IC Programmers Manual for the Handyboard Controller

Programmers Manual Index

Introduction

Interactive C (IC for short) is a C language consisting of a compiler (with interactive command-line compilation and debugging) and a run-time machine language module. IC implements a subset of C including control structures (for, while, if, else), local and global variables, arrays, pointers, structures, 16-bit and 32-bit integers, and 32-bit floating point numbers.

IC works by compiling into pseudo-code for a custom stack machine, rather than compiling directly into native code for a particular processor. This pseudo-code (or *p-code*) is then interpreted by the run-time machine language program. This unusual approach to compiler design allows IC to offer the following design tradeoffs:

- **Interpreted execution** that allows run-time error checking. For example, IC does array bounds checking at run-time to protect against some programming errors.
- **Ease of design**. Writing a compiler for a stack machine is significantly easier than writing one for a typical processor. Since IC's p-code is machine-independent, porting IC to another processor entails rewriting the p-code interpreter, rather than changing the compiler.
- **Small object code**. Stack machine code tends to be smaller than a native code representation.
- **Multi-tasking**. Because the pseudo-code is fully stack-based, a process's state is defined solely by its stack and its program counter. It is thus easy to task-switch simply by loading a new stack pointer and program counter. This task-switching is handled by the run-time module, not by the compiler.

Since IC's ultimate performance is limited by the fact that its output p-code is interpreted, these advantages are taken at the expense of raw execution speed.

> *IC 5 was written by Kyle Machulis/KISS Institute for Practical Robotics. Portions of the code and the libraries are based on the public distribution of IC 4.x by Randy Sargent with assistance from Mark Sherman, and IC 2.8, written by Randy Sargent, Anne Wright and Fred Martin.*

Using IC Index

When IC is running and has a connection to a compatible processor board such as the Handy Board or RCX, C expressions, function calls, and IC commands may be typed in the command entry portion of the interaction window.

For example, to evaluate the arithmetic expression 1 + 2, type in the following:

```
1 + 2;
```

When this expression is entered from the interaction window, it is compiled by the console computer and then downloaded to the attached system for evaluation. The connected board then evaluates the compiled form and returns the result, which is printed on the display section of console interaction window.

To evaluate a series of expressions, create a C block by beginning with an open curly brace { and ending with a close curly brace }. The following example creates a local variable Land prints 10 (the sum of [illegible]) to the board's LCD screen:

```
{int i=3; printf("%d", i+7);}
```

IC Interface Index

Both new (unsaved) and saved files can be opened for editing in IC. A row of tabs lists the files that have been opened. Clicking a file's tab activates it for editing. The first tab for the interface is always the interaction window.

The **File** button has standard entries for **New**, **Open**, **Close**, **Save**, **Save As**, **Print**, and **Exit**. Under **File - Save As**, if no file name extension is supplied, IC automatically saves with the "[illegible]" extension.

To download the active file, simply click the download button. The active file will also be saved, unless it is new, in which case the user is prompted for a "save as" file name. Remark: a preprocessor command IXVH has been added to IC to specify any other saved files (personal libraries) that need to be downloaded along with the active file [Note: IXVH is quite different from the IIQFOXGH prepreocessor command of standard C environments. IIQFOXGH is not implemented for reasons given later in the section describing the IC-preprocessor.]

If a downloaded program does not do what is intended, it may corrupt the p-code interpreter, particularly if pointers are being employed. The interface provides an option under the **Settings** button for downloading the firmware to reinitialize the board.

When there is a connection to a board and the downloaded programs include "main", then "main" can be executed using the **Run Main** button. The **Stop** button will halt execution of the attached system.

Under the **Tools** button, among other options, are ones for listing downloaded files, global variables, and functions (including library functions).

The interface provides additional capabilities for program entry/edit, minor adjustment to the display, and for setting up the serial interface to a board. In particular, the **Check** tool will perform a syntax check of the C program currently in the window. If there is an error, the approximate line number where the error is located is reported (the error is on the line or a nearby prior line). The edit button provides a **Go to line** option.

C programs are automatically formatted and indented. Keywords, library functions, comments, and text strings are high-lighted with color unless this feature is turned off.

IC does parenthesis-balance-highlighting when the cursor is placed to the right of any right parenthesis, bracket, or brace.

The PDLQII Function Index

After functions have been downloaded to a board, they can be invoked from IC so long as the board is connected. If one of the functions is named PDLQII, it can be run directly from the interface as noted earlier, and otherwise will be run automatically when the board is reset.

Note: to reset the Handy Board without running the PDLQII function (for instance, when hooking the board back to the computer), hold down the boarentry.d's **Start** button while activating the board. The board will then reset without running PDLQII

IC versus Standard C Index

The IC programming language is based loosely on ANSI C. However, there are major differences.

Many of these differences arise from the desire to have IC be "safer" than standard C. For instance, in IC, array bounds are checked at run time; for this reason, arrays cannot be converted to pointers in IC. Also, in IC, pointer arithmetic is not allowed.

Other differences are due to the desire that the IC runtime be small and efficient. For instance, the IC SULQW function does not understand many of the more exotic formatting options specified by ANSI C.

Yet other differences are due to the desire that IC be simpler than standard C. This is the reason for the global scope of all declarations.

In the rest of this document, when we refer to "C", the statement applies to both IC and standard C. When we wish to specify one or the other, we will refer to either "IC" or "standard C". When no such qualifiers are present, you should assume that we are talking about IC.

A Quick C Tutorial Index

Most C programs consist of function definitions and data structures. Here is a simple C program that defines a single function, called **main**.

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

The expression

[illegible] *text* ! [illegible]

forms a *multi-line* or *bracketed comment*. In contrast, text that starts with "[illegible]" forms a *single line comment*, which continues only to the end of the line. Comments are ignored by IC when the program is compiled.

All functions must have a return type. Since **main** does not return a value, it uses YRLG, the null type, as its return type. Other types include integers (LQW) and floating point numbers (IORDW). This *function declaration* information must precede each function definition.

Immediately following the function declaration is the function's name (in this case, **main**). Next, in parentheses, are any arguments (or inputs) to the function. **main** has none, but an empty set of parentheses is still required.

After the function arguments is an open curly-brace {. This signifies the start of the actual function code. Curly-braces signify program *blocks*, or chunks of code.

Next comes a series of *C statements*. Statements demand that some action be taken. Our demonstration program has a single statement, a SULQWI (formatted print). This will print the message **"Hello, world!"** to the LCD display. The **\n** indicates end-of-line. The SULQWI statement ends with a semicolon (;). *All* C statements must be ended by a semicolon. Beginning C programmers commonly make the error of omitting the semicolon that is required to end each statement.

The **main** function is ended by the close curly-brace }.

Let's look at an another example to learn some more features of C. The following code defines the function *square*, which returns the mathematical square of a number.

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

The function is declared as type LQW which means that it will return an integer value.

Next comes the function named **square**, followed by its argument list in parentheses. square has one argument, **n**, which is an integer. Notice how declaring the type of the argument is done similarly to declaring the type of the function.

When a function has arguments declared, those argument variables are valid within the "scope" of the function (i.e., they only have meaning within the function's own code). Other functions may use the same variable names independently.

The code for **square** is contained within the set of curly braces. In fact, it consists of a single statement: the UHWXUQ statement. The UHWXUQ statement exits the function and returns the value of the C *expression* that follows it (in this case "**n * n**").

Except where grouped by parentheses, expressions are evaluated according to a set of precedence rules associated with the various operations within the expression. In this case, there is only one operation (multiplication), signified by the "*", so precedence is not an issue.

Let's look at an example of a function that performs a function call to the square program.

```
float hypotenuse(int a, int b)
   {
       float h;
       h = sqrt((float)(square(a) + square(b)));
       return(h);
   }
```

This code demonstrates several more features of C. First, notice that the floating point variable **h** is defined at the beginning of the **hypotenuse** function. In general, whenever a new program block (indicated by a set of curly braces) is begun, new local variables may be defined.

The value of **h** is set to the result of a call to the VTUWfunction. It turns out that VTUWis a built-in IC function that takes a floating point number as its argument.

We want to use the **square** function we defined earlier, which returns its result as an integer. But the sqrt function requires a floating point argument. We get around this type incompatibility by coercing the integer sum **(square(a) + square(b))** into a float by preceding it with the desired type, in parentheses. Thus, the integer sum is made into a floating point number and passed along to VTUW

The **hypotenuse** function finishes by returning the value of **h**.

This concludes the brief C tutorial.

Data Objects Index

Variables and constants are the basic data objects in a C program. Declarations list the variables to be used, state what type they are, and may set their initial value.

Variables Index

Variable names are case-sensitive. The underscore character is allowed and is often used to enhance the readability of long variable names. C keywords like LI, ZKLOH, etc. may not be used as variable names.

Functions and global variables may not have the same name. In addition, if a local variable is named the same as a function or a global variable, the local use takes precedence; ie., use of the function or global variable is prevented within the scope of the local variable.

Declaration Index

In C, variables can be declared at the top level (outside of any curly braces) or at the start of each block (a functional unit of code surrounded by curly braces). In general, a variable declaration is of the form:

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

In IC, <*type*> can be LQW, ORQJ, IORDW, FKDU or VWUXFW<*struct-name*>, and determines the *primary type* of the variable declared. This form changes somewhat when dealing with pointer and array declarations, which are explained in a later section, but in general this is the way you declare variables.

Local and Global Scopes Index

If a variable is declared within a function, or as an argument to a function, its binding is *local*, meaning that the variable has existence only within that function definition. If a variable is declared outside of a function, it is a *global* variable. It is defined for all functions, including functions which are defined in files other than the one in which the global variable was declared.

Variable Initialization Index

Local and global variables can be initialized to a value when they are declared. If no initialization value is given, the variable is initialized to zero.

All global variable declarations must be initialized to constant values. 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.

Here is a small example of how initialized declarations are used.

```
int i=50;      /* declare i as global integer; initial value 50 */
long j=100L;   /* declare j as global long; initial value 100 */
int foo()
{
  int x;       /* declare x as local integer; initial value 0 */
  long y=j;    /* declare y as local integer; initial value j */
}
```

Local variables are initialized whenever the function containing them is executed. Global variables are initialized whenever a reset condition occurs. Reset conditions occur when:

1. Code is downloaded;
2. The **main()** procedure is run;
3. System hardware reset occurs.

Persistent Global Variables Index

A special persistent form of global variable, has been implemented for IC. A persistent global variable may be initialized just like any other global variable, but its value is only initialized when the code is downloaded and not on any other reset conditions. If no initialization information is included for a persistent variable, its value will be initialized to zero on download, but left unchanged on all other reset conditions.

To make a persistent global variable, prefix the type specifier with the keyword SHUVLVWHQW For example, the statement

```
persistent int i=500;
```

creates a global integer called **i** with the initial value **500**.

Persistent variables keep their state when the board is turned off and on, when **main** is run, and when system reset occurs. Persistent variables will lose their state when code is downloaded as a result of loading or unloading a file. However, it is possible to read the values of your persistent variables in IC if you are still running the same IC session from which the code was downloaded. In this manner you could read the final values of calibration persistent variables, for example, and modify the initial values given to those persistent variables appropriately.

Persistent variables were created with two applications in mind:

- Calibration and configuration values that do not need to be re-calculated on every reset condition.
- Robot learning algorithms that might occur over a period when the robot is turned on and off.

Constants Index

Integer Constants Index

Integers constants may be defined in decimal integer format (e.g., **4053** or **-1**), hexadecimal format using the "**0x**" prefix (e.g., **0x1fff**), and a non-standard but useful binary format using the "**0b**" prefix (e.g., **0b1001001**). Octal constants using the zero prefix are not supported.

Long Integer Constants Index

Long integer constants are created by appending the suffix "**l**" or "**L**" (upper- or lower- case alphabetic L) to a decimal integer. For example, **0L** is the long zero. Either the upper or lower-case "**L**" may be used, but upper-case is the convention for readability.

Floating Point Constants Index

Floating point numbers may use exponential notation (e.g., "**10e3**" or "**10E3**") or may contain a decimal period. For example, the floating point zero can be given as "**0.**", "**0.0**", or "**0E1**", but not as just "**0**". *Since the board has no floating point hardware, floating point operations are much slower than integer operations, and should be used sparingly.*

Characters and String Constants Index

Quoted characters return their ASCII value (e.g., '**x**').

Character string constants are defined with quotation marks, e.g., I7KLV LV D FKDUDFWHU VWULQJ I I

18// Index

The special constant **NULL** has the value of zero and can be assigned to and compared to pointer or array variables (which will be described in later sections). In general, you cannot convert other constants to be of a pointer type, so there are many times when **NULL** can be useful.

For example, in order to check if a pointer has been initialized you could compare its value to **NULL** and not try to access its contents if it was **NULL**. Also, if you had a defined a linked list type consisting of a value and a pointer to the next element, you could look for the end of the list by comparing the next pointer to **NULL**.

Data Types Index

IC supports the following data types:

16-bit Integers Index

16-bit integers are signified by the type indicator LQW They are signed integers, and may be valued from -32,768 to +32,767 decimal.

32-bit Integers Index

32-bit integers are signified by the type indicator ORQJ. They are signed integers, and may be valued from -2,147,483,648 to +2,147,483,647 decimal.

32-bit Floating Point Numbers Index

Floating point numbers are signified by the type indicator IRDW They have approximately seven decimal digits of precision and are valued from about 10^-38 to 10^38.

8-bit Characters Index

Characters are an 8-bit number signified by the type indicator FKDU A character's value typically represents a printable symbol using the standard ASCII character code, but this is not necessary; characters can be used to refer to arbitrary 8-bit numbers.

Pointers Index IC pointers are 16-bit numbers which represent locations in memory. Values in memory can be manipulated by calculating, passing and *dereferencing* pointers representing the location where the information is stored.

Arrays Index Arrays are used to store homogenous lists of data (meaning that all the elements of an array have the same type). Every array has a length which is determined at the time the array is declared. The data stored in the elements of an array can be set and retrieved in the same manner as for other variables.

Structures Index

Structures are used to store non-homogenous but related sets of data. Elements of a structure are referenced by name instead of number and may be of any supported type.

Structures are useful for organizing related data into a coherent format, reducing the number of arguments passed to functions, increasing the effective number of values which can be returned by functions, and creating complex data representations such as directed graphs and linked lists.

Pointers Index

The address where a value is stored in memory is known as the pointer to that value. 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.

Attempts to refer to invalid memory locations could corrupt your memory. Most computing environments that you are probably used to return helpful messages like 'Segmentation Violation' or 'Bus Error' on attempts to access illegal memory. However, you won't have this safety net on the board you are connecting to. Invalid pointer dereferencing is very likely to go undetected, and will likely render invalid your data, your program, or even the pcode interpreter.

Pointer Safety Index

In past versions of IC, you could not return pointers from functions or have arrays of pointers. In order to facilitate the use of structures, these features have been added to the current version. With this change, the number of opportunities to misuse pointers have increased. However, if you follow a few simple precautions you should do fine.

First, you should always check that the value of a pointer is not equal to **NULL** (a special zero pointer) before you try to access it. Variables which are declared to be pointers are initialized to **NULL**, so many uninitialized values could be caught this way.

Second, you should never use a pointer to a local variable in a manner which could cause it to be accessed after the function in which it was declared terminates. When a function terminates the space where its values were being stored is recycled. Therefore not only may dereferencing such pointers return incorrect values, but assigning to those addresses could lead to serious data corruption. A good way to prevent this is to never return the address of a local variable from the function which declares it and never store those pointers in an object which will live longer than the function itself (a global pointer, array, or VWUXFW). Global variables and variables local to main will not move once declared and their pointers can be considered to be secure.

The type checking done by IC will help prevent many mishaps, but it will not catch all errors, so be careful.

Pointer Declaration and Use Index

A variable which is a pointer to an object of a given type is declared in the same manner as a regular object of that type, but with an extra * in front of the variable name.

The value stored at the location the pointer refers to is accessed by using the * operator before the expression which calculates the pointer. This process is known as dereferencing.

The address of a variable is calculated by using the **&** operator before that variable, array element, or structure element reference.

There are two main differences between how you would use a variable of a given type and a variable declared as a pointer to that type.

For the following explanation, consider **X** and **Xptr** as defined as follows:

```
long X; long *Xptr;
```

- Space Allocation -- Declaring an object of a given type, as **X** is of type ORQJ, allocates the space needed to store that value. Because an IC long takes four bytes of memory, four bytes are reserved for the value of **X** to occupy. However, a pointer like **Xptr** does not have the same amount of space allocated for it that is needed for an object of the type it points to. Therefore it can only safely refer to space which has already been allocated for globals (in a special section of memory reserved for globals) or locals (temporary storage on the stack).
- Initial Value -- It is always safe to refer to a non-pointer type, even if it hasn't been initialized. However pointers have to be specifically assigned to the address of legally allocated space or to the value of an already initialized pointer before they are safe to use.

So, for example, consider what would happen if the first two statements after **X** and **Xptr** were declared were the following:

```
X=50L; *Xptr=50L;
```

The first statement is valid: it sets the value of **X** to **50L**. The second statement would be valid if **Xptr** had been properly initialized, but in this case it has not. Therefore, this statement would corrupt memory.

Here is a sequence of commands you could try which illustrate how pointers and the * and & operators are used. It also shows that once a pointer has been set to point at a place in memory, references to it actually share the same memory as the object it points to:

```
X=50L;                /* set the memory allocated for X to 50 */
Xptr=&X;              /* set Xptr to point to memory address of X */
printf("%d ",*Xptr);  /* dereference Xptr; value at address is 50 */
X=100L;               /* set X to the value 100 */
printf("%d ",*Xptr);  /* dereference again; value is now 100 */
*Xptr=200L;           /* set value at address given by Xptr to 200 */
printf("%d\n",X);     /* check that the value of X changed to 200 */
```

Passing Pointers as Arguments Index

Pointers can be passed to functions and functions can change the values of the variables that are pointed at. This is termed *call-by-reference*; a reference, or pointer, to a variable is given to the function that is being called. This is in contrast to *call-by-value*, the standard way that functions are called, in which the value of a variable is given the to function being called.

The following example defines an **average_sensor** function which takes a port number and a pointer to an integer variable. The function will average the sensor and store the result in the variable pointed at by **result**.

Prefixing an argument name with * declares that the argument is a pointer.

```
void average_sensor(int port, int *result)
{
   int sum = 0;
   int i;
   for (I = 0; I < 10; i++) sum += analog(port);
   *result =  sum/10;
}
```

Notice that the function itself is declared as a YRIG. It does not need to return anything, because it instead stores its answer in the memory location given by the pointer variable that is passed to it.

The pointer variable is used in the last line of the function. In this statement, the answer sum/10 is stored at the location pointed at by result. Notice that the * is used to assign a value to the location pointed by result.

Returning Pointers from Functions Index

Pointers can also be returned from functions. Functions are defined to return pointers by preceeding the name of the function with a star, just like any other type of pointer declaration.

```
int right,left;
int *dirptr(int dir)
{
    if (dir==0) {
      return(&right);
    }
    if (dir==1) {
       return(&left);
    }
    return(NULL);
}
```

The function **dirptr** returns a pointer to the global **right** when its argument **dir** is 0, a pointer to **left** when its argument is 1, and **NULL"** if its argument is other than 0 or 1.

Arrays Index

IC supports arrays of characters, integers, long integers, floating-point numbers, structures, pointers, and array pointers (multi-dimensional arrays). While unlike regular C arrays in a number of respects, they can be used in a similar manner. The main reasons that arrays are useful are that they allow you to allocate space for many instances of a given type, send an arbitrary number of values to functions, and provide the means for iterating over a set of values.

Arrays in IC are different and incompatible with arrays in other versions of C. This incompatibility is caused by the fact that references to IC arrays are checked to insure that the reference is truly within the bounds of that array. In order to accomplish this checking in the general case, it is necessary that the size of the array be stored with the contents of the array. *It is important to remember that an array of a given type and a pointer to the same type are incompatible types in IC, whereas they are largely interchangeable in regular C.*

Declaring and Initializing Arrays Index

Arrays are declared using square brackets. The following statement declares an array of ten integers:

```
int foo[10];
```

In this array, elements are numbered from 0 to 9. Elements are accessed by enclosing the index number within square brackets: **foo[4]** denotes the fifth element of the array foo (since counting begins at zero).

Arrays are initialized by default to contain all zero values. Arrays may also be initialized at declaration by specifying the array elements, separated by commas, within curly braces. If no size value is specified within the square brackets when the array is declared but initialization information is given, the size of the array is determined by the number of elements given in the declaration. For example,

```
int foo[]= {0, 4, 5, -8,  17, 301};
```

creates an array of six integers, with **foo[0]** equaling **0**, **foo[1]** equaling **4**, etc.

If a size is specified and initialization data is given, the length of the initialization data may not exceed the specified length of the array or an error results. If, on the other hand, you specify the size and provide fewer initialization elements than the total length of the array, the remaining elements are initialized to zero.

Character arrays are typically text strings. There is a special syntax for initializing arrays of characters. The character values of the array are enclosed in quotation marks:

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

This form creates a character array called **string** with the ASCII values of the specified characters. In addition, the character array is terminated by a zero. Because of this zero-termination, the character array can be treated as a string for purposes of printing (for example). Character arrays can be initialized using the curly braces syntax, but they will not be automatically null-terminated in that case. In general, printing of character arrays that are not null-terminated will cause problems.

Passing Arrays as Arguments Index

When an array is passed to a function as an argument, the array's pointer is actually passed, rather than the elements of the array. If the function modifies the array values, the array will be modified, since there is only one copy of the array in memory.

In normal C, there are two ways of declaring an array argument: as an array or as a pointer to the type of the array's elements. In IC array pointers are incompatible with pointers to the elements of an array so such arguments can only be declared as arrays.

As an example, the following function takes an index and an array, and returns the array element specified by the index:

```
int retrieve_element(int index, int array[])
{
    return array[index];
}
```

Notice the use of the square brackets to declare the argument array as a pointer to an array of integers.

When passing an array variable to a function, you are actually passing the value of the array pointer itself and not one of its elements, so no square brackets are used.

```
void foo()
{
    int array[10];
    retrieve_element(3, array);
}
```

Multi-dimensional Arrays Index

A two-dimensional array is just like a single dimensional array whose elements are one- dimensional arrays. Declaration of a two-dimensional array is as follows:

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

The number in the first set of brackets is the number of 1-D arrays of LQW The number in the second set of brackets is the length of each of the 1-D arrays of LQW In this example, **k** is an array containing two 1-D arrays; **k[0]** is a 1-D array of FRORU EXKH! LQW of length 3; **k[0][1]** is an FRORU EXKH! LQW Arrays of with any number of dimensions can be generalized from this example by adding more brackets in the declaration.

Determining the size of Arrays at Runtime Index

An advantage of the way IC deals with arrays is that you can determine the size of arrays at runtime. This allows you to do size checking on an array if you are uncertain of its dimensions and possibly prevent your program from crashing.

Since BDUUD\BVLJH *is not a standard C feature, code written using this primitive will only be able to be compiled with IC.*

The **_array_size** primitive returns the size of the array given to it regardless of the dimension or type of the array. Here is an example of declarations and interaction with the **_array_size** primitive:

```
int i[4]={10,20,30};
int j[3][2]={{1,2},{2,4},{15}};
int k[2][2][2];
_array_size(i);    /* returns 4 */
_array_size(j);    /* returns 3 */
_array_size(j[0]); /* returns 2 */
_array_size(k);    /* returns 2 */
_array_size(k[0]); /* returns 2 */
```

Uploading Arrays Index

When an executing program is paused or has finished, IC can upload the values stored in any global array via the serial port. This permits collecting and recording data for purposes such as experimentation or calibration.

The IC upload array capability is accessed using the tools tab. When upload array is activated, it lists all globally declared arrays. When an array is selected, it is opened in a (modal) view window. The array can be copied to the clipboard, or saved to a TXT or CSV (comma separated values) file for import to analysis software.

Structures Index

Structures are used to store non-homogenous but related sets of data. Elements of a structure are referenced by name instead of number and may be of any supported type. Structures are useful for organizing related data into a coherent format, reducing the number of arguments passed to functions, increasing the effective number of values which can be returned by functions, and creating complex data representations such as directed graphs and linked lists.

The following example shows how to define a structure, declare a variable of structure type, and access its elements.

```
struct foo
{
    int i;
    int j;
};
struct foo f1;
void set_f1(int i,int j)
{
    f1.i=i;
    f1.j=j;
}
void get_f1(int *i,int *j)
{
    *i=f1.i;
    *j=f1.j;
}
```

The first part is the structure definition. It consists of the keyword VWUXFW followed by the name of the structure (which can be any valid identifier), followed by a list of named elements in curly braces. This definition specifies the structure of the type VWUXFW**foo**. Once there is a definition of this form, you can use the type VWUXFW**foo** just like any other type. The line

```
 struct foo f1;
```

is a global variable declaration which declares the variable **f1** to be of type VWUXFW**foo**.

The dot operator is used to access the elements of a variable of structure type. In this case, **f1.i** and **f1.j** refer to the two elements of **f1**. You can treat the quantities **f1.i** and **f1.j** just as you would treat any variables of type LQW(the type of the elements was defined in the structure declaration at the top to be LQW).

Pointers to structure types can also be used, just like pointers to any other type. However, with structures, there is a special short-cut for referring to the elements of the structure pointed to.

```
struct foo *fptr;
void main()
{
    fptr=&f1;
    fptr->i=10;
    fptr->j=20;
}
```

In this example, **fptr** is declared to be a pointer to type VWUXFW**foo**. In main, it is set to point to the global **f1** defined above. Then the elements of the structure pointed to by **fptr** (in this case these are the same as the elements of **f1**), are set. The arrow operator is used instead of the dot operator because fptr is a pointer to a variable of type VWUXFW**foo**. Note that **(*fptr).i** would have worked just as well as **fptr▮! i**, but it would have been clumsier.

Note that only pointers to structures, not the structures themselves, can be passed to or returned from functions.

Complex Initialization examples Index

Complex types -- arrays and structures -- may be initialized upon declaration with a sequence of constant values contained within curly braces and separated by commas.

Arrays of character may also be initialized with a quoted string of characters.

For initialized declarations of single dimensional arrays, the length can be left blank and a suitable length based on the initialization data will be assigned to it. *Multi-dimensional arrays must have the size of all dimensions specified when the array is declared*. If a length is specified, the initialization data may not overflow that length in any dimension or an error will result. However, the initialization data may be shorter than the specified size and the remaining entries will be initialized to 0.

Following is an example of legal global and local variable initializations:

```
/* declare many globals of various types */
int i=50;
int *ptr=NULL;
float farr[3]={ 1.2, 3.6, 7.4 };
int tarr[2][4]={ { 1, 2, 3, 4 }, { 2, 4, 6, 8} };
char c[]="Hi there how are you?";
char carr[5][10]={"Hi","there","how","are","you"};
struct bar
{
   int i;
   int *p;
   long j;
} b={5, NULL, 10L};
struct bar barr[2] = { { 1, NULL, 2L }, { 3 } };
/* declare locals of various types */
int foo()
{
   int x;             /* local variable x with initial value 0 */
   int y= tarr[0][2]; /* local variable y with initial value 3 */
   int *iptr=&i;      /* local pointer to integer
                         which points to the global i */
   int larr[2]={10,20};  /* local array larr
                            with elements 10 and 20 */
   struct bar lb={5,NULL,10L};  /* local variable of type
                                   struct bar with i=5 and j=10 */
   char lc[]=carr[2];    /* local string lc with
                            initial value "how" */
   ...
}
```

Statements and Expressions Index

Operators act upon objects of a certain type or types and specify what is to be done to them. Expressions combine variables and constants to create new values. Statements are expressions, assignments, function calls, or control flow statements which make up C programs.

Operators Index

Each of the data types has its own set of operators that determine which operations may be performed on them.

Integer Operations Index

The following operations are supported on integers:

- **Arithmetic**. addition I, subtraction I, multiplication I, division I.
- **Comparison**. greater-than !, less-than I, equality , greater-than-equal ! , less-than-equal I .
- **Bitwise Arithmetic**. bitwise-OR _, bitwise-AND I, bitwise-exclusive-OR A, bitwise-NOT a.
- **Boolean Arithmetic**. logical-OR _, logical-AND I I, logical-NOT I.
  When a C statement uses a boolean value (for example, II), it takes the integer zero as meaning false, and any integer other than zero as meaning true. The boolean operators return zero for false and one for true. Boolean operators I I and _will stop executing as soon as the truth of the final expression is determined. For example, in the expression DII I E, if D is false, then E does not need to be evaluated because the result must be false. The I I operator therefore will not evaluate E.

Long Integers Index

A subset of the operations implemented for integers are implemented for long integers: arithmetic addition +, subtraction -, and multiplication *, and the integer comparison operations. Bitwise and boolean operations and division are not supported.

Floating Point Numbers Index

IC uses a package of public-domain floating point routines distributed by Motorola. This package includes arithmetic, trigonometric, and logarithmic functions. Since floating point operations are implemented in software, they are much slower than the integer operations; we recommend against using floating point if you are concerned about performance.

The following operations are supported on floating point numbers:

- **Arithmetic**. addition [illegible], subtraction [illegible], multiplication [illegible], division [illegible].
- **Comparison**. greater-than [illegible], less-than [illegible], equality [illegible], greater-than-equal [illegible], less-than-equal [illegible].
- **Built-in Math Functions**. A set of trigonometric, logarithmic, and exponential functions is supported. For details, go to the Library Function Descriptions. These functions are included among those itemized as "Math" functions.

Characters Index

Characters are only allowed in character arrays. When a cell of the array is referenced, it is automatically coerced into a integer representation for manipulation by the integer operations. When a value is stored into a character array, it is coerced from a standard 16- bit integer into an 8-bit character (by truncating the upper eight bits).

Assignment Operators and Expressions Index

The basic assignment operator is [illegible]. The following statement adds 2 to the value of [illegible].

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

The abbreviated form

```
    a += 2;
```

could also be used to perform the same operation. All of the following binary operators can be used in this fashion:

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

Increment and Decrement Operators Index

The increment operator "[illegible]" increments the named variable. For example, the construction "[illegible]" is equivalent to "[illegible]" or "[illegible]". A statement that uses an increment operator has a value. For example, the statement

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

will display the text "[illegible]". If the increment operator comes after the named variable, then the value of the statement is calculated after the increment occurs. So the statement

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

would display "[illegible]" but would finish with a set to 4. The decrement operator "[illegible]" is used in the same fashion as the increment operator.

Data Access Operators Index

[illegible]

A single ampersand preceding a variable, an array reference, or a structure element reference returns a pointer to the location in memory where that information is being stored. This should not be used on arbitrary expressions as they do not have a stable place in memory where they are being stored.

[illegible]

A single * preceeding an expression which evaluates to a pointer returns the value which is stored at that address. This process of accessing the value stored within a pointer is known as *dereferencing*.

[illegible] *expr* [illegible]

An expression in square braces following an expression which evaluates to an array (an array variable, the result of a function which returns an array pointer, etc.) checks that the value of the expression falls within the bounds of the array and references that element.

[illegible]

A dot between a structure variable and the name of one of its fields returns the value stored in that field.

[illegible]

An arrow between a pointer to a structure and the name of one of its fields in that structure acts the same as a dot does, except it acts on the structure pointed at by its left hand side. Where **f** is a structure of a type with **i** as an element name, the two expressions [illegible] and [illegible] are equivalent.

Precedence and Order of Evaluation Index

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.

Operator	Associativity
[illegible]	left to right
[illegible] *type* [illegible]	right to left
[illegible]	left to right
[illegible]	left to right
[illegible]	left to right
[illegible]	left to right
[illegible]	left to right
[illegible]	left to right
[illegible]	left to right
[illegible]	left to right
[illegible]	left to right
[illegible]	right to left
[illegible] etc.	right to left
[illegible]	left to right

Control Flow Index

IC supports most of the standard C control structures.

Statements and Blocks Index

A single C statement is ended by a semicolon. A series of statements may be grouped together into a *block* using curly braces. Inside a block, local variables may be defined. Blocks may be used in place of statements in the control flow constructs.

If-Else Index

The [illegible] statement is used to make decisions. The syntax is:

```
    if (<expression>)
        <statement-1>
    else
        <statement-2>
```

<*expression*> is evaluated; if it is not equal to zero (e.g., logic true), then <*statement-1*> is executed.

The HOVH clause is optional. If the LI part of the statement did not execute, and the HOVH is present, then <*statement-2*> executes.

While Index

The syntax of a ZKLOH loop is the following:

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

ZKLOH begins by evaluating <*expression*>. If it is false, then <*statement*> is skipped. If it is true, then <*statement*> is evaluated. Then the expression is evaluated again, and the same check is performed. The loop exits when <*expression*> becomes zero.

One can easily create an infinite loop in C using the ZKLOH statement:

```
while (1)
    <statement>
```

Do-While Index

The syntax of a GR-ZKLOH loop is the following:

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

The equivalent while loop would be the following:

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

For Index

The syntax of a IRU loop is the following:

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

The IRU construct is equivalent to the following construct using ZKLOH:

```
<expr-1>;
ZKLOH (<expr-2>)
{
    <statement>
    <expr-3>;
}
```

Typically, <*expr-1*> is an assignment, <*expr-2*> is a relational expression, and <*expr-3*> is an increment or decrement of some manner. For example, the following code counts from 0 to 99, printing each number along the way:

```
int i;
for (i = 0; i < 100; i++)
     printf("%d\n", i);
```

Switch Index

The syntax of a VZLWFK block is as follows:

```
VZLWFK (LQW)
{
     FDVH const1:
          <statement list1>
          EUHDN;
     FDVH const2:
          <statement list2>
          EUHDN;
     GHIDXOW:
          <statement list3>
}
```

The VZLWFK construct takes an integer variable as input, and compares it to each FDVH listed. The first matching *const* is selected, and execution begins there. The EUHDN is optional, and if no break is found then execution continues through each following statement. Also note that each FDVH has a list of single statements, as opposed to a block enclosed in curly braces.

Here's an example of how a VZLWFK might be used:

```
LQW i = 1;
VZLWFK(i)
{

     FDVH 0:
          SULQWI([illegible]);
          EUHDN;
     FDVH 1:
          SULQWI([illegible]);
          EUHDN;
     GHIDXOW:
          SULQWI([illegible]);
}
```

Since *i* is equal to 1, the text "Case 1" will be printed to the screen. If *i* were equal to 0, "Case 0" would be printed. If *i* were any number besides 0 or 1, "Default" would be printed.

Break Index

Use of the EUHDN statement provides an early exit from a ZKLOH, GR-ZKLOH or IRU loop. The break statement can also provide an exit from a VZLWFK block.

LCD Screen Printing Index

IC has a version of the C function SULQWI for formatted printing to the LCD screen.

The syntax of SULQWI is the following:

```
printf(<format-string>, <arg-1> , ... , <arg-N>);
```

This is best illustrated by some examples.

Printing Examples Index

Example 1: Printing a message PM Index

The following statement prints a text string to the screen.

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

In this example, the format string is simply printed to the screen. The character ?Q at the end of the string signifies end-of-line. When an end-of-line character is printed, the LCD screen will be cleared when a subsequent character is printed. Thus, most SULQWI statements are terminated by a ?Q.

Example 2: Printing a number [Index]

The following statement prints the value of the integer variable x with a brief message.

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

The special form G is used to format the printing of an integer in decimal format.

Example 3: Printing a number in binary [Index]

The following statement prints the value of the integer variable x as a binary number.

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

The special form E is used to format the printing of an integer in binary format. Only the low byte of the number is printed.

Example 4: Printing a floating point number [Index]

The following statement prints the value of the floating point variable Q as a floating point number.

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

The special form I is used to format the printing of floating point number.

Example 5: Printing two numbers in hexadecimal format [Index]

```
    printf("A=%x  B=%x\n", a, b);
```

The form [formats an integer to print in hexadecimal.

Formatting Command Summary [Index]

Format Command	Data Type	Description
G	LQW	decimal number
[	LQW	hexadecimal number
E	LQW	low byte as binary number
F	LQW	low byte as ASCII character
I	IORDW	floating point number
V	FKDU array	char array (string)

Special Notes

- Depending on the display being used, the final character position of the LCD screen may be used as a system "heartbeat." This character continuously blinks between a large and small heart when the board is operating properly. If the character stops blinking, the board has failed.
- When using a two-line LCD display, the SULQWI command treats the display as a single longer line.
- For 1 or 2 line displays any excess text is truncated.
- Printing of long integers is not presently supported.

Preprocessor Index

The preprocessor processes a file before it is sent to the compiler. The IC preprocessor allows definition of macros, and conditional compilation of sections of code. Using preprocessor macros for constants and function macros can make IC code more efficient as well as easier to read. Using ▌II to conditionally compile code can be very useful, for instance, for debugging purposes.

The special preprocessor command ▌XVH has been included to allow programs to cause a program to download to initiate the download of stored programs that are not in the IC library. For example, suppose you have a set of stored programs in a file named "P\OLE▌F", some of which you need for your current program to work.

```
/* load my library */
#use "mylib.ic"

void main()
{
    char s[32] = "text string wrapping badly\n";
    fix (s);    /* apply my fix function to s and print it */
    printf(s);
}
```

Preprocessor Macros Index

Preprocessor macros are defined by using the ▌GHILQH preprocessor directive at the start of a line. A macro is local to the file in which it is defined. The following example shows how to define preprocessor macros.

```
#define RIGHT_MOTOR 0
#define LEFT_MOTOR  1
#define GO_RIGHT(power)  (motor(RIGHT_MOTOR,(power)))
#define GO_LEFT(power)   (motor(LEFT_MOTOR,(power)))
#define GO(left,right) {GO_LEFT(left); GO_RIGHT(right);}
void main()
{
    GO(0,0);
}
```

Preprocessor macro definitions start with the ▌GHILQH directive at the start of a line, and continue to the end of the line. After ▌GHILQH is the name of the macro, such as **RIGHT_MOTOR**. If there is a parenthesis directly after the name of the macro, such as the **GO_RIGHT** macro has above, then the macro has arguments. The **GO_RIGHT** and **GO_LEFT** macros each take one argument. The GO macro takes two arguments. After the name and the optional argument list is the body of the macro.

Each time a macro is invoked, it is replaced with its body. If the macro has arguments, then each place the argument appears in the body is replaced with the actual argument provided.

Invocations of macros without arguments look like global variable references. Invocations of macros with arguments look like calls to functions. To an extent, this is how they act. However, macro replacement happens before compilation, whereas global references and function calls happen at run time. Also, function calls evaluate their arguments before they are called, whereas macros simply perform text replacement. For example, if the actual argument given to a macro contains a function call, and the macro instantiates its argument more than once in its body, then the function would be called multiple times, whereas it would only be called once if it were being passed as a function argument instead.

Appropriate use of macros can make IC programs and easier to read. It allows constants to be given symbolic names without requiring storage and access time as a global would. It also allows macros with arguments to be used in cases when a function call is desirable for abstraction, without the performance penalty of calling a function.

Conditional compilation Index

It is sometimes desirable to conditionally compile code. The primary example of this is that you may want to perform debugging output sometimes, and disable it at other times. The IC preprocessor provides a convenient way of doing this by using the `#ifdef` directive.

```
void go_left(int power)
{
    GO_LEFT(power);
#ifdef DEBUG
    printf("Going Left\n");
    beep();
#endif
}
```

In this example, when the macro **DEBUG** is defined, the debugging message "Going Left" will be printed and the board will beep each time **go_left** is called. If the macro is not defined, the message and beep will not happen. Each `#ifdef` must be follwed by an `#endif` at the end of the code which is being conditionally compiled. The macro to be checked can be anything, and `#ifdef` blocks may be nested.

Unlike regular C preprocessors, macros cannot be conditionally defined. If a macro definition occurs inside an `#ifdef` block, it will be defined regardless of whether the `#ifdef` evaluates to true or false. The compiler will generate a warning if macro definitions occur within an `#ifdef` block.

The `#if`, `#else`, and `#elif` directives are also available, but are outside the scope of this document. Refer to a C reference manual for how to use them.

Comparison with regular C preprocessors Index

The way in which IC deals with loading multiple files is fundamentally different from the way in which it is done in standard C. In particular, when using standard C, files are compiled completely independently of each other, then linked together. In IC, on the other hand, all files are compiled together. This is why standard C needs function prototypes and **extern** global definitions in order for multiple files to share functions and globals, while IC does not.

In a standard C preprocessor, preprocessor macros defined in one C file cannot be used in another C file unless defined again. Also, the scope of macros is only from the point of definition to the end of the file. The solution then is to have the prototypes, **extern** declarations, and macros in header files which are then included at the top of each C file using the `#include` directive. This style interacts well with the fact that each file is compiled independent of all the others.

However, since declarations in IC do not file scope, it would be inconsistent to have a preprocessor with file scope. Therefore, for consistency it was desirable to give IC macros the same behavior as globals and functions. Therefore, preprocessor macros have global scope. If a macro is defined anywhere in the files loaded into IC, it is defined everywhere. Therefore, the `#include` and `#undef` directives did not seem to have any appropriate purpose, and were accordingly left out.

The fact that `#define` directives contained within `#if` blocks are defined regardless of whether the `#if` evaluates to be true or false is a side effect of making the preprocessor macros have global scope.

Other than these modifications, the IC preprocessor should be compatible with regular C preprocessors.

The IC Library File Index

Library files provide standard C functions for interfacing with hardware on the robot controller board. These functions are written either in C or as assembly language drivers. Library files provide functions to do things like control motors, make tones, and input sensors values.

IC automatically loads the library file every time it is invoked. Depending on which board is being used, a different library file will be required. IC may be configured to load different library files as its default; IC will automatically load the correct library for the board you're using at the moment.

Separate documentation covers all library functions available for the Handy Board, the Lego RCX, and the XBC; if you have another board, see your owner's manual for documentation.

To understand better how the library functions work, study of the library file source code is recommended; e.g., the main library file for the Handy Board is named [illegible].

For convenience, a description of some of the more commonly used library functions follows.

Commonly Used IC Library Functions Index

```
digital();
  /* returns 0 if the switch attached to the port is open and
     returns 1 if the switch is closed.  Digital ports are numbered
     7-15.  Typically used for bumpers or limit switches. */

analog();
  /* returns the analog value of the port (a value in the range 0-255).
     Analog ports on the handy board are numbered 2-6 and 16-23.  Light
     sensors and range sensors are examples of sensors you would
     use in analog ports (only on Handy Board). */
sleep(<float_secs>);
  /* waits specified number of seconds */

beep();
  /* causes a beep sound */

tone(<float_frequency>, <float_secs>)
  /* plays at specified frequency for specified time (seconds) */

printf(<string>, <arg1>, <arg2>, ... );
  /* prints <string>.  If the string contains % codes then the <args>
     after the string will be printed in place of the % codes in the
     format specified by the code. %d prints a decimal number. %f
     prints a floating point number. %c prints a character, %b prints
     an integer in binary, %x prints an integer in hexadecimal. */

motor(<motor_#>, <speed>)
  /* controls the motors. <motor_#> is an integer between 0 and 3 (1
     less for RCX).  <speed> is an integer between -100 and 100 where 0
     means the motor is off and negative numbers run the motor in the
     reverse direction */

fd(<motor_#>);
  /* turns on the motor specified (direction is determined by plug
     orientation */

bk(<motor_#>);
  /* turns on the motor specified in the opposite direction from fd */

off(<motor_#>);
  /* turns off the motor specified */

ao();
  /* turns all motor ports off */
```

Processes Index

Processes work in parallel. Each process, once it is started, will continue until it finishes or until it is killed by another process using the NLOOBSURFHVV *process_id*! statement. Each process that is active gets 50ms of processing time. Then the process is paused temporarily and the next process gets its share of time. This continues until all the active process have gotten a slice of time, then it all repeats again. From the user's standpoint it appears that all the active processes are running in parallel.

Processes can communicate with one another by reading and modifying global variables. The globals can be used as semaphores so that one process can signal another. Process IDs may also be stored in globals so that one process can kill another when needed.

Up to 4 processes initiated by the VWDUWBSURFHVV library function can be active at any time.

The library functions for controlling processes are:

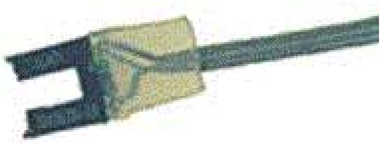
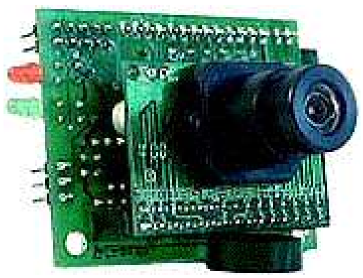
```
start_process(<function_name>(<arg1>, <arg2>, . . .));
  /* start_process returns an integer that is the <process_id>
     and starts the function <function_name> as a separate
     process */

defer();
  /* when placed in a function that is used as a process this
     will cause that process to give up the remainder of its time
     slice whenever defer is called */

kill_process(<process_id>);
  /* this will terminate the process specified by the
     <process_id> */
```

Encoders Index

The HQDEOHBHQFRGHU library function is used to start a process which updates the transition count for the encoder specified. The encoder library functions are designed for sensors connected to (digital) ports 7,8,12,13 on the Handyboard (The corresponding *encoder#*! values are 0,1,2,3), or in 1,2, or 3 on the RCX. Every enabled encoder uses a lot of the processor -- so don't enable an encoder unless you are going to use it, and *never put an enable statement inside of a loop*.

```
enable_encoder(<encoder#>);
  /* turns on the specified encoder (either 0,1,2, or 3 which are
     plugged into digital ports 7,8, 12 & 13 respectively, on the HB).
     This should be done only once - never enable an already enabled
     encoder.  If an encoder is not enabled, read_encoder will
     always return 0. */
disable_encoder(<encoder#>)
  /* turns off the specified encoder */
reset_encoder(<encoder#>)
  /* sets the specified encoder value to 0 */
read_encoder(<encoder#>)
  /* returns an int that is the current value of the specified
     encoder */
```

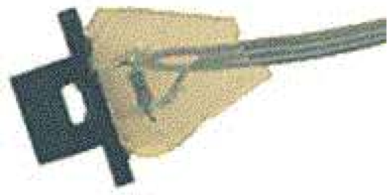
Sensors

Light Sensor (analog)	**Infrared "Top Hat" Reflectance Sensor** (analog)	**Infrared "E.T." Distance Sensor** (high-impedance analog)
Touch Sensors (digital)		 **Infrared "Break Beam" Sensor** (digital)
Sonar (Ultrasonic Rangefinder)	 **CMU Cam**	

Light Sensor (analog)

- Analog sensor
- Connect to ports 2-6 or 20-23
- Access with function DQDORJ II SRUW! I
- Low values indicate bright light
- High values indicate low light
- Sensor is somewhat directional and can be made more so using black paper or tape or an opaque straw or lego to shade extraneous light. Sensor can be attenuated by placing paper in front.

Infrared "Top Hat" Reflectance Sensor (analog)



- Analog sensor
- Connect to ports 2-6 or 20-23
- Access with function DQDORJ ll SRUW! l
- Low values indicate bright light, light color, or close proximity
- High values indicate low light, dark color, or distance of several inches
- Sensor has a reflectance range of about 3 inches

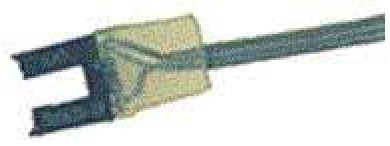
Infrared "E.T." Distance Sensor (high-impedance analog)

- Floating analog sensor
- Connect to ports 16-19
- Access with function DQDORJ ll SRUW! l
- Low values indicate large distance
- High values indicate distance approaching ~4 inches
- Range is 4-30 inches. Result is approximately 1/d2. Objects closer than 4 inches will appear to be 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 sensors
- Connect to ports 7-15
- Access with function GLJLWDO■■ *SRUW*! ■
- 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 7, 8, 12 or 13
- Access with function GLJLWDO■■ *SRUW*! ■
- 1 indicates slot is empty
- 0 indicates slot is blocked
- These can be used much like touch sensors (if the object being touched fits in the slot)
- Special abilities when used as encoders
- The handyboard can count the number of transitions that a digital sensor makes
- This is particularly useful for the slot sensor in conjunction with a wheel, which allows measuring rotational rate and total angle.
- **The four encoder channels use digital ports 7, 8, 12, and 13**:
  - Encoder channel 0 is labeled port 7 on your Handyboard (you cannot use encoder 0 if you are using the sonar)
  - Encoder channel 1 is labeled port 8
  - Encoder channel 2 is labeled port 12
  - Encoder channel 3 is labeled port 13

Encoder functions:

Remember that all encoder functions use the encoder channel number, not the digital port number which is labeled on the Handyboard. The encoder channel is a number between 0 and 3.

HQDEOHBHQFRGHU■■ *HQFRGHU*! ■■
This function enables an encoder channel. At the beginning of your program, you should call HQDEOHBHQFRGHU once for each encoder you plan to use.

GLVDEOHBHQFRGHU■■ *HQFRGHU*! ■■
This function disables an encoder channel. Normally there is no reason to call GLVDEOHBHQFRGHU

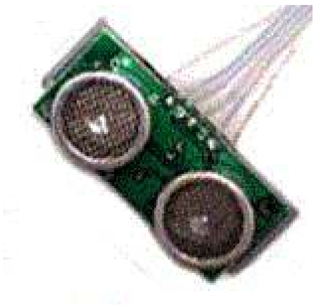
UHDGBHQFRGHU■■ *HQFRGHU*! ■■
Returns the number of transitions. Each transition counts (both 0 to 1 and 1 to 0).

UHVHWBHQFRGHU■■ *HQFRGHU*! ■■
Sets that encoder's count to zero

Encoder technical details

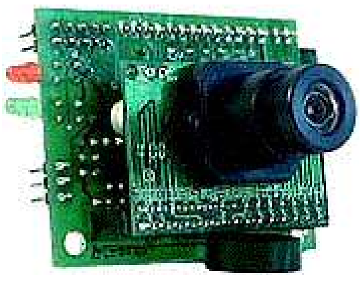
- Encoder channels 0 and 1 are "hardware" encoders, and can count very quickly. Some sensors, especially mechanical pushbuttons, have multiple transitions all at the same time (this is sometimes called "bounce"). The hardware encoders can count all the bounces.
- Encoder channels 2 and 3 are "software" encoders, and are sampled 1000 times per second. They are less sensitive to bounce than the hardware encoders. Because these channels are implemented in software, enabling them will cause your program to run slower.

Sonar (Ultrasonic Rangefinder)



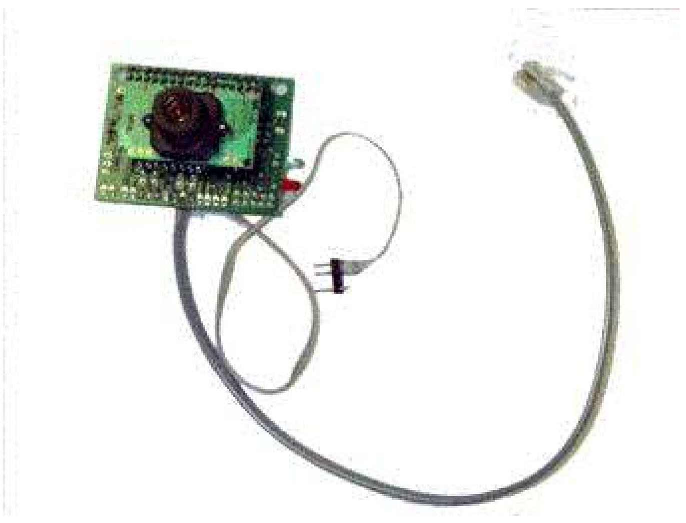
- Timed analog sensor. Sends a "ping" of high-pitched sound and listens for an echo
- Connect the **red** plug to expansion board (upper deck) port #0
- Connect **gray** plug to digital port #7
- Access with function VRQDU▮▮
- Returned value is distance in mm to closest object in field of view
- Range is approximately 30-2000mm
- If object is too close or too far, the sensor will not detect a return
- No return (because objects are too close or too far) gives value of 32767
- Wait at least .03 seconds between calls to VRQDU▮▮ to allow echos to die out (you could call VOHHS▮▮▮ ▮ ▮
- Sound frequency is 40 kilohertz (40,000 cycles per second). This is an ultrasonic frequency because most humans can only hear up 20 kilohertz.
- Speed of sound is ~300mm/ms
- sonar() times the echo, divides by two and multiplies by speed of sound
- Devantech part number SRF04
- The sonar field of view is a 30 degree teardrop:

CMUCam Vision System



The CMUCam uses a digital camera and can find colored objects.

The CMUcam and adapter dongle

 *CMUCam with power connector and serial cable*	*Adapter dongle* The dongle allows you to charge and Zap charge your HB while the camera is connected.

Attaching the CMUcam

1. Attach the CMUCam power connector to any Handyboard digital or analog port. This allows the Handyboard to power the CMUCam.
2. Attach the CMUCam serial cable to the jack marked with a red dot on the adapter dongle.
3. Use an RJ11 cable to connect the jack on the Handyboard interface to the unmarked jack on the dongle.
4. Connect the cable which is soldered to the adapter dongle to the Handyboard's serial jack.

Setting the switch

The dongle has a 3-position switch:

1. Switch by red dot (camera connector): HB is talking with the camera

2. Switch in middle: CMUcam talks with host PC
3. Switch away from red dot (interface board connector): HB talks with host PC

So long as the dongle is connected to the (powered) HB interface, the HB will continue to charge no matter what the switch position.

When you run a program that uses the CMUcam, slide the switch to the red side before you start the program. (If you forget, turn the switch and start your program again).

When you download from Interactive C to your Handyboard, slide the switch to the **unmarked** side.

The middle setting for the switch allows your desktop or laptop computer to talk directly to the CMUcam, allowing you to run the Java Camera GUI.

Charging with the CMUcam

- The CMUcam uses as much power as the HB
- Together they use more power than is supplied by normal charge
- When the camera is connected and the HB is turned on, it is useful to have the charger in ZAP mode. The charger may be left in ZAP for a few minutes, even if the HB is turned off.
- ALWAYS reset the charger to normal mode and turn off the HB when you leave the HB unattended!

Programming with the CMUcam

- To use any camera routines, be sure to put
  `#use "cmucam3.ic"`
  at the top of your file
- You must call
  `init_camera();`
  to initialize the camera before any other camera functions will work
- Use
  `clamp_camera_yuv();`
  to automatically set the camera white balance for the current lighting conditions. The camera should be pointed at a white surface when this call is being made (it waits for the start button to be pressed). It takes 15 seconds for this function to complete!
- Call
  `track_blue();` and/or `track_orange();`
  to check for two types of color blobs that the CMUcam can see (for other colors, you are on your own!). These functions return a "confidence" value when a blob having the right color value is detected. A good confidence is 80 and up. A confidence of 4 or 5 is poor. A confidence of zero means no blob with the right color value was found.

  `track_raw(rmin, rmax, gmin, gmax, bmin, bmax)` is used to track an arbitrary color defined by the values of its parameters. When the camera is in YUV mode then r is really Cr, g is really Y and b is really Cb.

Each time you call `track_blue()` or `track_orange()`, information is stored in global variables as follows:

`track_size`	stores the approximate number of pixels matching in the blob
`track_x`	stores the pixel x coordinate of the centroid of the color blob

- WDFNB\ stores the pixel y coordinate of the color blob
  (note: 0,0 is the center; 40,80 is upper right and -40,-80 is lower left)
- WDFNBDUHD stores the size of the bounding rectangle of the color blob
- WDFNBFRQ[illegible]GHQFH stores the confidence for seeing the blob

Example:

```
#use "cmucamlib.ic"
/* demonstrate color blob sensing for poof balls and blue paper */
void main()
{
   init_camera();          // initialize the camera in YUV mode
   clamp_camera_yuv();      // clamp camera white balance in YUV mode
   while(!stop_button()) {   // hold down Stop for a long time
      if (track_blue()>4) { // you could make this 0 bigger number, like 80 for example
         printf("blue found:%d\n", track_confidence);
      } else if (track_orange()>4) {
         printf("orange found:%d\n", track_confidence);
      } else {
         beep();
         printf("nothing...\n");
      }
   }
}
```

CMUcam notes

- Don't forget to include the [illegible]XVH[illegible]FP XFDP[illegible]E[illegible]F[illegible]
- Once your handyboard has called [illegible]MBFDP HUD[illegible], its serial port is reconfigured to talk to the CMUcam and will remain so until the board is reset; ie., you cannot download a new program once [illegible]MBFDP HUD[illegible] has run, since the Handyboard is now set to talk to the camera rather than IC. In order to download a new program, you will need to restart your Handyboard without running [illegible]MBFDP HUD[illegible]. To do this simply press the start button while restarting the Handyboard (restart means turn off then back on) - this prevents the main program from automatically running.

For experts:

- WDFN5DZ[illegible] allows you track any color range you choose. Like WDFNBEOXH[illegible] and WDFNBRUDQJH[illegible], this function returns the confidence of the blob detected, or 0 for no blob. A return value of -1 means there was an error talking to the CMUcam (e.g. cable disconnected or camera unitialized).
- VHW[illegible]Q[illegible] allows you to tell the CMUcam to only look at a "subwindow" in the image and ignore the rest.
- More details and descriptions of low level functions are given in the comments at the beginning of FP XFDP[illegible]E[illegible]F and FP XFDP[illegible]F, which are in the Handyboard folder that's in the IC directory

Handy Board Library Function Descriptions

(alphabetic order) The return value is in the range [illegible]

DOORII [Category: Motors]

Format: YRLG DOORII
Turns off all motors. DR is a short form for DOORII

DQDORJ [Category: Sensors]

Format: LQW DQDORJ LQW S
Returns value of sensor port numbered p. Result is integer between 0 and 255. If the DQDORJ function is applied to a port that is implemented digitally in hardware, then the value 255 is returned if the hardware digital reading is 1 (as if a digital switch is open, and the pull up resistors are causing a high reading), and the value 0 is returned if the hardware digital reading is 0 (as if a digital switch is closed and pulling the reading near ground). Ports are numbered as marked. Note that on the HB ports 16-22 are floating, so without a sensor inserted, the value cannot be predicted.

DR [Category: Motors]

Format: YRLG DR
Turns off all motors.

DWDQ [Category: Math]

Format: IORDW DWDQ IORDW DQJOH
Returns the arc tangent of the angle. Angle is specified in radians; the result is in radians.

EHHS [Category: Sound]

Format: YRLG EHHS
Produces a tone of 500 Hertz for a period of 0.3 seconds. Returns when the tone is finished.

EHHSHUbRII [Category: Sound]

Format: YRLG EHHSHUbRII
Turns off the beeper.

EHHSHUbRQ [Category: Sound]

Format: YRLG EHHSHUbRQ
Turns on the beeper at last frequency selected by the former function. The beeper remains on until the EHHSHUbRII function is executed.

EN [Category: Motors]

Format: YRLG EN LQW P
Turns motor P on in the backward direction.
Example:
EN

FKRRVHbEXWWRQ [Category: Sensors]

Format: LQW FKRRVHbEXWWRQ
Returns value of button labeled Start.
Example:

```
IIIII Z DLWIRUEXWRQ WR EH SUHVVHG IWKHQ Z DLWIRUEWR EH
IIIIIUHOHDVHG WR WKDWEXWRQ SUHVV LV GHERXQFHG III
IIZ KLOH IIIFKRRVHBEXWRQ IIII^^
IIZ KLOH IIFKRRVHBEXWRQ IIII^^
```

FOHDUBGLJLWDOBRXW [Category: DIO]

Format: LQWFOHDUBGLJLWDOBRXWLQWFKDQQHO
Set expansion board digital output to logic low.

FRV [Category: Math]

Format: IORDWFRVIIORDWDQJOHI
Returns cosine of angle. Angle is specified in radians; result is in radians.

GHIHU [Category: Processes]

Format: YRLG GHIHUII
Makes a process swap out immediately after the function is called. Useful if a process knows that it will not need to do any work until the next time around the scheduler loop. GHIHUII is implemented as a C built-in function.

GLJLWDO [Category: Sensors]

Format: LQWGLJLWDOLQWSI
Returns the value of the sensor in sensor port p, as a true/false value (1 for true and 0 for false). Sensors are expected to be active low, meaning that they are valued at zero volts in the active, or true, state. Thus the library function returns the inverse of the actual reading from the digital hardware: if the reading is zero volts or logic zero, the GLJLWDOII function will return true.

GLVDEOHBHQFRGHU [Category: Sensors]

Format: YRLG GLVDEOHBHQFRGHULQWHQFRGHUI
Disables the given encoder and prevents it from counting. Each shaft encoder uses processing time every time it receives a pulse while enabled, so they should be disabled when you no longer need the encoder's data.

HQDEOHBHQFRGHU [Category: Sensors]

Format: YRLG HQDEOHBHQFRGHULQWHQFRGHUI
Enables the given encoder to start counting pulses and resets its counter to zero. By default encoders start in the disabled state and must be enabled before they start counting.

HVFDSHBEXWRQ [Category: Sensors]

Format: LQWHVFDSHBEXWRQ II
Returns value of button labeled Stop: 1 if pressed and 0 if released. Example:

```
IIIII Z DLWKQWLOHVFDSH EXWRQ SUHVVHG III
IIZ KLOH IIHVFDSHBEXWRQ IIII^^
```

HI[SII [Category: Math]

Run-Time Errors

Format: IORDWH[SIIIIORDWQXP I

Returns 10 to the num power.

When a run-time error occurs, an ERR indicator with error code is displayed on the LCD panel.

Error Code	Description
1	no stack space for VWDUWBSLRFHVV II
2	no process slots remaining
3	array reference out of bounds
4	stack overflow error in running process
5	operation with invalid pointer
6	floating point underflow
7	floating point overflow
8	floating point divide-by-zero
9	number too small or large to convert to integer
10	tried to take square root of negative number
11	tangent of 90 degrees attempted
12	log or ln of negative number or zero
13	unassigned
14	unassigned
15	floating point format error in printf
16	integer divide-by-zero

Using the IC Simulator

1. Setting up to use the simulator
There are 3 requirements that must be fulfilled in order to use the simulator:

- A board that has a valid simulator library must be loaded. If a board does not have a simulator library, a warning will be shown when the user tries to activate the simulator.
- A valid IC file must be loaded
- The IC file must compile with no errors

After these conditions are met, the simulator can be activated.

2.Opening the simulator
With the IC file you would like to simulate selected in the tab interface, hit the "simulate" button on the top toolbar. This will open the simulator window, with the current IC file loaded and ready to run.

3. Running the simulator
To start the simulator running, hit the "Execute" button at the bottom of the simulator window. This will execute whatever code is currently loaded into the simulator. To pause execution, hit the pause button. To resume it, hit the execute button again. To reset the simulator back to its starting state (similar to turning a controller board off and back on), hit the reset button. To exit the simulator and return to the IC compiler window, hit the cancel button.

4. A tour of the simulator window
The simulator provides all the inputs and outputs of the controller board it is simulating, including sensors, motors, and any board specific features.
Sensors are located in the upper left hand box. Sensor type is denoted by the letter next to the sensor number, 'a' standing for analog sensor, 'd' for digital. The values for sensors may be set by either typing in

the value in the corresponding box, or using the up and down arrows to change the value. The simulator does not need to be paused in order to change the value, and the program that is currently running will react as soon as any change is made.
In addition to the usual analog and digital sensors, boards might have certain unique features such as buttons and knobs (i.e. the start/stop buttons on the Handyboard, Prgm/View on the RCX, etc...). These features are available in the box directly below the sensors, and are settable through the same methods as the sensors.
The right side of the simulator window houses the help and output features.
In the upper right hand corner is the Show Diagram button. By pressing the board diagram button, the user can bring up a picture of the controller board being simulated, in order to relate port positions to their physical positions on the actual board.
Below this is the Print Buffer display. This display is used to show what would usually be on the screen of the controller board, and is customized to show the text in the same format as it will be shown on the controller board.
Finally, the motor output values are displayed below the Print Buffer window. These reflect the PWM settings of the motors available on the board.

5. The simulator interaction and global windows
The simulator has an interaction feature, similar to using a real controller board. Users may type commands into the interaction window and it will return the results from the current state of the simulated board. In addition, there is now a global tracking window. This displays the names, types, addresses, and values of globals in the currently loaded program, and updates their values as they change during the course of the program's execution.