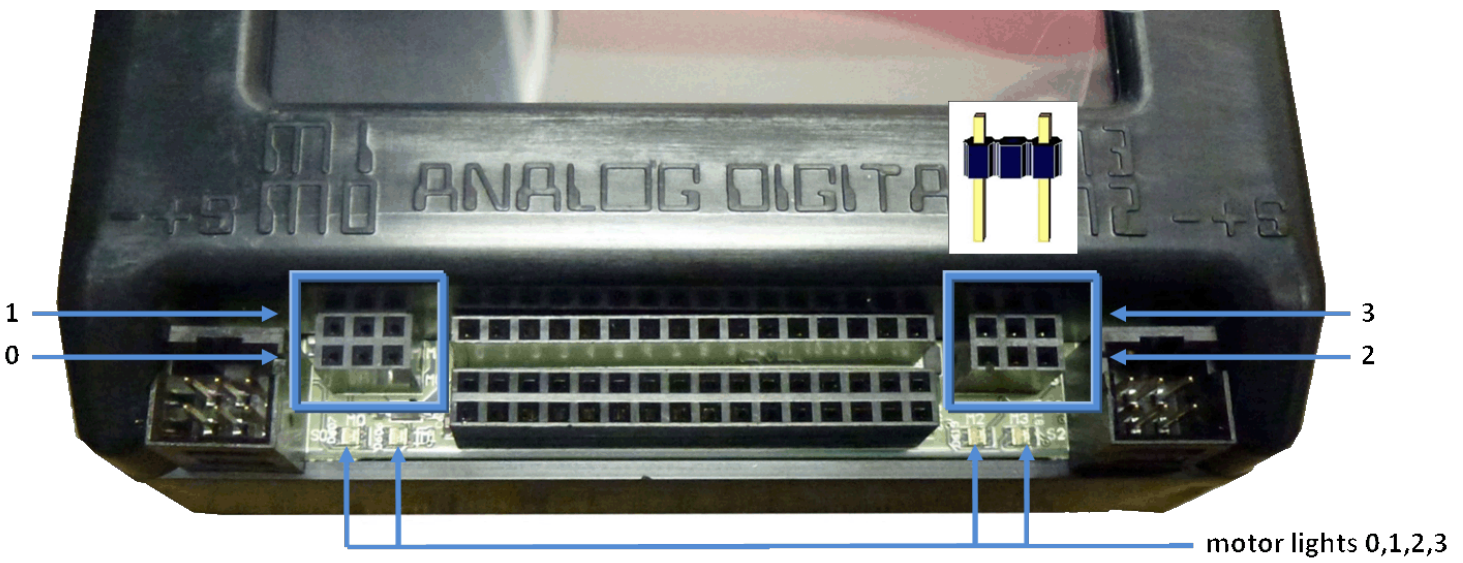
- Vision (for use with a USB web camera)
- Depth analysis (for use with an ASUS Xtion)

A web camera used with KIPR Robot Controller
(see the Vision Function List)

ASUS Xtion used with KIPR Robot Controller
(see the Depth Function List)

DC Motor Images

The image below is the front of the KIPR Robot Controller showing where the DC motor ports are located.

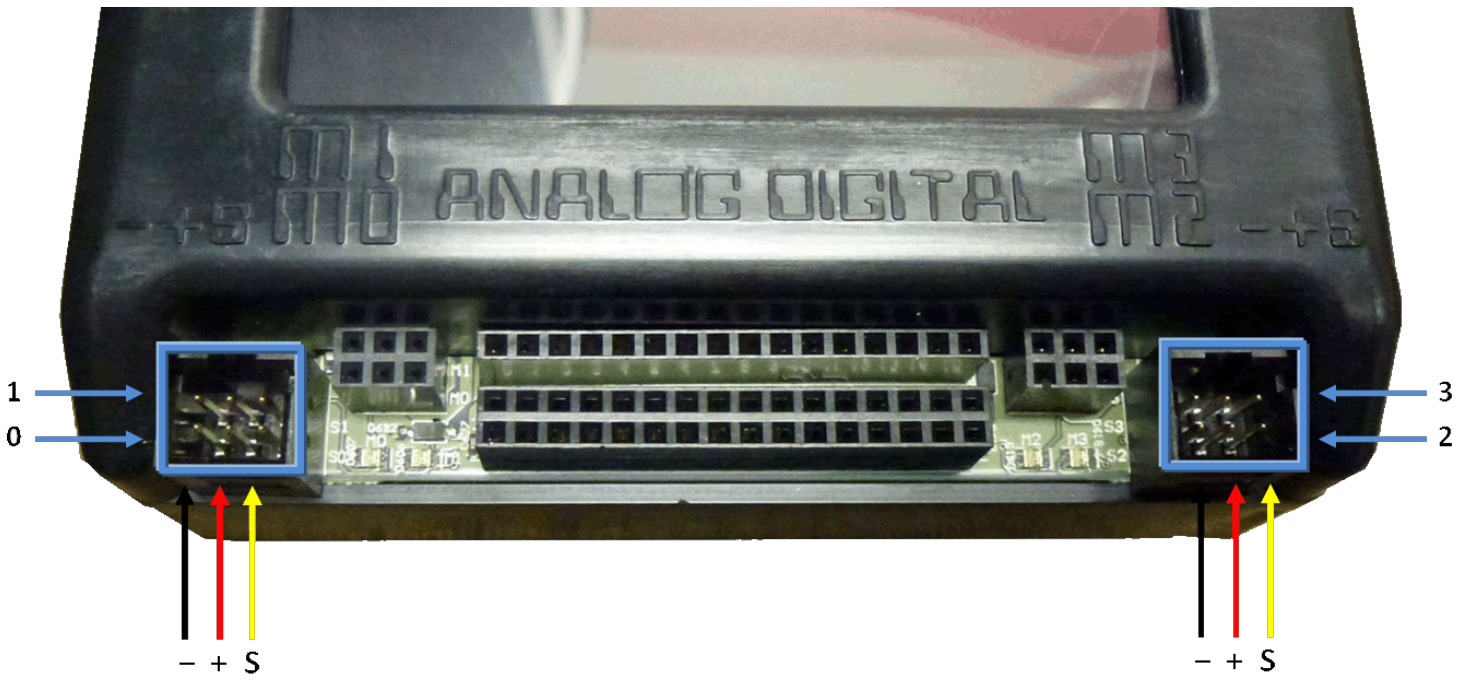


Example DC motor used with the KIPR Robot Controller
(modified servo - potentiometer removed and wiring with two prong plug installed).

DC motors are controlled using the KIPR Robot Controller motor functions.

Servo Images

The image below is the front of the KIPR Robot Controller showing where the servo ports are located.



Example servo motors used with the KIPR Robot Controller
(hobbyist servo - used as packaged by manufacturer)

Servo motors

Servo motor plug

Servos are controlled by using the KIPR Robot Controller servo functions.

KIPR Robot Controller Simulator

When the KIPR Software Suite is installed, in addition to the KISS IDE, a graphical simulator is installed which can be selected as a target for a project being developed using the KISS IDE. The simulator provides a simulated robot and a stylized operating environment for the robot, including a simulated (Botball style) starting light. For motors and sensors incorporated into the simulated robot, side panels show motor and sensor response values. A representation of the KIPR Robot Controller console screen is also provided for display output. The simulated robot can be arbitrarily positioned in its environment using keyboard entry and mouse functions, including turning the simulated light on/off.

The simulator provides an effective means for trying out functions in the KIPR Robot Controller Library and for testing small examples when learning how to use a library function. More fundamentally, it provides a user friendly means for obtaining visual feedback for those who are learning to program in C.

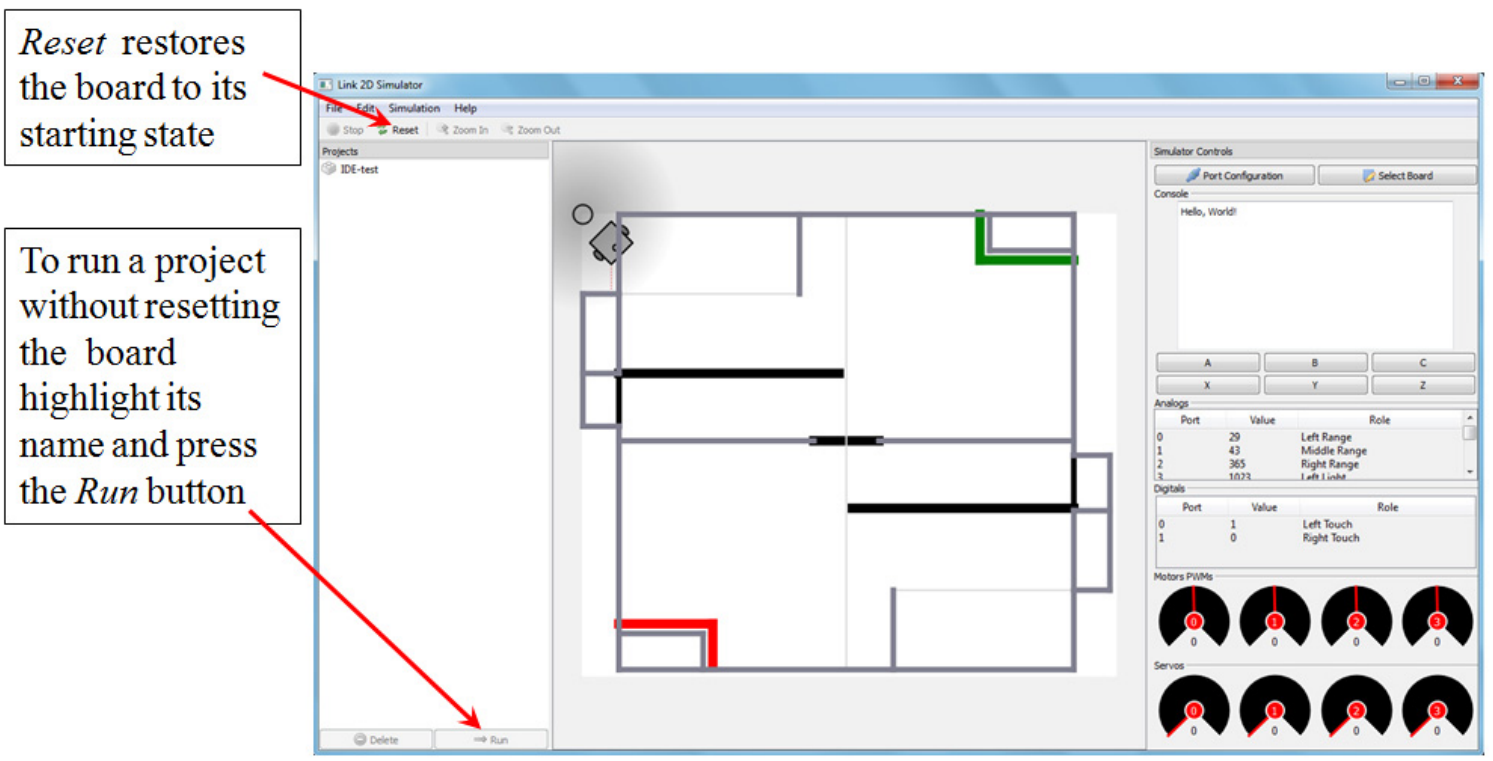
Perhaps most importantly, the simulator provides means for testing KIPR Robot Controller program code before downloading a project to a KIPR Robot Controller, speeding up the process of obtaining a working program for a KIPR Robot Controller controlled robot. No simulator can anticipate every variation a real robot might encounter in performing its mission, but in general, simulation serves to limit the amount of program tweaking required to attain acceptable performance on the actual robot.

Running a Project

If a program's action does not involve the KIPR Robot Controller Library functions, its target can be "My Computer". When KIPR Robot Controller Library functions are being used, the target needs to be "My KIPR Robot Controller Simulator" or an attached KIPR Robot Controller. If no KIPR Robot Controller is attached, the simulator can be used to see what a program does.

- If the project is the only project open in the IDE, it is automatically the *active project*.
- If there is more than one project open in the IDE, clicking on the project name in the side panel will make it the active project.
- Clicking on either the *Compile* or the *Run* button in the IDE will send the active project to the target for

compilation, launching the simulator if it is the selected target (and not already running). If there are any compilation errors they are reported back to the IDE to be displayed in its results panel. If the *Run* button was used in the IDE, the target will also run the project. For the Simulator, the names of projects loaded into the Simulator will be displayed in a side panel (analogous to the *Programs* listing on the KIPR Robot Controller).



For executing a project already loaded in the Simulator

When the target for the active project is the simulator

- Clicking *Run* in the IDE saves, loads, compiles, and runs the active project in the simulator.
- Clicking *Compile* in the IDE saves, loads, and compiles the active project in the simulator.
    - Compiling allows you to position the robot and the light on the simulator game board before the program is run.
    - Unless you are using a custom board, compiling allows you to change the board the robot runs on to one of the alternative default boards.
    - Compiled projects are listed in the side panel and can be run by highlighting the name and clicking *Run* in the simulator.

Configuring the Simulator Starting Light, Robot, and Game Board

1. Positioning and using the light
    - The light is positioned by clicking on the light and dragging it.
    - Double clicking on the light toggles it on/off.
2. Positioning the robot
    - The robot is positioned by clicking on the robot and dragging it.
    - You can turn the robot by holding down the shift key while clicking on the nose of the robot, using it as a handle for turning.
    - You can then run your project by highlighting its name and clicking *Run*
3. Selecting a game board
    - You can either
        - provide a custom board for your project (See IDE *Help* for information on configuring a custom board)

      or
        - select one of the default boards under the SimulatorÂ's *Select Board* tab
    - If you have a custom board file in your project, the Simulator will not let you change to one of its default boards
    - The *Select Board* tab brings up a *Board Selection* window with the currently available default game

boards you can select from among