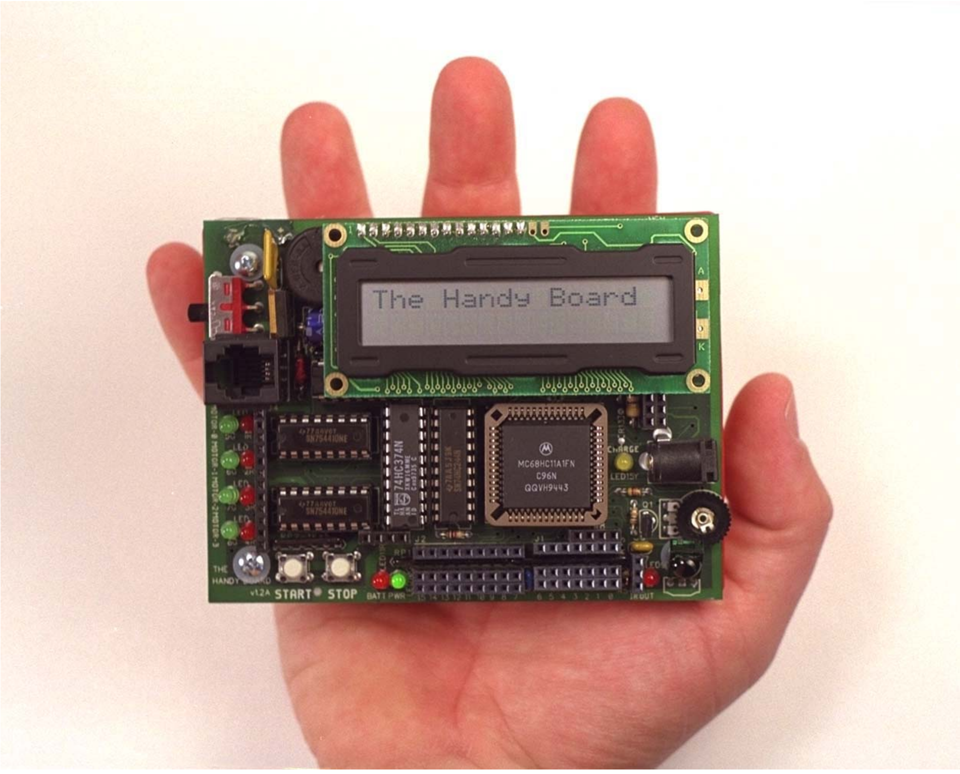
The Handy Board Technical Reference

Fred G. Martin*

November 15, 2000



The Handy Board is a hand-held, battery-powered microcontroller board ideal for personal and educational robotics projects. Based on the Motorola 68HC11 microprocessor, the Handy Board includes 32K of battery-backed static RAM, outputs for four DC motors, inputs for a variety of sensors, and a 16×2 character LCD screen. The Handy Board runs Interactive C, a cross-platform, multi-tasking version of the C programming language.

The Handy Board is distributed under MIT's free licensing policy, in which the design may be licensed for for personal, educational, or commercial use with no charge.

---

*1595 Main Street, Concord MA 01742. E-mail: `fredm@alum.mit.edu`. An electronic version of this document and the freely distributable software described herein are available from the Handy Board home page at `http://handyboard.com/`.

Contents

1 Specifications

The Handy Board features:

- 52–pin Motorola 6811 microprocessor with system clock at 2 MHz.
- 32K of battery-backed CMOS static RAM.
- Two L293D chips capable of driving four DC motors.
- 16 $\times$ 2 character LCD screen.
- Two user-programmable buttons, one knob, and piezo beeper.
- Powered header inputs for 7 analog sensors and 9 digital sensors.
- Internal 9.6v nicad battery with built-in recharging circuit.
- Hardware 38 kHz oscillator and drive transistor for IR output and on-board 38 kHz IR receiver.
- 8-pin powered connector to 6811 SPI circuit (1 Mbaud serial peripheral interface).
- Expansion bus with chip selects allows easy expansion using inexpensive digital I/O latches.
- Board size of 4.25 $\times$ 3.15 inches, designed for a commercial, high grade plastic enclosure which holds battery pack beneath the board.

2 Ports and Connectors

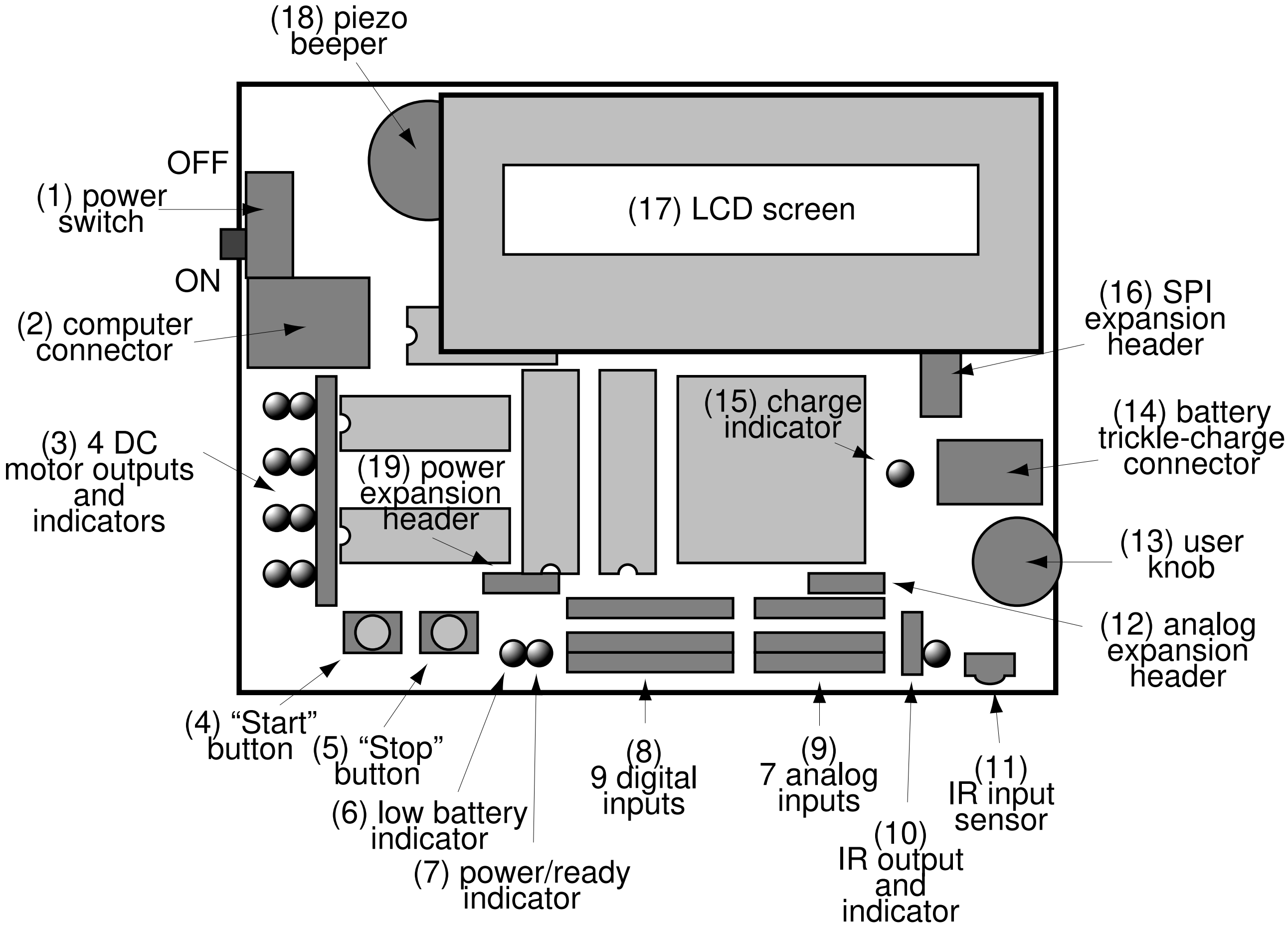


Figure 1: Labeled Handy Board Diagram

Figure 1, above, shows a labeled view of the Handy Board’s ports, connectors, inputs, and outputs. In the following, each of these is briefly described.

1. **Power Switch.** The power switch is used to turn the Handy Board on and off. The Handy Board retains the contents of its memory even when the board is switched off.

2. **Computer Connector.** Via this RJ11 connector, the Handy Board attaches to a desktop computer (using the separate Interface/Charger Board).

3. **4 DC Motor Outputs and Indicators.** The Handy Board’s four motor outputs are located at this single 12–pin connector. Each motor output consists of three pins; the motor connects to the outer two pins and the center pin is not used. Red and green LEDs indicate motor direction. From top to bottom, the motor outputs are numbered 0 to 3.

4. **Start Button.** The Start button is used to control the execution of Interactive C programs. Also, its state may be read under user program control.

5. **Stop Button.** The Stop button is used to put the Handy Board into a special bootstrap download mode. Also, its state may be read under user program control.

6. **Low Battery Indicator.** The red Low Battery LED lights when for a brief interval each time the Handy Board is switched on. If this LED is on steadily, it indicates that the battery is low and that the CPU is halted.

7. **Power/Ready Indicator.** The green Power/Ready LED lights when the Handy Board is in normal operation, and flashes when the Handy Board is transmitting serial data. If the board is powered on and this LED is off, then the Handy Board is in special bootstrap mode.

8. **9 Digital Inputs.** The bank of digital input ports is here. From right to left, the digital inputs are numbered 7 to 15.

9. **7 Analog Inputs.** The bank of analog input ports is here. From right to left, the analog inputs are numbered 0 to 6.

10. **IR Output and Indicator.** The infrared output port is here. The red indicator LED lights when the output is enabled.

11. **IR Input Sensor.** The dark green-colored infrared sensor is here.

12. **Analog Expansion Header.** The analog expansion header is a 1$\times$4 connector row located above analog inputs 0 to 3.

13. **User Knob.** The user knob is a trimmer potentiometer whose value can be read under user program control.

14. **Battery Trickle-Charge Connector.** The battery charge connector is a coaxial power jack to accept a 12 volt signal for trickle-charging the Handy Board's internal battery.

15. **Charge Indicator.** The yellow charge indicator LED lights when the Handy Board is charging via the coaxial power jack.

16. **SPI Expansion Header.** The SPI expansion header is a 2$\times$4 pin jack that allows connection with the 6811's *serial peripheral interface* circuit. See the CPU and memory schematic diagram for a pin-out of this connector.

17. **LCD Screen.** The Handy Board is provided with a 16$\times$2 LCD screen which can display data under user control.

18. **Piezo Beeper.** The Handy Board has a simple piezo beeper for generating tones under user control.

19. **Power Expansion Header.** The power expansion header is a 1$\times$4 pin jack that provides access to the unregulated motor power and ground signals.

3 Quick Start

Here are the steps to getting started with the Handy Board and Interactive C:

1. Connect the Handy Board to the serial port of the host computer, using the separate Serial Interface board. The Serial Interface board connects to the host computer using a standard modem cable; the Handy Board connects to the Serial Interface using a standard 4–wire telephone cable.

2. Put the Handy Board into bootstrap download mode, by holding down the STOP button while turning on system power. The pair of LED's by the two push buttons should light up, and then turn off. When power is on and both of the LED's are off, the Handy Board is in download mode.

3. Run the appropriate downloader for the host computer platform, and download the file `pcode_hb.s19`.

4. Turn the Handy Board off and then on, and the Interactive C welcome message should appear on the Handy Board's LCD screen.

5. Run Interactive C.

4 6811 Downloaders

There are two primary components to the Interactive C software system:

- The *6811 downloader program*, which is used to load the runtime 6811 operating program on the Handy Board. There are a number of different 6811 downloaders for each computer platform.
- The *Interactive C application*, which is used to compile and download IC programs to the Handy Board.

This software is available for a variety of computer platforms/operating systems, including MS-DOS, Windows 3.1/Windows 95, Macintosh, and Unix. The remainder of this section explains the choices in the 6811 downloaders.

4.1 Overview

The 6811 downloaders are general purpose applications for downloading a Motorola hex file (also called an S19 record) into the Handy Board's memory. Each line hex file contains ASCII-encoded binary data indicating what data is to be loaded where into the Handy Board's memory.

For use with Interactive C, the program named "`pcode_hb.s19`" must be present in the Handy Board. The task of the downloaders, then, is simply to initialize the Handy Board's memory with the contents of this file.

An additional purpose of the downloaders is to program the 6811's "CONFIG" register. The CONFIG register determines the nature of the 6811 memory map. For use with Interactive C, the CONFIG register must be set to the value `0x0c`, which allows the 6811 to access the Handy Board's 32K static RAM memory in its entirety. Some downloaders automatically program the CONFIG register; others require a special procedure to do so. Please note that programming of the CONFIG register *only needs to be done once* to factory-fresh 6811's. It is then set in firmware until deliberately reprogrammed to a different value.

Another consideration related to downloaders is the type of 6811 in use. The Handy Board can use both the "A" and "E" series of 6811. These two chip varieties are quite similar, but not all downloaders support the E series' bootstrap sequence. (The E series chips have more flexibility on their Port A input/output pins and can run at a higher clock speed.)

4.2 Putting the Handy Board into Bootstrap Download Mode

When using any of the downloaders, the Handy Board must first be put into its bootstrap download mode. This is done by first turning the board off, and then turning it on *while holding down the STOP button* (the button closer to the pair of LEDs to the right of the buttons). When the board is first turned on, these two LEDs should light for about $\frac{1}{3}$ of a second and then both should turn off. The STOP button must be held down continuously during this sequence. *When the board is powered on and both of these LEDs are off, it is ready for bootstrap download.*

4.3 MS-DOS

Two downloaders are available for MS-DOS machines: *dl*, by Randy Sargent and *dlm*, by Fred Martin.

dl is compatible only with the A series of 6811, and automatically programs the CONFIG register. Type "`dl pcode_hb.s19`" at the MS-DOS prompt.

dlm is compatible with both the A and E series of 6811, but does not automatically program the CONFIG register. Type "`dlm pcode_hb.s19 -256`" to download to an A series chip and "`dlm pcode_hb.s19 -512`" to download to an E series chip.

Neither *dl* nor *dlm* runs very well under Windows. It is generally necessary to run them from a full-screen DOS shell to get them to work at all. Under Windows, *hbdl* is recommended instead.

4.4 Windows 3.1 and Windows 95

hbdl, by Vadim Gerasimov, is the recommended Windows 6811 downloader. *hbdl* features automatic recognition of both A and E series 6811s and automatic programming of the CONFIG register.

To use *hbdl*, run the `hbdl.exe` application and select the "`pcode_hb.s19`" file for download. Make sure the text box for the CONFIG register has the value "`0c`."

4.5 Macintosh

There are two choices available for the Macintosh: *Initialize Board*, by Randy Sargent, and *6811 Downloader MCL*, by Fred Martin.

Initialize Board features automatic programming of the CONFIG register, but only works with A series 6811's. It comes in two versions, one using the modem port and one using the printer port.

In order to get *Initialize Board* to use the Handy Board's `pcode_hb.s19` file, one must edit its `STR` resources to name this file. Then using it is just a matter of double-clicking on the application icon.

6811 Downloader MCL features automatic recognition of both A and E series 6811's. In order to program the CONFIG register, one can select the *Set Config...* option from the *HC11* menu.

6811 Downloader MCL is run by double-clicking on the application icon and typing the name of the file to be downloaded into a text field. The S19 file to be downloaded must be located in the same folder as the application.

An earlier version of *6811 Downloader* (note the lack of the MCL suffix in the application name) may no longer be compatible with contemporary Macintosh designs.

4.6 Unix

The *dl* downloader, written by Randy Sargent, is available for a number of Unix platforms, including DECstations, Linux, Sparc Solaris, Sparc Sun OS, SGI, HPUX, and RS6000.

This downloader only works with the A series of 6811, and supports automatic programming of the CONFIG register.

5 Interactive C

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, 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 and prevents crashing. For example, IC does array bounds checking at run-time to protect against 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. Still, IC is no slouch.

IC was designed and implemented by Randy Sargent with the assistance of Fred Martin.
This manual covers the freeware distribution of IC (version 2.8x).

5.1 Using IC

When IC is booted, it immediately attempts to connect with the Handy Board, which should be turned on and running the `pcode_hb.s19` program.

After synchronizing with the Handy Board, IC compiles and downloads the default set of library files, and then presents the user with the "`C>`" prompt. At this prompt, either an IC command or C–language expression may be entered.

All C expressions must be ended with a semicolon. For example, to evaluate the arithmetic expression $1 + 2$, type the following:

```
C> 1 + 2;
```

(The underlined portion indicates user input.) When this expression is typed, it is compiled by IC and then downloaded to the Handy Board for evaluation. The Handy Board then evaluates the compiled form and returns the result, which is printed on the IC console.

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 `i` and prints the sum `i+7` to the Handy Board's LCD screen:

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

5.1.1 IC Commands

IC responds to the following commands:

- **Load file.** The command `load` <*filename*> compiles and loads the named file. The Handy Board must be attached for this to work. IC looks first in the local directory and then in the IC library path for files.

  Several files may be loaded into IC at once, allowing programs to be defined in multiple files.

- **Unload file.** The command `unload` < *filename* > unloads the named file, and re-downloads remaining files.

- **List files, functions, or globals.** The command `list files` displays the names of all files presently loaded into IC. The command `list functions` displays the names of presently defined C functions. The command `list globals` displays the names of all currently defined global variables.

- **Kill all processes.** The command `kill_all` kills all currently running processes.

- **Print process status.** The command `ps` prints the status of currently running processes.

- **Help.** The command `help` displays a help screen of IC commands.

- **Quit.** The command `quit` exits IC. In the MS-DOS version, CTRL-C can also be used.

5.1.2 Line Editing

IC has a built-in line editor and command history, allowing editing and re-use of previously typed statements and commands. The mnemonics for these functions are based on standard Emacs control key assignments.

To scan forward and backward in the command history, type CTRL-P or $\boxed{\uparrow}$ for backward, and CTRL-N or $\boxed{\downarrow}$ for forward.

Figure 2 shows the keystroke mappings understood by IC.

IC does parenthesis-balance-highlighting as expressions are typed.

5.1.3 The Main Function

After functions have been downloaded to the Handy Board, they can be invoked from the IC prompt. If one of the functions is named `main()`, it will automatically be run when the Handy Board is reset.

To reset the Handy Board *without* running the `main()` function (for instance, when hooking the board back to the computer), hold down the START button when turning on the Handy Board. The board will reset without running `main()`.

Keystroke	Function
CTRL-A	beginning-of-line
CTRL-B	backward-char
$\boxed{\leftarrow}$	backward-char
CTRL-D	delete-char
CTRL-E	end-of-line
CTRL-F	forward-char
$\boxed{\rightarrow}$	forward-char
CTRL-K	kill-line

Figure 2: IC Command-Line Keystroke Mappings

5.2 A Quick C Tutorial

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

```
void main()
{
    printf("Hello, world!\n");
}
```

All functions must have a return value; that is, the value that they return when they finish execution. `main` has a return value type of `void`, which is the "null" type. Other types include integers (`int`) and floating point numbers (`float`). 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 a 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 `printf` (formatted print). This will print the message "`Hello, world!`" to the LCD display. The `\n` indicates end-of-line.

The `printf` 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 at the end of 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 `int`, which means that it will return an integer value. Next comes the function name `square`, followed by its argument list in parenthesis. `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 `return` statement. The `return` statement exits the function and returns the value of the C *expression* that follows it (in this case "`n * n`").

Expressions are evaluated according set of precendence rules depending on 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 `sqrt` function. It turns out that `sqrt` is a built-in 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 `sqrt`.

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

This concludes the brief C tutorial.

5.3 Data Types, Operations, and Expressions

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. Operators specify what is to be done to them. Expressions combine variables and constants to create new values.

5.3.1 Variable Names

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 `if`, `while`, etc. may not be used as variable names.

Global variables and functions may not have the same name. In addition, local variables named the same as functions prevent the use of that function within the scope of the local variable.

5.3.2 Data Types

IC supports the following data types:

16-bit Integers 16-bit integers are signified by the type indicator `int`. They are signed integers, and may be valued from $-32{,}768$ to $+32{,}767$ decimal.

32-bit Integers 32-bit integers are signified by the type indicator `long`. They are signed integers, and may be valued from $-2{,}147{,}483{,}648$ to $+2{,}147{,}483{,}647$ decimal.

32-bit Floating Point Numbers Floating point numbers are signified by the type indicator `float`. They have approximately seven decimal digits of precision and are valued from about 10^{-38} to 10^{38}.

8-bit Characters Characters are an 8-bit number signified by the type indicator `char`. A character's value typically represents a printable symbol using the standard ASCII character code.

Arrays of characters (character strings) are supported, but individual characters are not.

5.3.3 Local and Global Variables

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 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 that are defined in files other than the one in which the global variable was declared.

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

```
int foo()
{
  int x;        /* create local variable x
                   with initial value 0    */
  int y= 7;     /* create local variable y
                   with initial value 7    */
  ...
}

float z=3.0;    /* create global variable z
                   with initial value 3.0  */
```

Local variables are initialized whenever the function containing them runs.

Global variables are initialized whenever a reset condition occurs. Reset conditions occur when:

1. New code is downloaded;
2. The `main()` procedure is run;
3. System hardware reset occurs.

Persistent Global Variables A special *uninitialized* form of global variable, called the "persistent" type, has been implemented for IC. A persistent global is *not* initialized upon the conditions listed for normal global variables.

To make a persistent global variable, prefix the type specifier with the key word `persistent`. For example, the statement

```
persistent int i;
```

creates a global integer called `i`. The initial value for a persistent variable is arbitrary; it depends on the contents of RAM that were assigned to it. Initial values for persistent variables cannot be specified in their declaration statement.

Persistent variables keep their state when the Handy Board is turned off and on, when `main` is run, and when system reset occurs. Persistent variables, in general, will lose their state when a new program is downloaded. However, it is possible to prevent this from occurring. If persistent variables are declared at the beginning of the code, before any function or non-persistent globals, they will be re-assigned to the same location in memory when the code is re-compiled, and thus their values will be preserved over multiple downloads.

If the program is divided into multiple files and it is desired to preserve the values of persistent variables, then all of the persistent variables should be declared in one particular file and that file should be placed first in the load ordering of the files.

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.

5.3.4 Constants

Integers Integers 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 Integers 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 Numbers Floating point numbers may use exponential notation (e.g., "`10e3`" or "`10E3`") or must contain the decimal period. For example, the floating point zero can be given as "`0.`", "`0.0`", or "`0E1`", but not as just "`0`".

Characters and Character Strings Quoted characters return their ASCII value (e.g., `'x'`).

Character strings are defined with quotation marks, e.g., `"This is a character string."`.

5.3.5 Operators

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

Integers The following operations are supported on integers:

- **Arithmetic.** addition +, subtraction -, multiplication *, division /.
- **Comparison.** greater-than >, less-than <, equality ==, greater-than-equal >=, less-than-equal <=.
- **Bitwise Arithmetic.** bitwise-OR |, bitwise-AND &, bitwise-exclusive-OR ^, bitwise-NOT ~.
- **Boolean Arithmetic.** logical-OR ||, logical-AND &&, logical-NOT !.

  When a C statement uses a boolean value (for example, `if`), 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 `&&` and `||` stop executing as soon as the truth of the final expression is determined. For example, in the expression `a && b`, if `a` is false, then `b` does not need to be evaluated because the result must be false. The `&&` operator "knows this" and does not evaluate `b`.

Long Integers 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 IC uses a package of public-domain floating point routines distributed by Motorola. This package includes arithmetic, trigonometric, and logarithmic functions.

The following operations are supported on floating point numbers:

- **Arithmetic.** addition +, subtraction -, multiplication *, division /.
- **Comparison.** greater-than >, less-than <, equality ==, greater-than-equal >=, less-than-equal <=.
- **Built-in Math Functions.** A set of trigonometric, logarithmic, and exponential functions is supported, as discussed in Section 5.9 of this document.

Characters 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).

5.3.6 Assignment Operators and Expressions

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

```
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:

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

5.3.7 Increment and Decrement Operators

The increment operator "++" increments the named variable. For example, the statement "`a++`" is equivalent to "`a= a+1`" or "`a+= 1`".

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 "`a=3 a+1=4`."

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 "`a=3 a+1=3`" but would finish with `a` set to 4.

The decrement operator "`--`" is used in the same fashion as the increment operator.

5.3.8 Precedence and Order of Evaluation

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**
`() []`	left to right
`! ~ ++ -- - (` *type* `)`	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
`||`	right to left
`= += -=` etc.	right to left
`,`	left to right

5.4 Control Flow

IC supports most of the standard C control structures. One notable exception is the `case` and `switch` statement, which is not supported.

5.4.1 Statements and Blocks

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.

There is never a semicolon after a right brace that ends a block.

5.4.2 If-Else

The `if else` 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 `else` clause is optional. If the `if` part of the statement did not execute, and the `else` is present, then *statement-2* executes.

5.4.3 While

The syntax of a `while` loop is the following:

`while (` *expression* `)`
  statement

`while` 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 `while` statement:

`while (1)`
  statement

5.4.4 For

The syntax of a `for` loop is the following:

`for (` *expr-1* `;` *expr-2* `;` *expr-3* `)`
  statement

This is equivalent to the following construct using `while`:

expr-1 ;
while (*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);
```

5.4.5 Break

Use of the `break` provides an early exit from a `while` or a `for` loop.

5.5 LCD Screen Printing

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

The syntax of `printf` is the following:

`printf(` *format-string* `, [` *arg-1* `] , ..., [` *arg-N* `] )`

This is best illustrated by some examples.

5.5.1 Printing Examples

Example 1: Printing a message. 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 "`\n`" 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 `printf` statements are terminated by a `\n`.

Example 2: Printing a number. The following statement prints the value of the integer variable `x` with a brief message.

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

The special form `%d` is used to format the printing of an integer in decimal format.

Example 3: Printing a number in binary. The following statement prints the value of the integer variable `x` as a binary number.

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

The special form `%b` 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. The following statement prints the value of the floating point variable `n` as a floating point number.

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

The special form `%f` is used to format the printing of floating point number.

Example 5: Printing two numbers in hexadecimal format.

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

The form `%x` formats an integer to print in hexadecimal.

5.5.2 Formatting Command Summary

Format Command	Data Type	Description
`%d`	`int`	decimal number
`%x`	`int`	hexadecimal number
`%b`	`int`	low byte as binary number
`%c`	`int`	low byte as ASCII character
`%f`	`float`	floating point number
`%s`	`char array`	char array (string)

5.5.3 Special Notes

- The final character position of the LCD screen is used as a system "heartbeat." This character continuously blinks back and forth when the board is operating properly. If the character stops blinking, the Handy Board has crashed.
- Characters that would be printed beyond the final character position are truncated.
- The `printf()` command treats the two-line LCD screen as a single longer line.
- Printing of long integers is not presently supported.

5.6 Arrays and Pointers

IC supports one-dimensional arrays of characters, integers, long integers, and floating-point numbers. Pointers to data items and arrays are supported.

5.6.1 Declaring and Initializing Arrays

Arrays are declared using the 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. Using this syntax, the size of the array would not specified within the square braces; it 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]` equalling 0, `foo[1]` equalling 4, etc.

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.

5.6.2 Passing Arrays as Arguments

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. IC only allows declaring array arguments 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 an array of integers.

When passing an array variable to a function, use of the square brackets is not needed:

```
{
int array[10];

retrieve_element(3, array);
}
```

5.6.3 Declaring Pointer Variables

Pointers can be passed to functions which then go on to modify the value of the variable being pointed to. This is useful because the same function can be called to modify different variables, just by giving it a different pointer.

Pointers are declared with the use of the asterisk (`*`). In the example

```
int *foo;
float *bar;
```

`foo` is declared as a pointer to an integer, and `bar` is declared as a pointer to a floating point number.

To make a pointer variable point at some other variable, the ampersand operator is used. The ampersand operator returns the *address* of a variable's value; that is, the place in memory where the variable's value is stored. Thus:

```
int *foo;
int x= 5;

foo= &x;
```

makes the pointer `foo` "point at" the value of `x` (which happens to be 5).

This pointer can now be used to retrieve the value of `x` using the asterisk operator. This process is called *de-referencing*. The pointer, or reference to a value, is used to fetch the value being pointed at. Thus:

```
int y;

y= *foo;
```

sets `y` equal to the value pointed at by `foo`. In the previous example, `foo` was set to point at `x`, which had the value 5. Thus, the result of dereferencing `foo` yields 5, and `y` will be set to 5.

5.6.4 Passing Pointers as Arguments

Pointers can be passed to functions; then, functions can change the values of the variables that are pointed at. This is termed *call-by-reference*; the reference, or pointer, to the 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`.

In the code, the function argument is specified as a pointer using the asterisk:

```
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 `void`. It does not need to return anything, because it instead stores its answer in 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 asterisk is used to get the *location* pointed by `result`.

5.7 Library Functions

Library files provide standard C functions for interfacing with hardware on the Handy 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. The name of the default library file is is contained as a resource within the IC application. On command-line versions of IC, this resource may be modified by invoking "`ic -config`". On the Macintosh, the IC application has a `STR` resource that defines the name of the library file.

The Handy Board's root library file is named `lib_hb.lis`.

5.7.1 Output Control

DC Motors DC motor ports are numbered from 0 to 3.

Motors may be set in a "forward" direction (corresponding to the green motor LED being lit) and a "backward" direction (corresponding to the motor red LED being lit).

The functions `fd(int m)` and `bk(int m)` turn motor `m` on or off, respectively, at full power. The function `off(int m)` turns motor `m` off.

The power level of motors may also be controlled. This is done in software by a motor on and off rapidly (a technique called *pulse-width modulation*. The `motor(int m, int p)` function allows control of a motor's power level. Powers range from `100` (full on in the forward direction) to `-100` (full on in the backward direction). The system software actually only controls motors to seven degrees of power, but argument bounds of -100 and $+100$ are used.

`void fd(int m)`

Turns motor `m` on in the forward direction. Example: `fd(3);`

`void bk(int m)`

Turns motor `m` on in the backward direction. Example: `bk(1);`

`void off(int m)`

Turns off motor `m`. Example: `off(1);`

`void alloff()`

`void ao()`

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

`void motor(int m, int p)`

Turns on motor `m` at power level `p`. Power levels range from `100` for full on forward to `-100` for full on backward.

Servo Motor A library routine allows control of a single servo motor, using digital input 9, which is actually the 6811's Port A bit 7 (PA7), a bidirectional control pin. Loading the servo library files causes this pin to be employed as a digital output suitable for driving the control wire of the servo motor.

The servo motor has a three-wire connection: power, ground, and control. These wires are often color-coded red, black, and white, respectively. The control wire is connected to PA7; the ground wire, to board ground; the power wire, to a +5 volt source. The Handy Board's regulated +5v supply may be used, though this is not an ideal solution because it will tax the regulator. A better solution is a separate battery with a common ground to the Handy Board or a tap at the +6v position of the Handy Board's battery back.

The position of the servo motor shaft is controlled by a rectangular waveform that is generated on the PA7 pin. The duration of the positive pulse of the waveform determines the position of the shaft. This pulse repeats every 20 milliseconds.

The length of the pulse is set by the library function `servo`, or by functions calibrated to set the position of the servo by angle.

`void servo_on()`

Enables PA7 servo output waveform.

`void servo_on()`

Disables PA7 servo output waveform.

`int servo(int period)`

Sets length of servo control pulse. Value is the time in half-microseconds of the positive portion of a rectangular wave that is generated on the PA7 pin for use in controlling a servo motor. Minimum allowable value is 1400 (i.e., 700 μsec); maximum is 4860.

Function return value is actual period set by driver software.

`int servo_rad(float angle)`

Sets servo angle in radians.

`int servo_deg(float angle)`

Sets servo angle in degrees.

In order to use the servo motor functions, the files `servo.icb` and `servo.c` must be loaded.

5.7.2 Sensor Input

`int digital(int p)`

Returns the value of the sensor in sensor port `p`, as a true/false value (1 for true and 0 for false).

Sensors are expected to be *active low*, meaning that they are valued at zero volts in the active, or true, state. Thus the library function returns the inverse of the actual reading from the digital hardware: if the reading is zero volts or logic zero, the `digital()` function will return true.

If the `digital()` function is applied to port that is implemented in hardware as an analog input, the result is true if the analog measurement is less than 127, and false if the reading is greater than or equal to 127.

Ports are numbered as marked on the Handy Board.

`int analog(int p)`

Returns value of sensor port numbered `p`. Result is integer between 0 and 255.

If the `analog()` function is applied to a port that is implemented digitally in hardware, then the value 0 is returned if the digital reading is 0, and the value 255 is returned if the digital reading is 1.

Ports are numbered as marked on the Handy Board.

User Buttons and Knob The Handy Board has two buttons and a knob whose value can be read by user programs.

`int stop_button()`

Returns value of button labelled STOP: 1 if pressed and 0 if released.
Example:

```
/*  wait until stop button pressed  */
while (!stop_button()) {}
```

`int start_button()`

Returns value of button labelled START.

`void stop_press()`

Waits for STOP button to be pressed, then released. Then issues a short beep and returns.
The code for `stop_press` is as follows:

```
while (!stop_button());
while (stop_button());
beep();
```

`void start_press()`

Like `stop_press`, but for the START button.

`int knob()`

Returns the position of a knob as a value from 0 to 255.

Infrared Subsystem The Handy Board provides an on-board infrared receiver (the Sharp IS1U60), for infrared input, and a 40 kHz modulation and power drive circuit, for infrared output. The output circuit requires an external infrared LED.

As of this writing, only the infrared receive function is officially supported. On the Handy Board web site, contributed software to extend the infrared functionality is available.

To use the infrared reception function, the file `sony-ir.icb` must be loaded into Interactive C. This file may be added to the Handy Board default library file, `lib_hb.lis`. *Please make sure that the file* `r22_ir.lis` *is not present in the* `lib_hb.lis` *file.*

The `sony-ir.icb` file adds the capability of receiving infrared codes transmitted by a Sony remote, or a universal remote programmed to transmit Sony infrared codes.

`int sony_init(1)`

Enables the infrared driver.

`int sony_init(0)`

Disables the infrared driver.

`int ir_data(int dummy)`

Returns the data byte last received by the driver, or zero if no data has been received since the last call. A value must be provided for the `dummy` argument, but its value is ignored.

The infrared sensor is the dark green component in the Handy Board's lower right hand corner.

5.7.3 Time Commands

System code keeps track of time passage in milliseconds. The time variables are implemented using the long integer data type. Standard functions allow use floating point variables when using the timing functions.

`void reset_system_time()`

Resets the count of system time to zero milliseconds.

`long mseconds()`

Returns the count of system time in milliseconds. Time count is reset by hardware reset (i.e., turning the board off and on) or the function `reset_system_time()`. `mseconds()` is implemented as a C primitive (not as a library function).

`float seconds()`

Returns the count of system time in seconds, as a floating point number. Resolution is one millisecond.

`void sleep(float sec)`

Waits for an amount of time equal to or slightly greater than `sec` seconds. `sec` is a floating point number.

Example:

```
/*  wait for 1.5 seconds  */
sleep(1.5);
```

`void msleep(long msec)`

Waits for an amount of time equal to or greater than `msec` milliseconds. `msec` is a long integer. Example:

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

5.7.4 Tone Functions

Several commands are provided for producing tones on the standard beeper.

`void beep()`

Produces a tone of 500 Hertz for a period of 0.3 seconds.

`void tone(float frequency, float length)`

Produces a tone at pitch `frequency` Hertz for `length` seconds. Both `frequency` and `length` are floats.

`void set_beeper_pitch(float frequency)`

Sets the beeper tone to be `frequency` Hz. The subsequent function is then used to turn the beeper on.

`void beeper_on()`

Turns on the beeper at last frequency selected by the former function.

`void beeper_off()`

Turns off the beeper.

5.8 Multi-Tasking

5.8.1 Overview

One of the most powerful features of IC is its multi-tasking facility. Processes can be created and destroyed dynamically during run-time.

Any C function can be spawned as a separate task. Multiple tasks running the same code, but with their own local variables, can be created.

Processes communicate through global variables: one process can set a global to some value, and another process can read the value of that global.

Each time a process runs, it executes for a certain number of *ticks*, defined in milliseconds. This value is determined for each process at the time it is created. The default number of ticks is five; therefore, a default process will run for 5 milliseconds until its "turn" ends and the next process is run. All processes are kept track of in a *process table*; each time through the table, each process runs once (for an amount of time equal to its number of ticks).

Each process has its own *program stack*. The stack is used to pass arguments for function calls, store local variables, and store return addresses from function calls. The size of this stack is defined at the time a process is created. The default size of a process stack is 256 bytes.

Processes that make extensive use of recursion or use large local arrays will probably require a stack size larger than the default. Each function call requires two stack bytes (for the return address) plus the number of argument bytes; if the function that is called creates local variables, then they also use up stack space. In addition, C expressions create intermediate values that are stored on the stack.

It is up to the programmer to determine if a particular process requires a stack size larger than the default. A process may also be created with a stack size *smaller* than the default, in order to save stack memory space, if it is known that the process will not require the full default amount.

When a process is created, it is assigned a unique *process identification number* or *pid*. This number can be used to kill a process.

5.8.2 Creating New Processes

The function to create a new process is `start_process`. `start_process` takes one mandatory argument—the function call to be started as a process. There are two optional arguments: the process's number of ticks and stack size. (If only one optional argument is given, it is assumed to be the ticks number, and the default stack size is used.)

`start_process` has the following syntax:

`int start_process(` *function-call(...)* `, [`*TICKS*`] , [`*STACK-SIZE*`] )`

`start_process` returns an integer, which is the process ID assigned to the new process.

The function call may be any valid call of the function used. The following code shows the function `main` creating a process:

```
void check_sensor(int n)
{
  while (1)
    printf("Sensor %d is %d\n", n, digital(n));
}

void main()
{
  start_process(check_sensor(2));
}
```

Normally when a C functions ends, it exits with a return value or the "void" value. If a function invoked as a process ends, it "dies," letting its return value (if there was one) disappear. (This is okay, because processes communicate results by storing them in globals, not by returning them as return values.) Hence in the above example, the `check_sensor` function is defined as an infinite loop, so as to run forever (until the board is reset or a `kill_process` is executed).

Creating a process with a non-default number of ticks or a non-default stack size is simply a matter of using `start_process` with optional arguments; e.g.

```
start_process(check_sensor(2), 1, 50);
```

will create a `check_sensor` process that runs for 1 milliseconds per invocation and has a stack size of 50 bytes (for the given definition of `check_sensor`, a small stack space would be sufficient).

5.8.3 Destroying Processes

The `kill_process` function is used to destroy processes. Processes are destroyed by passing their process ID number to `kill_process`, according to the following syntax:

```
int kill_process(int pid)
```

`kill_process` returns a value indicating if the operation was successful. If the return value is `0`, then the process was destroyed. If the return value is `1`, then the process was not found.

The following code shows the `main` process creating a `check_sensor` process, and then destroying it one second later:

```
void main()
{
  int pid;

  pid= start_process(check_sensor(2));
  sleep(1.0);
  kill_process(pid);
}
```

5.8.4 Process Management Commands

IC has two commands to help with process management. The commands only work when used at the IC command line. They are not C functions that can be used in code.

`kill_all`

kills all currently running processes.

`ps`

prints out a list of the process status.

The following information is presented: process ID, status code, program counter, stack pointer, stack pointer origin, number of ticks, and name of function that is currently executing.

5.8.5 Process Management Library Functions

The following functions are implemented in the standard C library.

`void hog_processor()`

Allocates an additional 256 milliseconds of execution to the currently running process. If this function is called repeatedly, the system will wedge and only execute the process that is calling `hog_processor()`. Only a system reset will unwedge from this state. Needless to say, this function should be used with extreme care, and should not be placed in a loop, unless wedging the machine is the desired outcome.

`void defer()`

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. `defer()` is implemented as a C built-in function.

5.9 Floating Point Functions

In addition to basic floating point arithmetic (addition, subtraction, multiplication, and division) and floating point comparisons, a number of exponential and transcendental functions are built in to IC. These are implemented with a public domain library of routines provided by Motorola.

Keep in mind that all floating point operations are quite slow; each takes one to several milliseconds to complete. If Interactive C's speed seems to be poor, extensive use of floating point operations is a likely cause.

`float sin(float angle)`
Returns sine of `angle`. Angle is specified in radians; result is in radians.

`float cos(float angle)`
Returns cosine of `angle`. Angle is specified in radians; result is in radians.

`float tan(float angle)`
Returns tangent of `angle`. Angle is specified in radians; result is in radians.

`float atan(float angle)`
Returns arc tangent of `angle`. Angle is specified in radians; result is in radians.

`float sqrt(float num)`
Returns square root of `num`.

`float log10(float num)`
Returns logarithm of `num` to the base 10.

`float log(float num)`
Returns natural logarithm of `num`.

`float exp10(float num)`
Returns 10 to the `num` power.

`float exp(float num)`
Returns e to the `num` power.

`(float) a ^ (float) b`
Returns `a` to the `b` power.

5.10 Memory Access Functions

IC has primitives for directly examining and modifying memory contents. These should be used with care as it would be easy to corrupt memory and crash the system using these functions.

There should be little need to use these functions. Most interaction with system memory should be done with arrays and/or globals.

`int peek(int loc)`
Returns the byte located at address `loc`.

`int peekword(int loc)`
Returns the 16-bit value located at address `loc` and `loc`+1. `loc` has the most significant byte, as per the 6811 16-bit addressing standard.

`void poke(int loc, int byte)`
Stores the 8-bit value `byte` at memory address `loc`.

`void pokeword(int loc, int word)`
Stores the 16-bit value `word` at memory addresses `loc` and `loc`+1.

`void bit_set(int loc, int mask)`
Sets bits that are set in `mask` at memory address `loc`.

`void bit_clear(int loc, int mask)`
Clears bits that are set in `mask` at memory address `loc`.

5.11 Error Handling

There are two types of errors that can happen when working with IC: *compile-time* errors and *run-time* errors.

Compile-time errors occur during the compilation of the source file. They are indicative of mistakes in the C source code. Typical compile-time errors result from incorrect syntax or mis-matching of data types.

Run-time errors occur while a program is running on the board. They indicate problems with a valid C form when it is running. A simple example would be a divide-by-zero error. Another example might be running out of stack space, if a recursive procedure goes too deep in recursion.

These types of errors are handled differently, as is explained below.

5.11.1 Compile-Time Errors

When compiler errors occur, an error message is printed to the screen. All compile-time errors must be fixed before a file can be downloaded to the board.

5.11.2 Run-Time Errors

When a run-time error occurs, an error message is displayed on the LCD screen indicating the error number. If the board is hooked up to ICwhen the error occurs, a more verbose error message is printed on the terminal.

Here is a list of the run-time error codes:

Error Code	Description
1	no stack space for `start_process()`
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
15	floating point format error in printf
16	integer divide-by-zero

5.12 Binary Programs

With the use of a customized 6811 assembler program, IC allows the use of machine language programs within the C environment. There are two ways that machine language programs may be incorporated:

1. Programs may be called from C as if they were C functions.
2. Programs may install themselves into the interrupt structure of the 6811, running repetitiously or when invoked due to a hardware or software interrupt.

When operating as a function, the interface between C and a binary program is limited: a binary program must be given one integer as an argument, and will return an integer as its return value. However, programs in a binary file can declare any number of global integer variables in the C environment. Also, the binary program can use its argument as a pointer to a C data structure.

5.12.1 The Binary Source File

Special keywords in the source assembly language file (or module) are used to establish the following features of the binary program:

Entry point. The entry point for calls to each program defined in the binary file.

Initialization entry point. Each file may have one routine that is called automatically upon a reset condition. (The reset conditions are explained in Section 5.3.3, which discusses global variable initialization.) This initialization routine particularly useful for programs which will function as interrupt routines.

C variable definitions. Any number of two-byte C integer variables may be declared within a binary file. When the module is loaded into IC, these variables become defined as globals in C.

```
/* Sample icb file */

/* origin for module and variables */
        ORG     MAIN_START

/* program to return twice the argument passed to us */
subroutine_double:
        ASLD
        RTS

/* declaration for the variable "foo" */
variable_foo:
        FDB     55

/* program to set the C variable "foo" */
subroutine_set_foo:
        STD     variable_foo
        RTS

/* program to retrieve the variable "foo" */
subroutine_get_foo:
        LDD     variable_foo
        RTS

/* code that runs on reset conditions */
subroutine_initialize_module:
        LDD     #69
        STD     variable_foo
        RTS
```

Figure 3: Sample IC Binary Source File: `testicb.asm`

To explain how these features work, let's look at a sample IC binary source program, listed in Figure 3.

The first statement of the file ("`ORG MAIN_START`") declares the start of the binary programs. This line must precede the code itself itself.

The entry point for a program to be called from C is declared with a special form beginning with the text `subroutine_`. In this case, the name of the binary program is `double`, so the label is named `subroutine_double`. As the comment indicates, this is a program that will double the value of the argument passed to it.

When the binary program is called from C, it is passed one integer argument. This argument is placed in the 6811's D register (also known as the "Double Accumulator") before the binary code is called.

The `double` program doubles the number in the D register. The `ASLD` instruction ("Arithmetic Shift Left Double [Accumulator]") is equivalent to multiplying by 2; hence this doubles the number in the D register.

The `RTS` instruction is "Return from Subroutine." All binary programs must exit using this instruction. When a binary program exits, the value in the D register is the return value to C. Thus, the `double` program doubles its C argument and returns it to C.

Declaring Variables in Binary Files The label `variable_foo` is an example of a special form to declare the name and location of a variable accessable from C. The special label prefix "`variable_`" is followed the name of the variable, in this case, "`foo`."

This label must be immediately followed by the statement `FDB <number>`. This is an assembler directive that creates a two-byte value (which is the initial value of the variable).

Variables used by binary programs must be declared in the binary file. These variables then become C globals when the binary file is loaded into C.

The next binary program in the file is named "`set_foo`." It performs the action of setting the value of the variable `foo`, which is defined later in the file. It does this by storing the D register into the memory contents reserved for `foo`, and then returning.

The next binary program is named "`get_foo`." It loads the D register from the memory reserved for `foo` and then returns.

Declaring an Initialization Program The label `subroutine_initialize_module` is a special form used to indicate the entry point for code that should be run to initialize the binary programs. This code is run upon standard reset conditions: program download, hardware reset, or running of the `main()` function.

In the example shown, the initialization code stores the value 69 into the location reserved for the variable `foo`. This then overwrites the 55 which would otherwise be the default value for that variable.

Initialization of globals variables defined in an binary module is done differently than globals defined in C. In a binary module, the globals are initialized to the value declared by the `FDB` statement only when the code is downloaded to the 6811 board (not upon reset or running of main, like normal globals).

However, the initialization routine is run upon standard reset conditions, and can be used to initialize globals, as this example has illustrated.

5.12.2 Interrupt-Driven Binary Programs

Interrupt-driven binary programs use the initialization sequence of the binary module to install a piece of code into the interrupt structure of the 6811.

The 6811 has a number of different interrupts, mostly dealing with its on-chip hardware such as timers and counters. One of these interrupts is used by the runtime software to implement time-keeping and other periodic functions (such as LCD screen management). This interrupt, dubbed the "System Interrupt," runs at 1000 Hertz.

Instead of using another 6811 interrupt to run user binary programs, additional programs (that need to run at 1000 Hz. or less) may install themselves into the System Interrupt. User programs would be then become part of the 1000 Hz interrupt sequence.

This is accomplished by having the user program "intercept" the original 6811 interrupt vector that points to runtime interrupt code. This vector is made to point at the user program. When user program finishes, it jumps to the start of the runtime interrupt code.

Figure 4 depicts the interrupt structure before user program installation. The 6811 vector location points to system software code, which terminates in a "return from interrupt" instruction.

Figure 5 illustrates the result after the user program is installed. The 6811 vector points to the user program, which exits by jumping to the system software driver. This driver exits as before, with the RTI instruction.

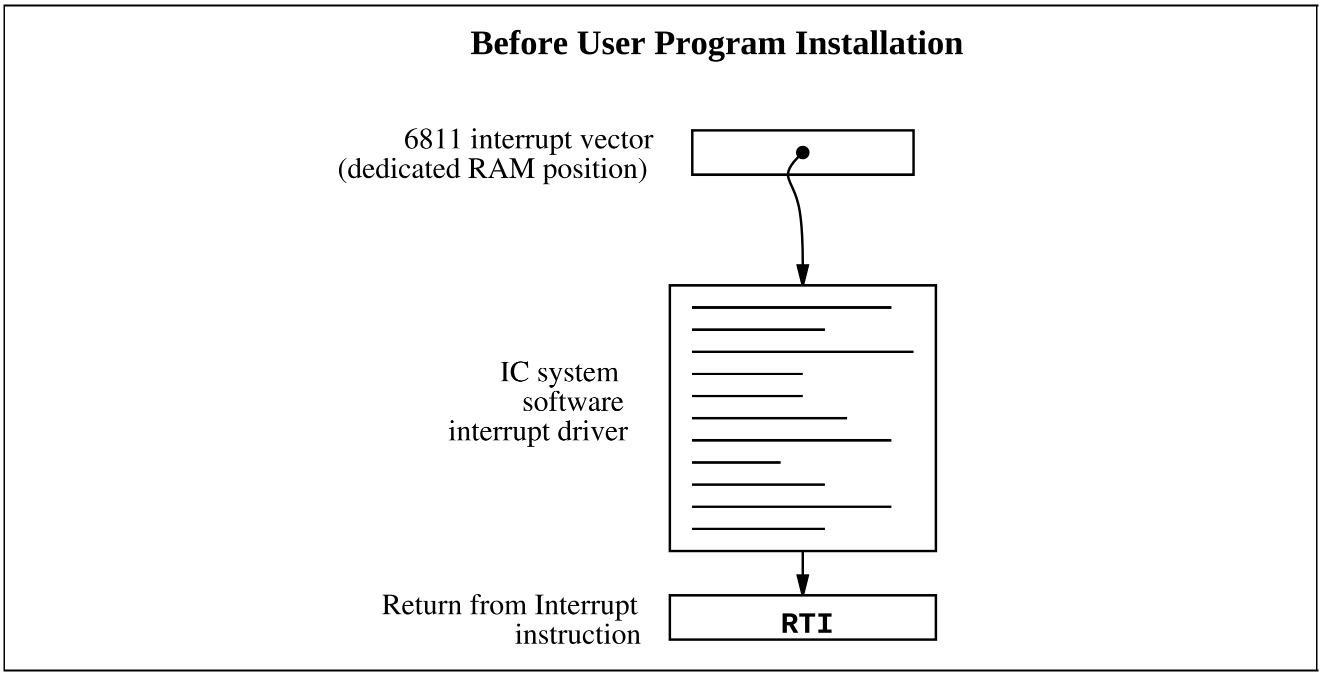


Figure 4: Interrupt Structure Before User Program Installation

Multiple user programs could be installed in this fashion. Each one would install itself ahead of the previous one. Some standard library functions, such as the shaft encoder software, is implemented in this fashion.

Figure 6 shows an example program that installs itself into the System Interrupt. This program toggles the signal line controlling the piezo beeper every time it is run; since the System Interrupt runs at 1000 Hz., this program will create a continous tone of 500 Hz.

The first line after the comment header includes a file named "`6811regs.asm`". This file contains equates for all 6811 registers and interrupt vectors; most binary programs will need at least a few of these. It is simplest to keep them all in one file that can be easily included.

The `subroutine_initialize_module` declaration begins the initialization portion of the program. The file "`ldxibase.asm`" is then included. This file contains a few lines of 6811 assembler code that perform the function of determining the base pointer to the 6811 interrupt vector area, and loading this pointer into the 6811 X register.

The following four lines of code install the interrupt program (beginning with the label `interrupt_code_start`) according to the method that was illustrated in Figure 5.

First, the existing interrupt pointer is fetched. As indicated by the comment, the 6811's TOC4 timer is used to implement the System Interrupt. The vector is poked into the JMP instruction that will conclude the interrupt code itself.

Next, the 6811 interrupt pointer is replaced with a pointer to the new code. These two steps complete the initialization sequence.

The actual interrupt code is quite short. It toggles bit 3 of the 6811's PORTA register. The PORTA register controls the eight pins of Port A that connect to external hardware; bit 3 is connected to the piezo beeper.

The interrupt code exits with a jump instruction. The argument for this jump is poked in by the

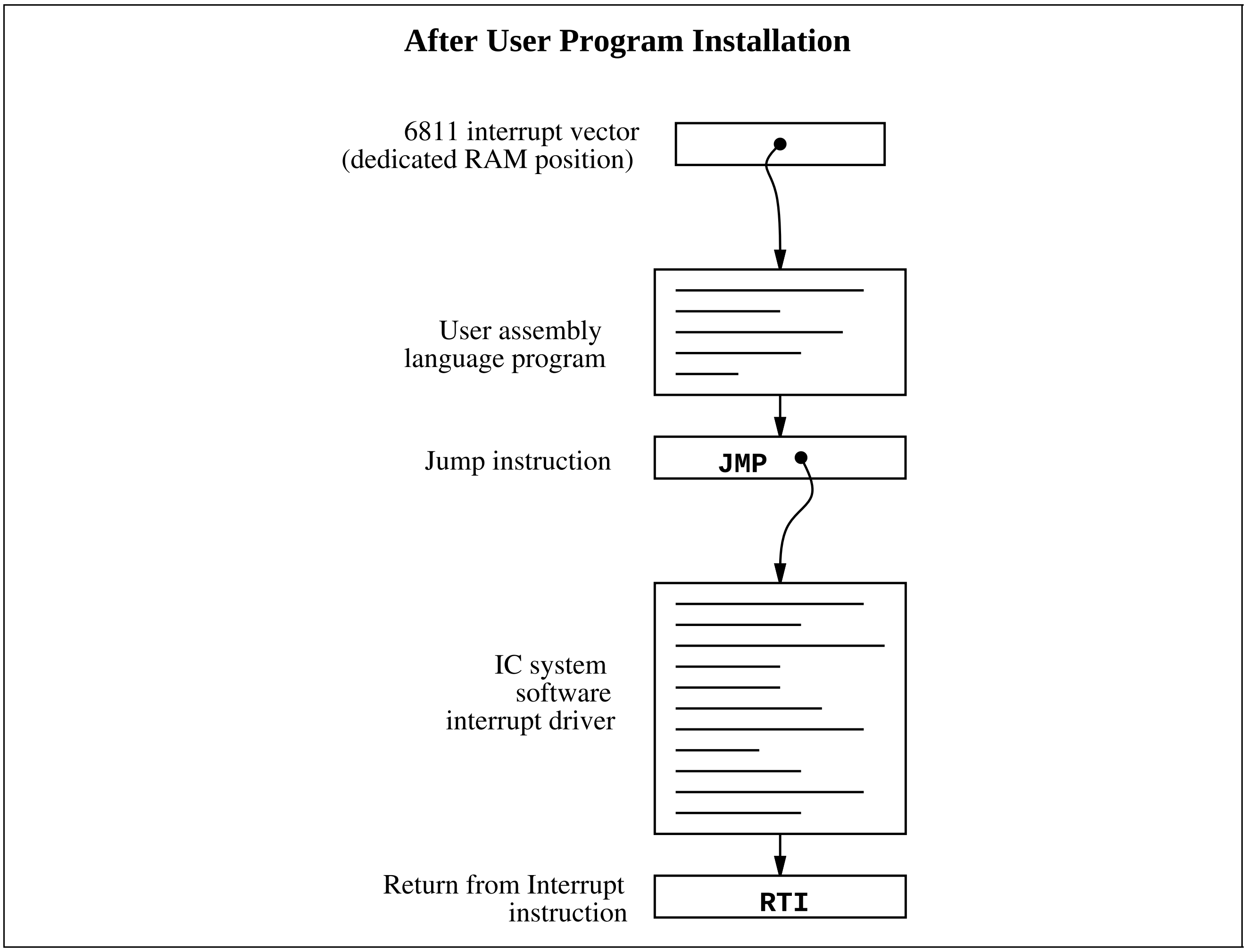


Figure 5: Interrupt Structure After User Program Installation

```
* icb file:  "sysibeep.asm"

*
*   example of code installing itself into
*   SystemInt 1000 Hz interrupt
*
*   Fred Martin
*   Thu Oct 10 21:12:13 1991
*

#include <6811regs.asm>

        ORG     MAIN_START

subroutine_initialize_module:

#include <ldxibase.asm>
* X now has base pointer to interrupt vectors ($FF00 or $BF00)

* get current vector; poke such that when we finish, we go there
        LDD     TOC4INT,X           ; SystemInt on TOC4
        STD     interrupt_code_exit+1

* install ourself as new vector
        LDD     #interrupt_code_start
        STD     TOC4INT,X

        RTS

* interrupt program begins here
interrupt_code_start:
* frob the beeper every time called
        LDAA    PORTA
        EORA    #%00001000     ; beeper bit
        STAA    PORTA

interrupt_code_exit:
        JMP     $0000    ; this value poked in by init routine
```

Figure 6: `sysibeep.asm`: Binary Program that Installs into System Interrupt

initialization program.

The method allows any number of programs located in separate files to attach themselves to the System Interrupt. Because these files can be loaded from the C environment, this system affords maximal flexibility to the user, with small overhead in terms of code efficiency.

5.12.3 The Binary Object File

The source file for a binary program must be named with the `.asm` suffix. Once the `.asm` file is created, a special version of the 6811 assembler program is used to construct the binary object code. This program creates a file containing the assembled machine code plus label definitions of entry points and C variables.

```
S116802005390037FD802239FC802239CC0045FD8022393C
S9030000FC
S116872B05390037FD872D39FC872D39CC0045FD872D39F4
S9030000FC
6811 assembler version 2.1  10-Aug-91
  please send bugs to Randy Sargent (rsargent@athena.mit.edu)
  original program by Motorola.
subroutine_double 872b *0007
subroutine_get_foo 8733 *0021
subroutine_initialize_module 8737 *0026
subroutine_set_foo 872f *0016
variable_foo 872d *0012 0017 0022 0028
```

Figure 7: Sample IC Binary Object File: `testicb.icb`

The program `as11_ic` is used to assemble the source code and create a binary object file. It is given the filename of the source file as an argument. The resulting object file is automatically given the suffix `.icb` (for IC Binary). Figure 7 shows the binary object file that is created from the `testicb.asm` example file.

5.12.4 Loading an ICB File

Once the `.icb` file is created, it can be loaded into IC just like any other C file. If there are C functions that are to be used in conjunction with the binary programs, it is customary to put them into a file with the same name as the `.icb` file, and then use a `.lis` file to loads the two files together.

5.12.5 Passing Array Pointers to a Binary Program

A pointer to an array is a 16-bit integer address. To coerce an array pointer to an integer, use the following form:

```
array_ptr= (int) array;
```

where `array_ptr` is an integer and `array` is an array.

When compiling code that performs this type of pointer conversion, IC must be used in a special mode. Normally, IC does not allow certain types of pointer manipulation that may crash the system. To compile this type of code, use the following invokation:

```
ic -wizard
```

Arrays are internally represented with a two-byte length value followed by the array contents.

5.13 IC File Formats and Management

This section explains how IC deals with multiple source files.

5.13.1 C Programs

All files containing C code must be named with the "`.c`" suffix.

Loading functions from more than one C file can be done by issuing commands at the IC prompt to load each of the files. For example, to load the C files named `foo.c` and `bar.c`:

```
C> load foo.c
C> load bar.c
```

Alternatively, the files could be loaded with a single command:

```
C> load foo.c bar.c
```

If the files to be loaded contain dependencies (for example, if one file has a function that references a variable or function defined in the other file), then the second method (multiple file names to one load command) or the following approach must be used.

5.13.2 List Files

If the program is separated into multiple files that are always loaded together, a "list file" may be created. This file tells IC to load a set of named files. Continuing the previous example, a file called `gnu.lis` can be created:

Listing of `gnu.lis`:

```
foo.c
bar.c
```

Then typing the command `load gnu.lis` from the C prompt would cause both `foo.c` and `bar.c` to be loaded.

5.13.3 File and Function Management

Unloading Files When files are loaded into IC, they stay loaded until they are explicitly unloaded. This is usually the functionality that is desired. If one of the program files is being worked on, the other ones will remain in memory so that they don't have to be explicitly re-loaded each time the one undergoing development is reloaded.

However, suppose the file `foo.c` is loaded, which contains a definition for the function `main`. Then the file `bar.c` is loaded, which happens to also contain a definition for `main`. There will be an error message, because both files contain a `main`. IC will unload `bar.c`, due to the error, and re-download `foo.c` and any other files that are presently loaded.

The solution is to first unload the file containing the `main` that is not desired, and then load the file that contains the new `main`:

```
C> unload foo.c
C> load bar.c
```

5.14 Configuring IC

IC has a multitude of command-line switches that allow control of a number of things. With command-line versions of IC, explanations for these switches can be gotten by issuing the command "`ic -help`".

IC stores the search path for and name of the library files internally; theses may be changed by executing the command "`ic -config`". When this command is run, IC will prompt for a new path and library file name, and will create a new executable copy of itself with these changes.

The Macintosh version of IC is configured by changing the values of `STR` resources using a utility like ResEdit.

6 Sensors and Motors

This section explains how to interface a variety of devices to the Handy Board:

- A DC motor.
- A microswitch touch sensor.
- A photocell-based light sensor.
- An infrared reflectance sensor.

First, proper connector wiring technique, applicable to all devices is explained. Then individual wiring diagrams for each of the devices are presented.

6.1 Connector Wiring Technique

Connectors are the bane of existence of all electronics. If there is one weak link in the reliable performance of any electronic system, it is its connectors. With this in mind, the importance of patiently and neatly built robot connectors cannot be overemphasized. Particularly since a robot is a mobile system subjected to various jolts and shocks, care taken in the construction of the robot's connectors will always pay off in the long run.

The Handy Board uses 0.1 inch male header as its connector for both motors and sensors. These are not the easiest connectors to work with, but they have a very compact footprint, allowing a large number of devices to be individually connected to the Handy Board.

The technique presented here has been time-tested to yield reliable results. There are four basic steps in the process:

1. Stripping and tinning wire ends.
2. Inserting heat shrink tubing on the individual wires.
3. Soldering wire ends to male header connector.
4. Shrinking tubing around the joints.

The remainder of this section explains the technique, showing diagrams for building the standard DC motor connector.

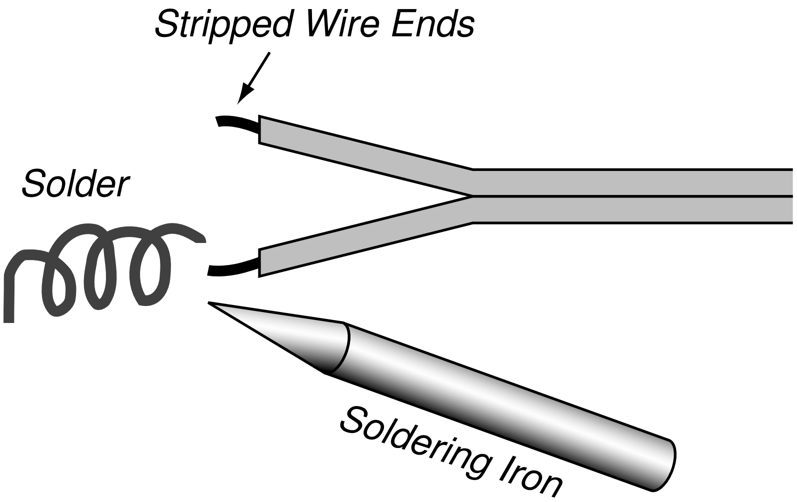
6.1.1 Wire Type

It is important to use stranded, not solid, wire cable. Each length of stranded wire consists of a twisted bundle of very thin thread-like wires. Solid wire, on the other hand, is a single thick wire segment.

The advantage of stranded wire is that it is much more flexible than solid wire, and also less susceptible to breakage. One thread of a stranded wire lengths can break without affecting the performance of the connection, but if a solid wire breaks the connection is lost.

An ideal wire for building sensor and motor cables is 28 gauge ribbon cable. Ribbon cable is stranded; the 28 gauge is the right weight to carry the current required to drive motors while still providing excellent flexibility. Ribbon cable also "zips" easily, so that sets of two or three wires can easily be made. Finally, rainbow ribbon cable is brightly colored in a ten-color sequence, making it easy to keep track of which wire connects where.

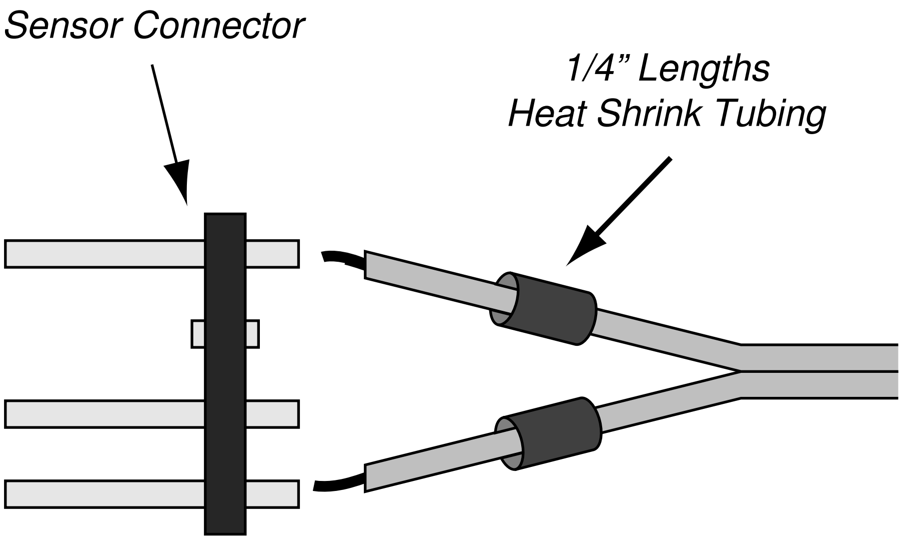
6.1.2 Stripping and Tinning Wire Ends



The first step is to strip insulation from the wire cable and *tin* the wire ends. "Tinning" is the process of infusing the stranded wire end with solder.

Remove between 1/8 and 1/4 inch of insulation from the end of each wire. With your fingertips, individually twist the threads of each wire end tightly (follow the existing weave of the stranded wire bundle). Then, put a dab of solder onto the soldering iron, hold it to the wire end, and add some solder to the wire end. Draw the iron tip along the wire end to evenly distribute solder into the wire end.

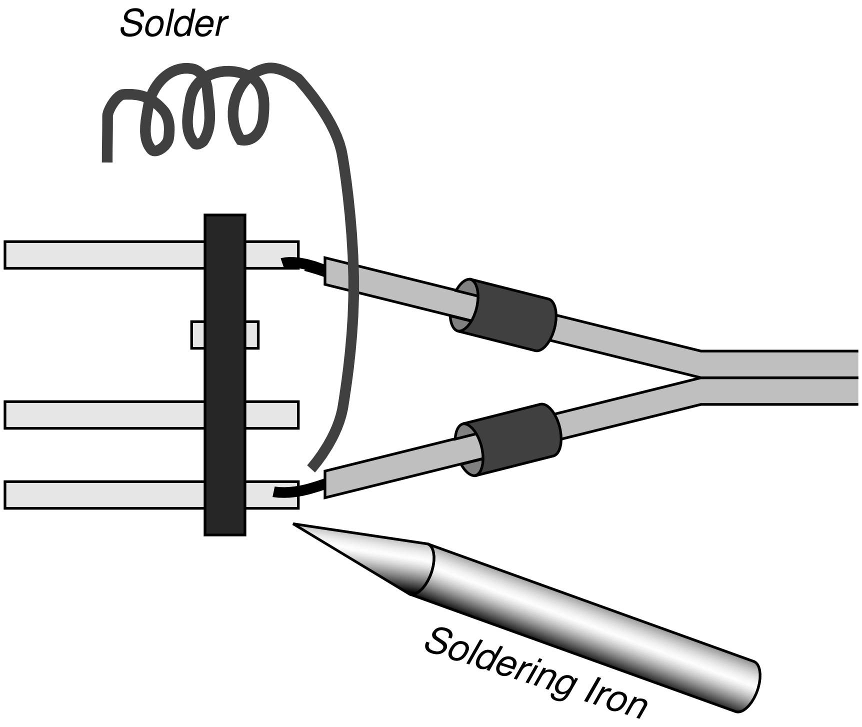
6.1.3 Installing Heat Shrink Tubing



Cut a 1/4 inch length of heat shrink tubing for each connection, and feed a tubing segment onto each wire.

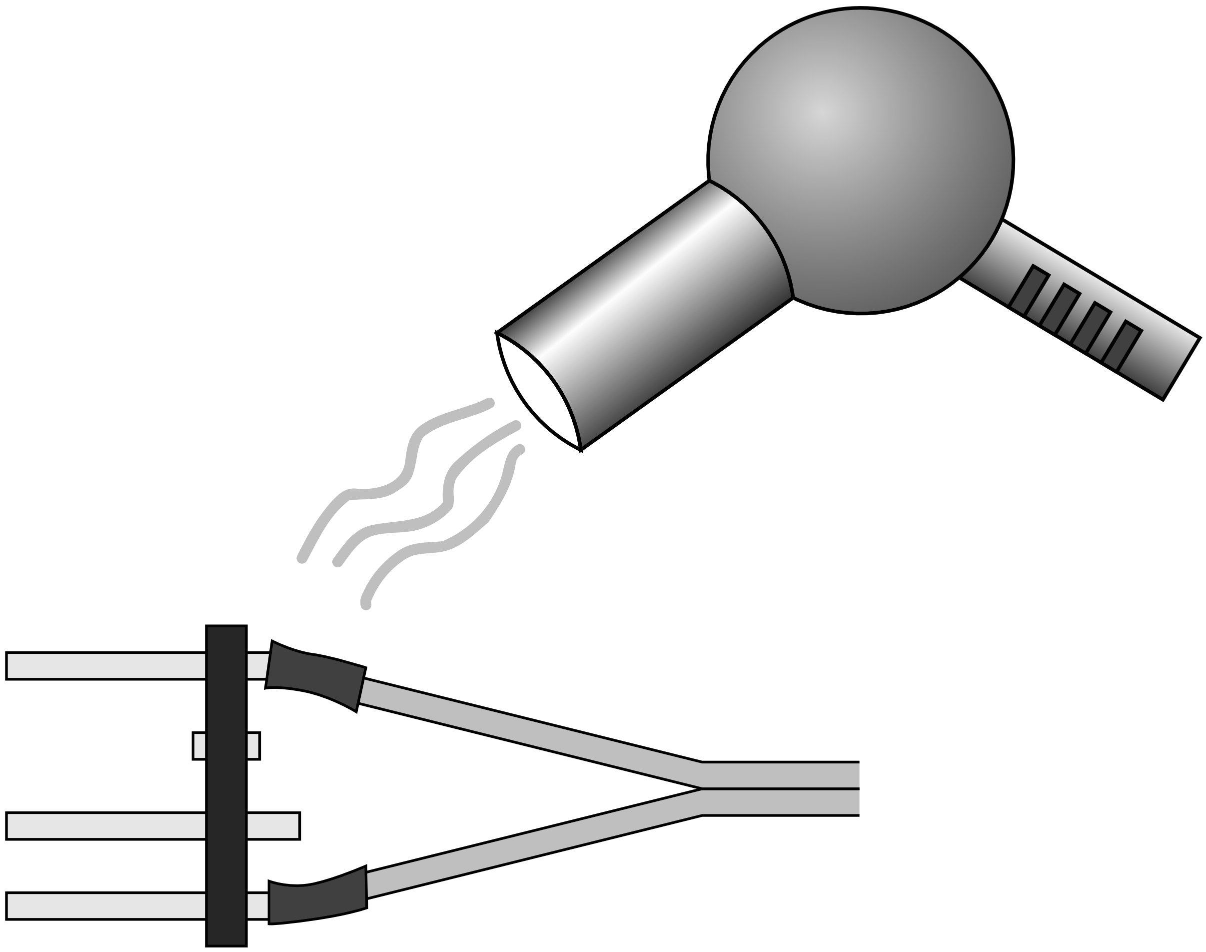
In preparation for soldering, align the wires with the male header pins as indicated in the diagram. If necessary, zip back the individual wires so that the tubing does not get in the way of the connection. (The use of a "helping hands" tool is helpful here—a tool with two alligator clips on flexible arms.)

6.1.4 Soldering to Male Header



Line up the wire ends with the male header pins and solder. Make sure that the heat shrink tubing is far enough away from the joint that the tubing does not shrink prematurely.

6.1.5 Shrinking the Tubing

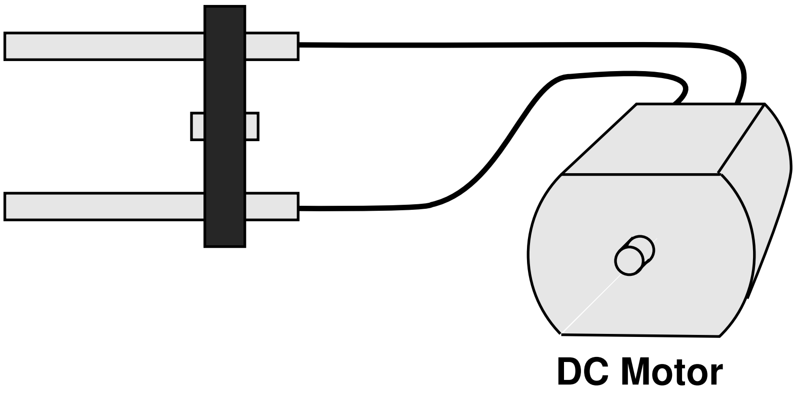


Gently apply heat from heat gun to shrink the tubing over the joints.

Slide the heat shrink tubing over the joints, and apply heat from a heat gun. If a heat gun is not available, the open flame from a match or butane lighter may be used. Hold the joint so the heat shrink tubing is at least 1 inch above the tip of the flame.

That's it! The connector end that plugs into the Handy Board is now complete.

6.2 Motors

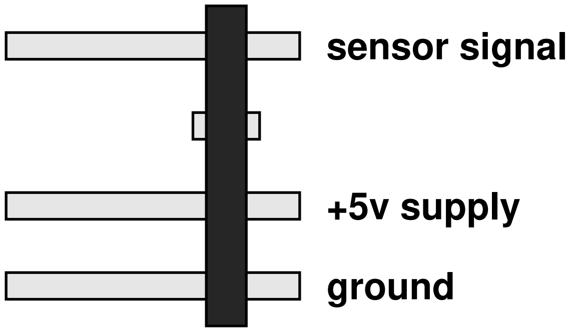


The DC motor connector uses two male pins on 0.2 inch spacing; i.e., the outer two of three pins. The center pin can be clipped away from the assembly.

Motors used with the Handy Board should be rated for 9 volt operation with a maximum current draw of about 600 mA.

6.3 Sensors

6.3.1 Basic Sensor Connector



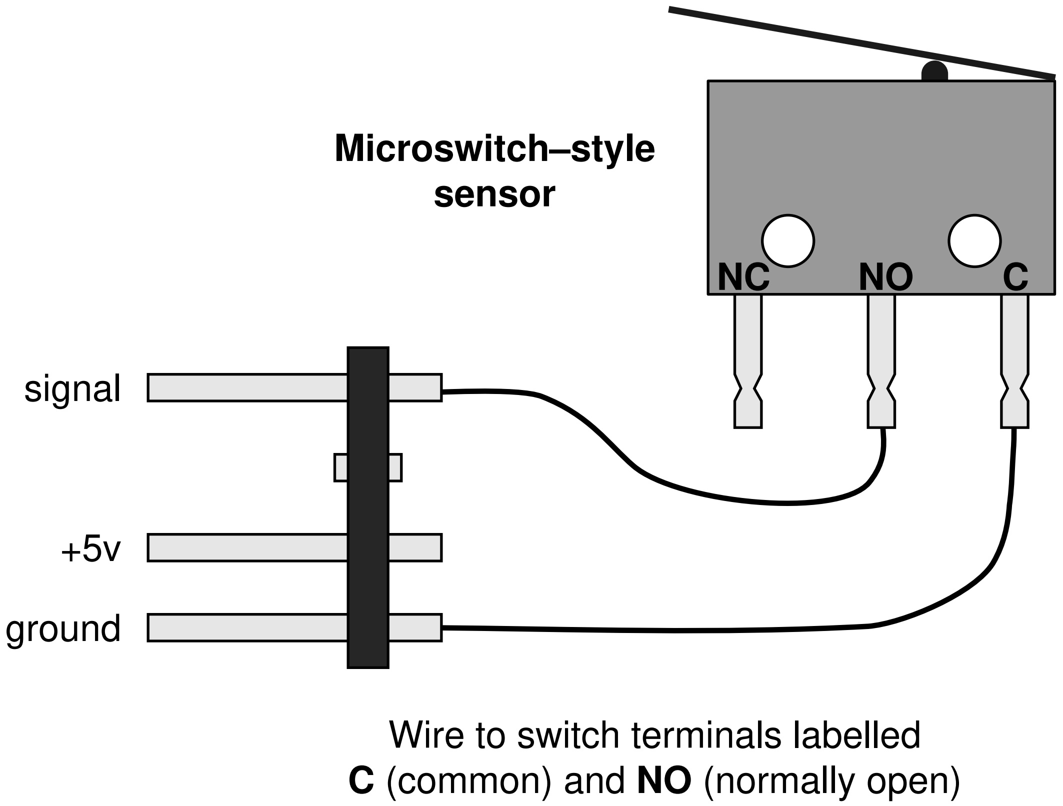
The Handy Board uses a three-conductor connector for plugging in sensor devices. As indicated in the diagram, the connector is formed from 4–prong male header pins, with one pin clipped away to polarize the connector (i.e., prevent it from being plugged in improperly).

The pin labelled "+5v supply" may be used to power an active sensor (e.g., the transmitter LED of a reflective optosensor). The pin labelled "sensor signal" is the input to the Handy Board circuitry; this must be in the range of 0 to 5 volts. The pin labelled "ground" is the system ground.

The Handy Board includes a 47K pullup resistor that is wired between the sensor signal line and the +5v supply on all of its inputs, both analog and digital. This simplifies sensor design in several regards:

- All sensors have a default level of +5v when nothing is plugged in.
- For switch-type or resistive-type sensors, the sensor device just needs to be wired from the sensor signal pin to ground. Thus many sensor devices reduce to a simple two-wire connection.

6.3.2 Switch Sensor

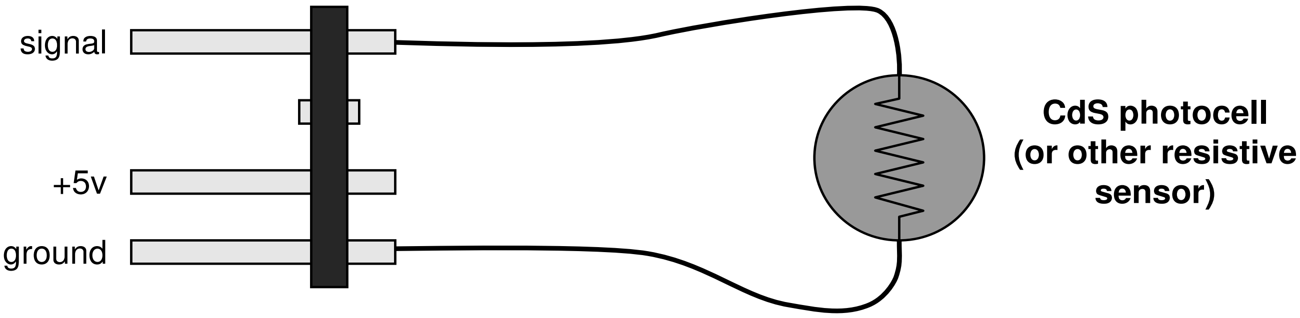


The above diagram shows how to wire a microswitch-style sensor to the Handy Board. As indicated in the diagram, the switch terminals labelled "C" (common) and "NO" (normally open) should be connected to the sensor plug.

This wiring creates a switch sensor that is normally open, or disconnected, except when the switch is pressed. The normally open case means that the sensor line is pulled high by the 47K resistor on the Handy Board. The standard software for reading the state of a switch interprets this logic high value as "not pressed" or false. When the switch is closed, the sensor line is connected to ground, and the software reads a logic low value, which is interpreted as "pressed" or true.

A pushbutton-style switch, or any simple switch, may be wired in the same fashion.

6.3.3 Photocell Sensor

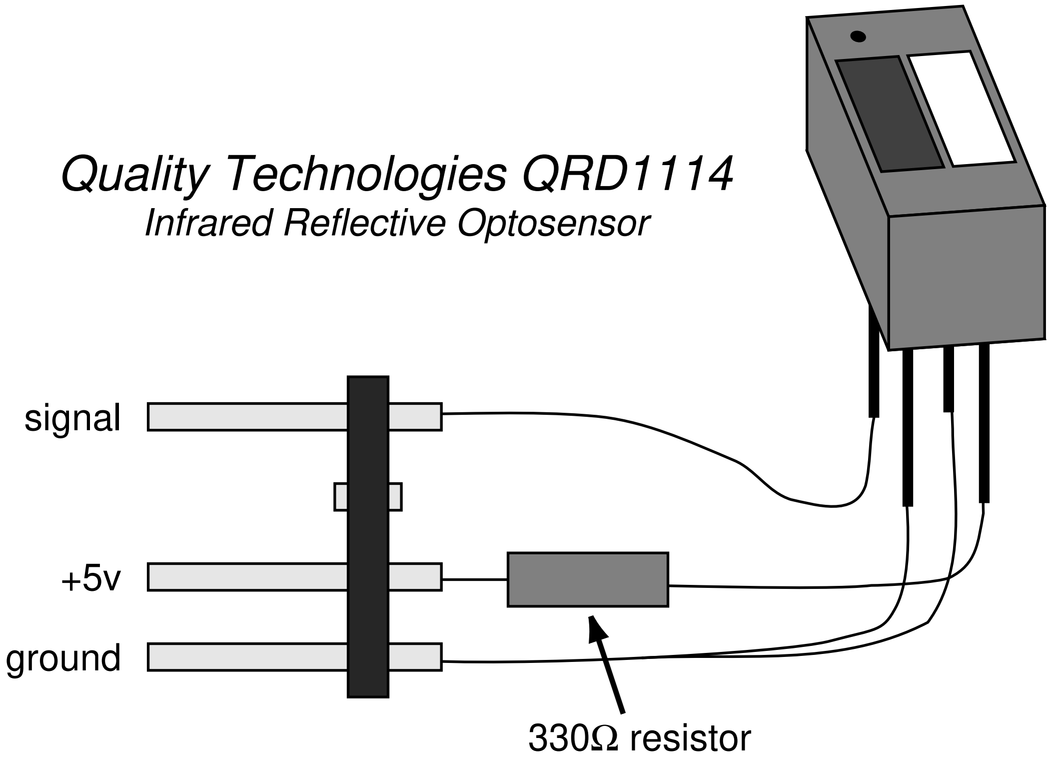


The photocell sensor wiring also makes use of the on-board 47K resistor that connects the sensor signal line to +5v. When wired from the signal line to ground, the photocell becomes part of a voltage divider circuit as indicated in the schematic to the right. The output voltage *Vout* in the circuit is the sensor signal line.

Vout varies as to the ratio between the two resistances (the fixed 47K resistance and the varying *Rphoto* resistance. When the photocell resistance is small (as when brightly illuminated), the *Vout* signal is close to zero volts; when the photocell resistance is large (as in the dark), *Vout* is close to $+5$ volts, with a continuously varying range between the extremes.

This means that the sensor will report small values when brightly illuminated and large values in the dark.

6.3.4 Infrared Reflectance Sensor



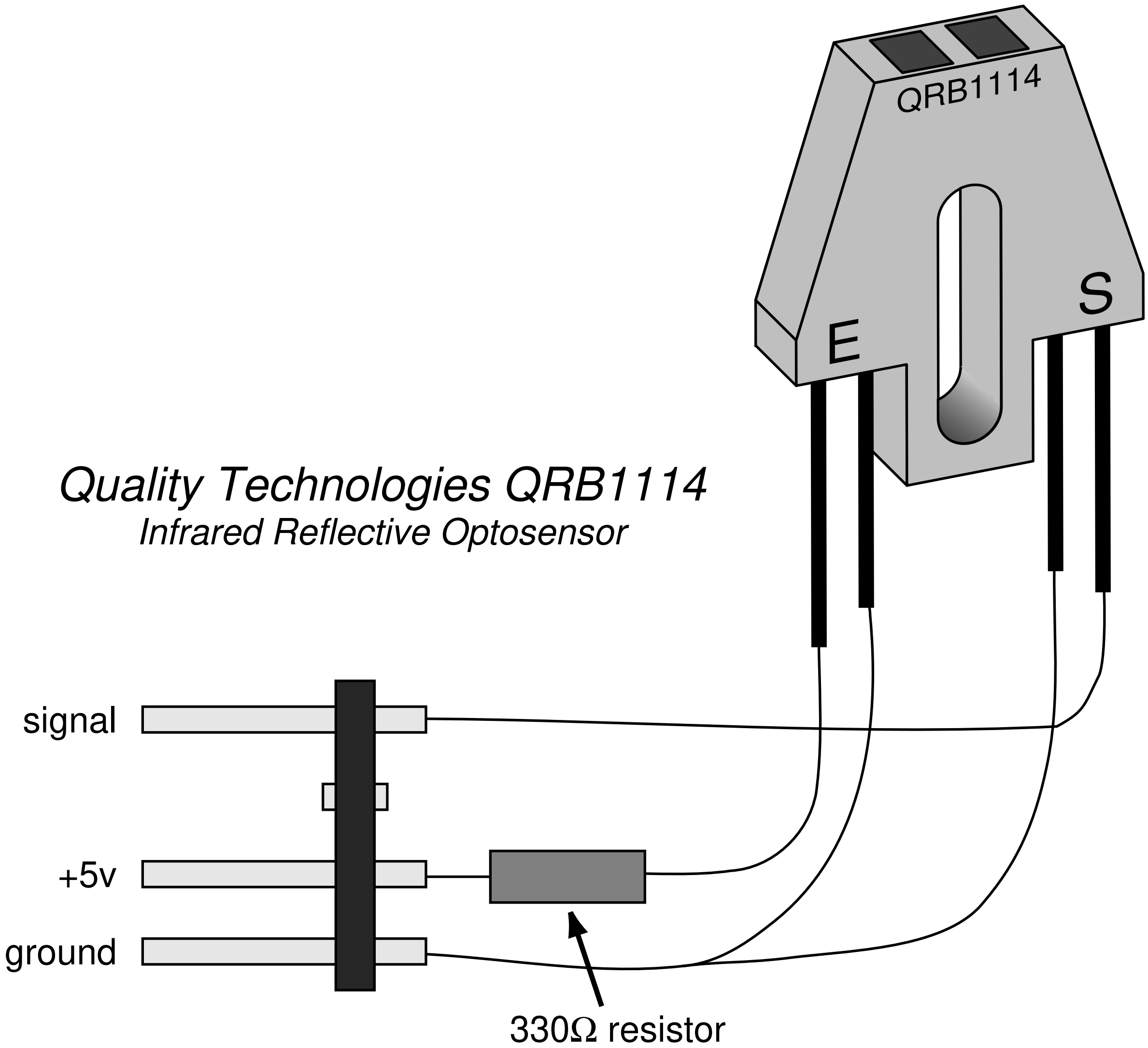
The infrared reflectance sensor consists of two discrete devices: an infrared LED emitter and an infrared phototransistor receiver. The receiver and emitter are matched, so that the peak sensitivity of the receiver is at the same wavelength of the emissions of the emitter. In the example **Quality Technologies QRD1114** sensor diagram, the detector LED is on the left and the emitter is on the right.

The wiring for the reflectance sensor is straightforward. The emitter LED is powered by the Handy Board's $+5$v supply, with a 330 ohm resistor in series to limit the current through the LED to an appropriate value. The detector transistor is pulled high with the Handy Board's internal 47K resistor.

When increasing amounts of light from the emitter LED is reflected back into the receiver, increasing amounts of current flow through the receiver transistor and hence the internal 47K resistor. The voltage drop across this resistor results in a lower voltage presented to the Handy Board's analog input.

Different varieties of phototransistor may perform better with a smaller resistor value than the on-board 47K resistor. If the sensitivity of the device is poor, try connecting the signal line to the $+5$v supply through 10K, 4.7K, or 2.2K resistors to determine the best response. For the QRD1114 device, however, the default 47K value is ideal.

> *Special note for working with infrared light:* Infrared light is indeed invisible (unless you are a bumblebee), making it hard to ascertain that a given infrared emitter LED is indeed working. Here are two methods that may be used to visualize its presence: (1) Look at the IR LED through a video-camera that has a viewfinder CRT screen. The CCD lens of a standard video-camera is sensitive to infrared light, and it will be visible on its display. (2) Purchase an infrared detector card (Radio Shack 276–099 or MCM 72–003 and 72–005), which contains a phosphorescent panel that glows visibly under infrared illumination.



The **Quality Technologies QRB1114** sensor, above, is another good reflective optosensor. In the diagram, the left-hand component, marked "E" on the device package, is the infrared emitter, and the right-hand component, marked "S," is the infrared sensor.

7 Battery Maintenance

The Handy Board has a 9.6v, 600 mA battery pack consisting of eight AA-cell nickel-cadmium rechargeable batteries.

7.1 Battery Charging

There are three ways to charge the internal battery:

1. *Adapter plugged directly into the HB.* Just plug the adapter into the power jack on the HB, and the yellow “CHARGE” LED on the HB will light. This is a trickle-charge mode, which means that (1) the Handy Board will fully charge in about 12 to 14 hours, and (2) the HB may be left in this mode indefinitely.

2. *Adapter plugged into the Serial Interface/Battery Charger board; HB connected via telephone wire; “NORMAL CHARGE” mode selected.* The yellow “CHARGE” LED on the interface board will light. This is a trickle-charge mode, which means that (1) the Handy Board will fully charge in about 12 to 14 hours, and (2) the HB may be left in this mode indefinitely.

3. *Adapter plugged into the Serial Interface/Battery Charger board; HB connected via telephone wire; “ZAP CHARGE” mode selected.* The yellow “CHARGE” LED on the interface board will *not* light. The ZAP CHARGE will fully charge the HB’s battery in just 3 hours, *after which time the battery will become warm and it should be removed from charge or placed into either of the two trickle-charge modes*.

When using one of the trickle-charge modes, the Handy Board itself should be turned off so that the charge current goes toward charging the battery and not simply running the board. In Zap charge, there is enough charge current to operate the board and charge the batteries at the same time (assuming that the board is not driving motors or other external loads).

7.2 Adapter Specifications

The specifications of the Handy Board’s DC adapter are as follows:

- 12 volt, 500 mA DC output
- 2.1 mm inside, 5.5 mm outside diameter barrel-type plug
- center conductor negative

Most “universal” type adapters will work properly at one of their settings. Look for the yellow charge LED to light up indicating proper charge (make sure the Charge Rate switch is set to “Normal” mode).

Please be careful not to get an adapter that is *overpowered*. Problems have been reported using adapters that are rated for 1 to 2 amps.

Also, do not use an adapter that is underpowered or undervoltage. A 9 volt adapter will appear to work—the charge LED will light—but it won’t be able to charge the battery for more than a few minutes’ worth of power.

8 Part Listing

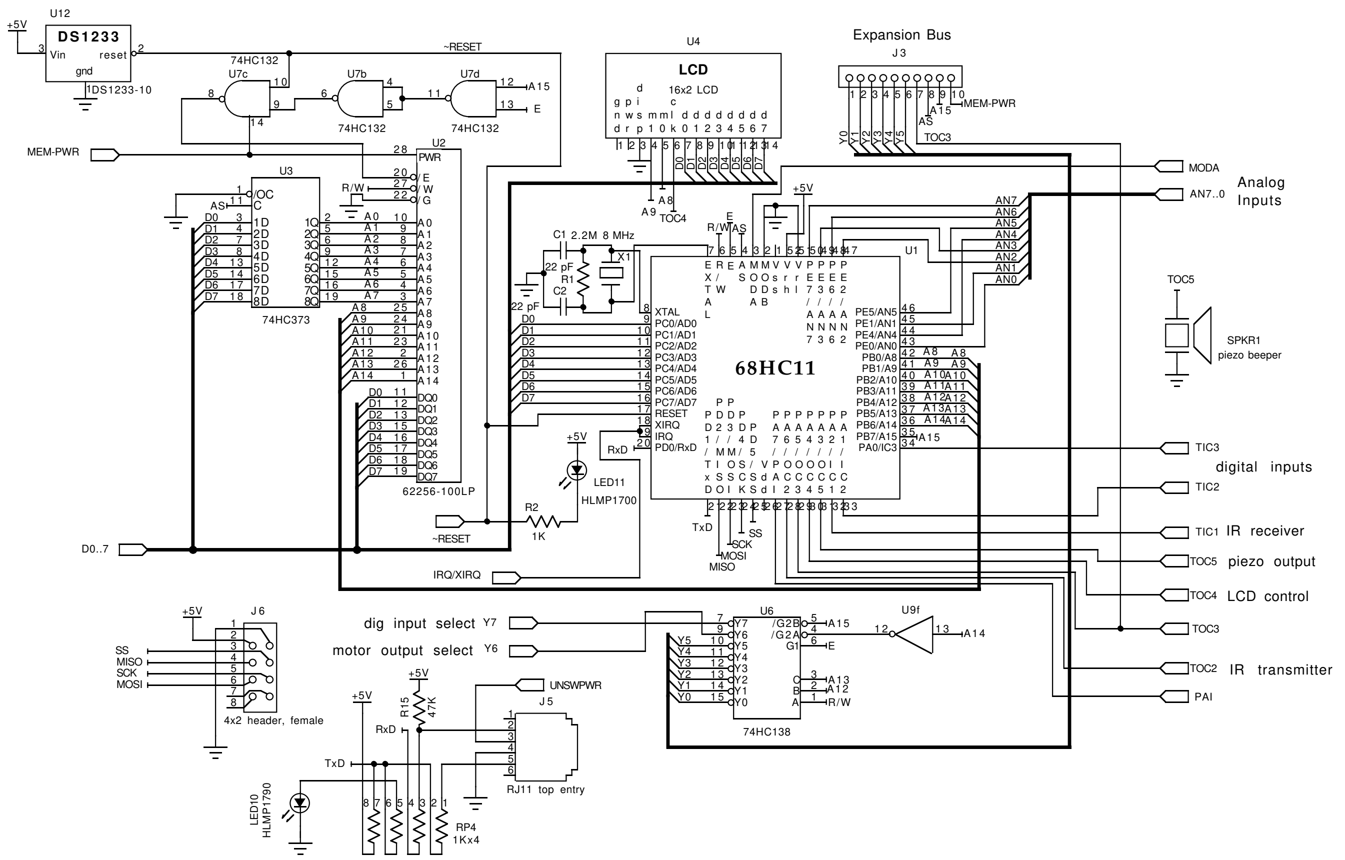
Circuit: hbsch12
Date: Thursday, November 30, 1995 - 9:58 AM

Device Type	Num.	Value	References	Price Ea.	Catalog No.	Supplier
8 cell AA nicad pack	1		BAT1	19.28	P227-L024-ND	Digikey
2% polyprop cap	1	0.0068 uF	C6	0.49	P3682-ND	Digikey
monolithic cer cap	4	0.1 uF	C5 C7 C9 C14	0.21	P4917-ND	Digikey
mini radial 'lytic	4	10uF	C10 C11 C12 C13	0.08	P6248-ND	Digikey
monolithic cer cap	2	22 pF	C1 C2	0.18	P4841-ND	Digikey
telephone cable	1	4-wire	CAB1	1.60	17MP007	Mouser
tantalum	2	4.7 uF	C4 C8	0.29	P2011-ND	Digikey
mini axial 'lytic	2	47 uF	C15 C16	0.29	P5972-ND	Digikey
mini axial 'lytic	1	470 uF	C3	0.65	P6305-ND	Digikey
power diode	1	1N4001	D3	0.15	333-1N4001	Mouser
signal diode	1	1N914	D1	0.15	333-1N914	Mouser
bridge rectifier	1	DB101	D2	0.62	DB101-ND	Digikey
AC or DC adapter	1	12v, 500mA	DC1	3.95	100087	Jameco
CPU board enclosure	1		ENCL1	5.12	537-402-RD	Mouser
interface enclosure	1		ENCL2	1.94	400-5043	Mouser
PolySwitch¤ fuse	1		F1	1.32	RUE250-ND	Digikey
Coax Power Jack	2	2.1mm ID	J11 J12	0.34	CP-002A-ND	Digikey
RJ11 top entry	1	6/4	J5	1.08	154-UL6642	Mouser
RJ12 side entry	1	6/6	J10	1.28	154-UL6661	Mouser
10-pin female header	1		J3			
12-pin female header	1		J4			
14-pin female header	1		J14			
14-pin male header	1		J15			
3 pcs 9-pin female hdr	1		J2	[FEMALE HEADER IS CUT		
3 pcs. 7-pin female hdr	1		J1	FROM 36-PIN HEADER		
3-pin female header	1		J7	LISTED AT END OF PAGE]		
4-pin header	2		J8 J13			
4x2 header, female	1		J6			
DB-25 female connector	1		J9	1.54	152-3425	Mouser
iron core inductor	1	1 uH	L1	0.84	M7010-ND	Digikey
high-eff red LED	7	HLMP1700	LED1 LED2 LED3 LED4 LED9 LED11 LED13	0.282	HLMP-1700QT-ND	Digikey
hi-eff yellow LED	2	HLMP1719	LED14 LED15	0.282	HLMP-1719QT-ND	Digikey
hi-eff green LED	6	HLMP1790	LED5 LED6 LED7 LED8 LED10 LED12	0.282	HLMP-1790QT-ND	Digikey
NPN darlington	1	ZTX614	Q1	0.59	ZTX614-ND	Digikey
	2	10K	R3 R7	0.0235	10KEBK-ND	Digikey
	3	1K	R2 R5 R10	0.0235	1KEBK-ND	Digikey
	1	2.2K	R9	0.0235	2.2KEBK-ND	Digikey
	1	2.2M	R1	0.0235	2.2MEBK-ND	Digikey
1% precision res	1	3.83K	R4	0.11	3.83KXBK-ND	Digikey
trimpot	1	20K	VR1	0.72	569-91AR-20K	Mouser
	3	47K	R6 R8 R15	0.0235	47KEBK-ND	Digikey
	1	47‰, 5W	R11	0.41	47W-5-ND	Digikey
	2	47‰	R12 R13	0.0235	47EBK-ND	Digikey
	2	47‰, 1/2W	R14	0.06	47H-ND	Digikey
	1	1Kx4	RP4	0.21	592-8A-1K	Mouser
RPACK6	1	1Kx5	RP2	0.16	592-6S-1K	Mouser
RPACK9	2	47Kx9	RP1 RP3	0.27	592-10S-47K	Mouser
14-pin DIP socket	2		DIP4 DIP5	0.57	ED3114-ND	Digikey
16-pin DIP socket	4		DIP6 DIP7 DIP8 DIP9	0.65	ED3116-ND	Digikey
20-pin DIP socket	2		DIP1 DIP2	0.81	ED3120-ND	Digikey
28-pin DIP socket	1		DIP3	1.13	ED3728-ND	Digikey
52-pin PLCC socket	1		PLCC	2.03	A2123-ND	Digikey
piezo beeper	1		SPKR1	1.90	P9957-ND	Digikey
SPDT slide switch	1		SW1	4.47	CKN5006-ND	Digikey
SPDT switch	1		SW4	1.10	SW101-ND	Digikey
pushbutton switch	2		SW2 SW3	0.20	P8006S-ND	Digikey
32K static CMOS RAM	1	62256-100LP	U2	3.95	42833	Jameco
hex inverters	1	74HC04	U9	0.29	570-CD74HC04E	Mouser
quad Schmitt NANDs	1	74HC132	U7	0.46	511-M74HC132	Mouser
3-to-8 decoder	1	74HC138	U6	0.46	570-CD74HC138E	Mouser
tristate bus driver	1	74HC244	U5	0.70	570-CD74HC244E	Mouser
transparent octal latch	1	74HC373	U3	0.68	570-CD74HC373E	Mouser
octal latch	1	74HC374	U8	0.61	570-CD74HC374E	Mouser
voltage monitor	1	DS1233-10	U12	1.25	manufacturer	Dallas Semi
infrared demodulator	1	IS1U60	U15	3.00	manufacturer	Sharp
motor driver	2	L293D	U10 U11	3.00	manufacturer	SGS-Thomson
voltage regulator	2	LM2931Z-5.0	U14 U17	0.90	LM2931Z-5.0-ND	Digikey
voltage regulator	1	LM7805CTB	U13	0.53	NJM7805FA-ND	Digikey
RS232 converter	1	MAX232CPE	U16	1.95	24811	Jameco
6811 microprocessor	1	MC68HC11A1FN	U1	8.00	manufacturer	Motorola
16x2 LCD	1	Hitachi	U4	8.00	LM052L	Timeline
microproc crystal	1	8 MHz	X1	2.32	332-5080	Mouser
female strip header	4			1.10	929974-01-36-ND	Digikey
male strip header	1			0.76	929834-01-36-ND	Digikey

9 Schematic Drawings

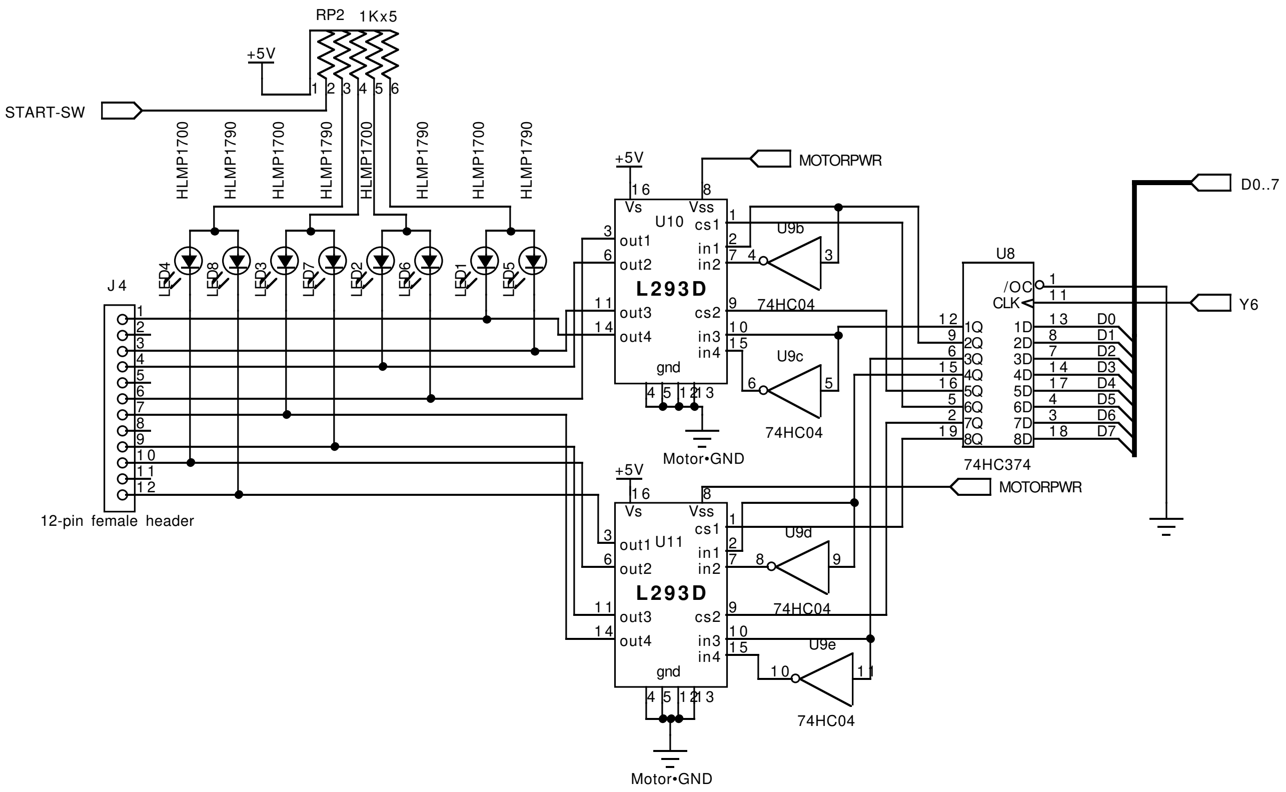
9.1 CPU and Memory

Handy Board version 1.2 CPU Board: CPU and Memory Circuit



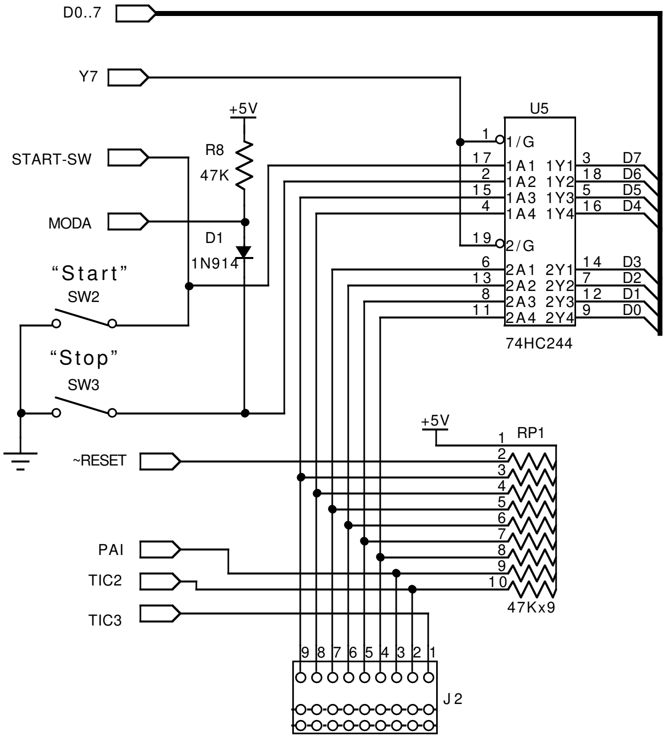
9.2 Motor Outputs

Handy Board version 1.2 CPU Board: Motor Output Circuit



9.3 Digital Inputs

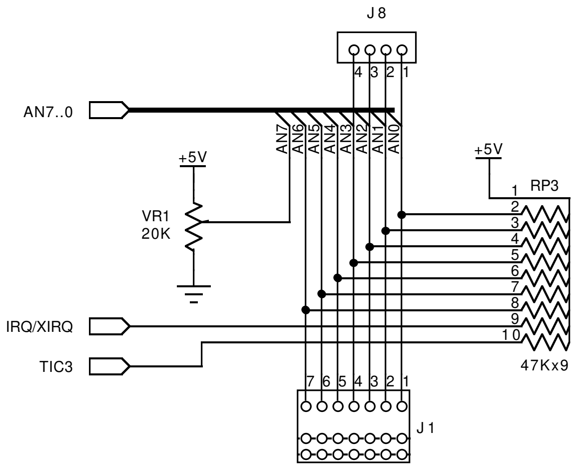
Handy Board version 1.2 CPU Board: Digital Input Circuit



Digital Inputs

9.4 Analog Inputs

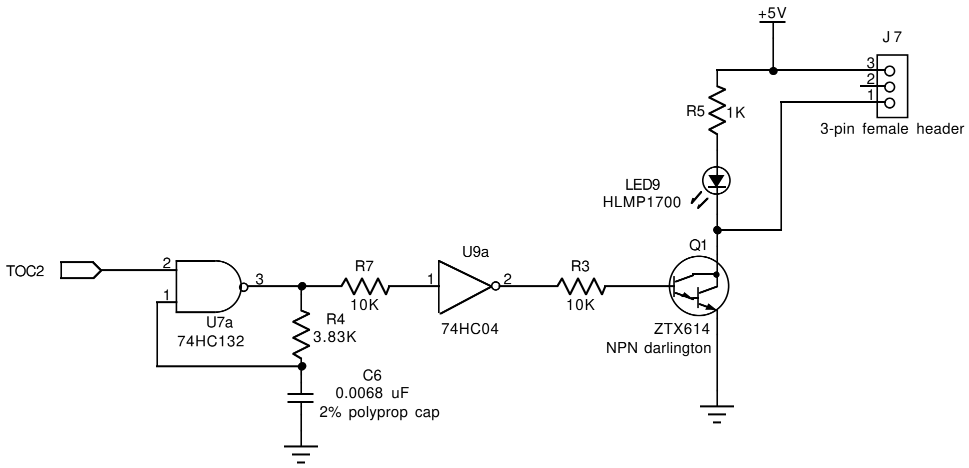
Handy Board version 1.2 CPU Board: Analog Input Circuit



Analog Inputs

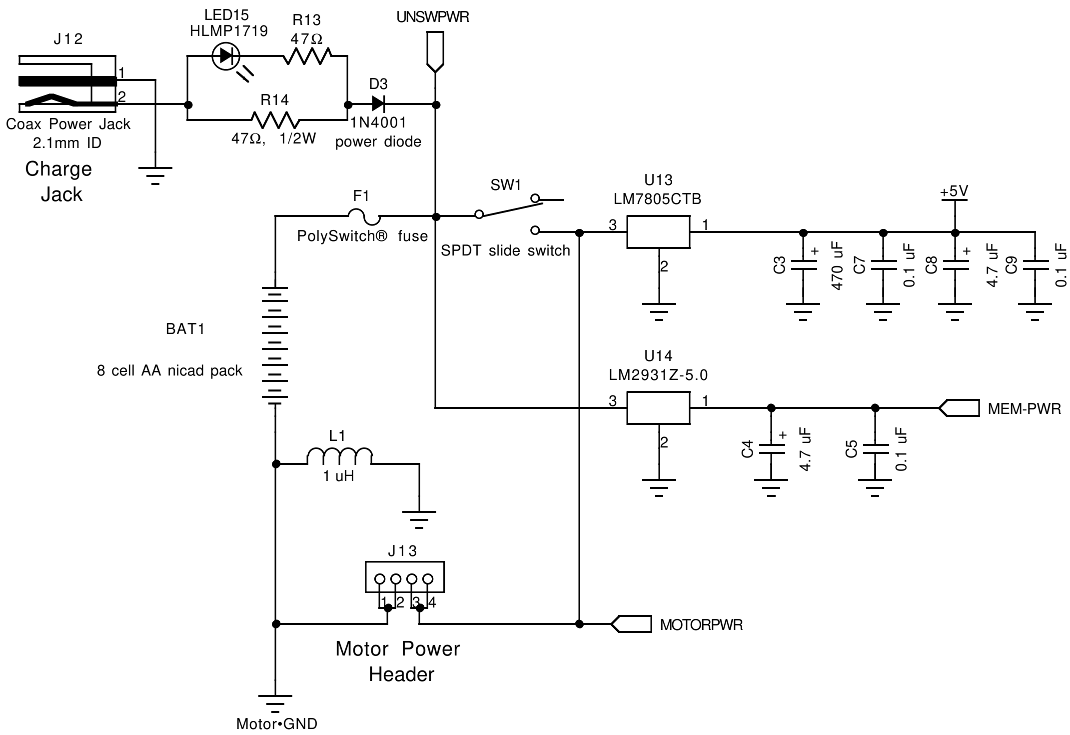
9.5 Infrared Transmission

Handy Board version 1.2 CPU Board: Infrared Transmission Circuit



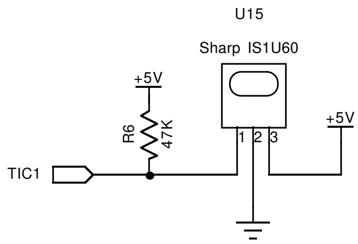
9.6 Power Supply

Handy Board version 1.2 CPU Board: Power Supply Circuit



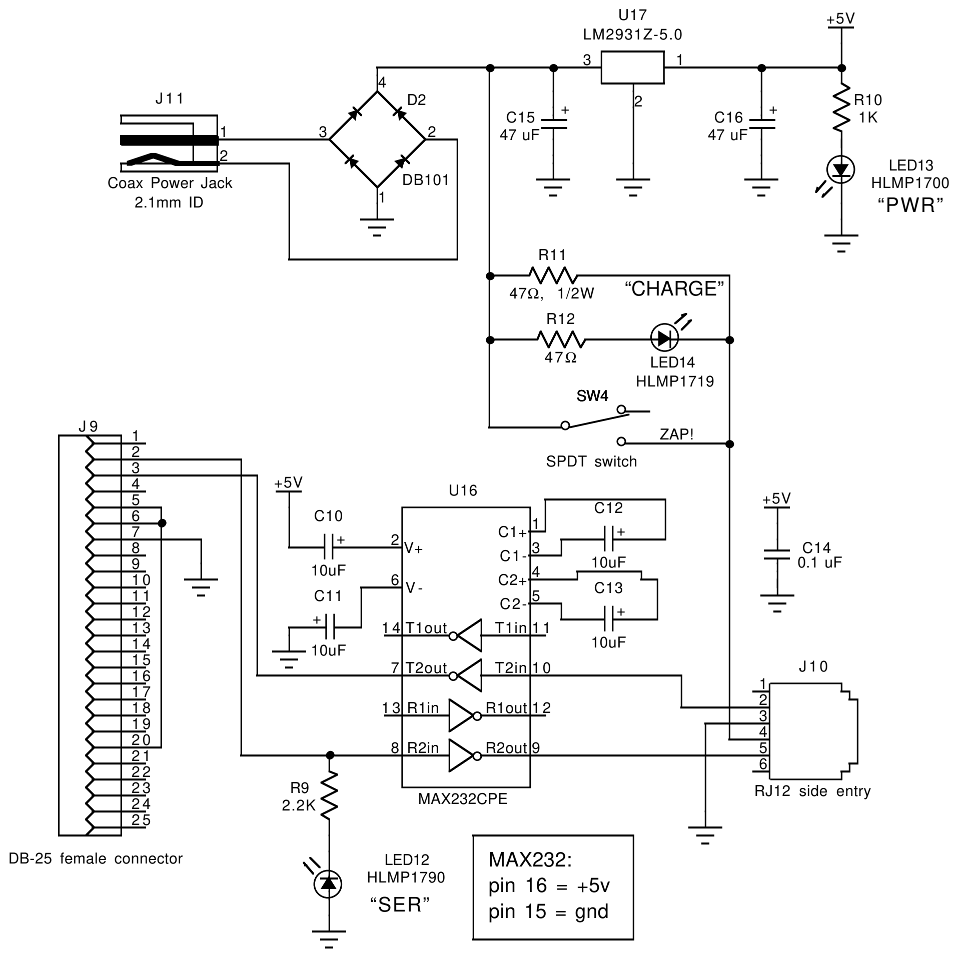
9.7 Infrared Reception

Handy Board version 1.2 CPU Board: Infrared Input Circuit



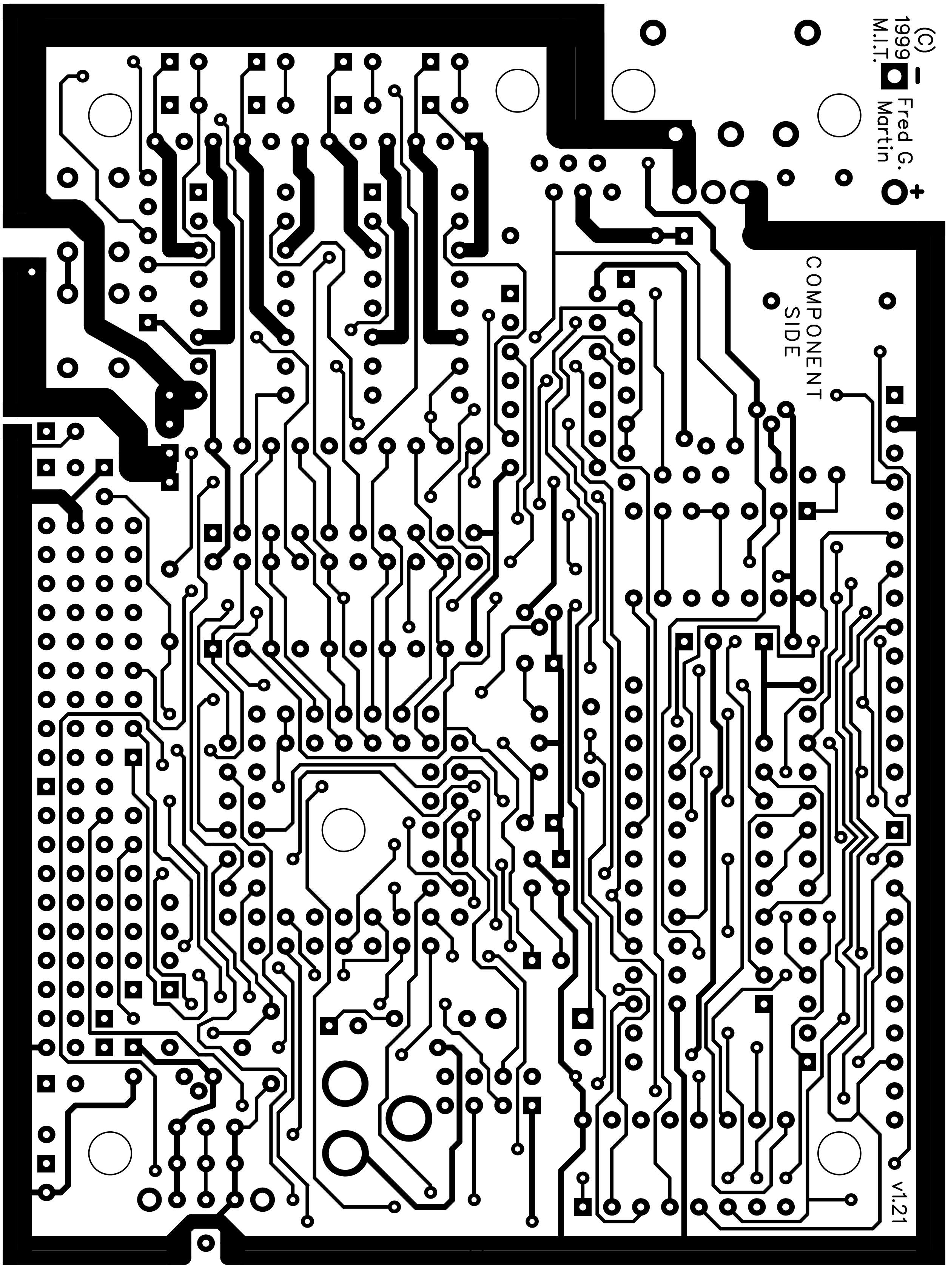
9.8 Serial Interface and Battery Charger

Handy Board Interface/Charger Unit, version 1.0

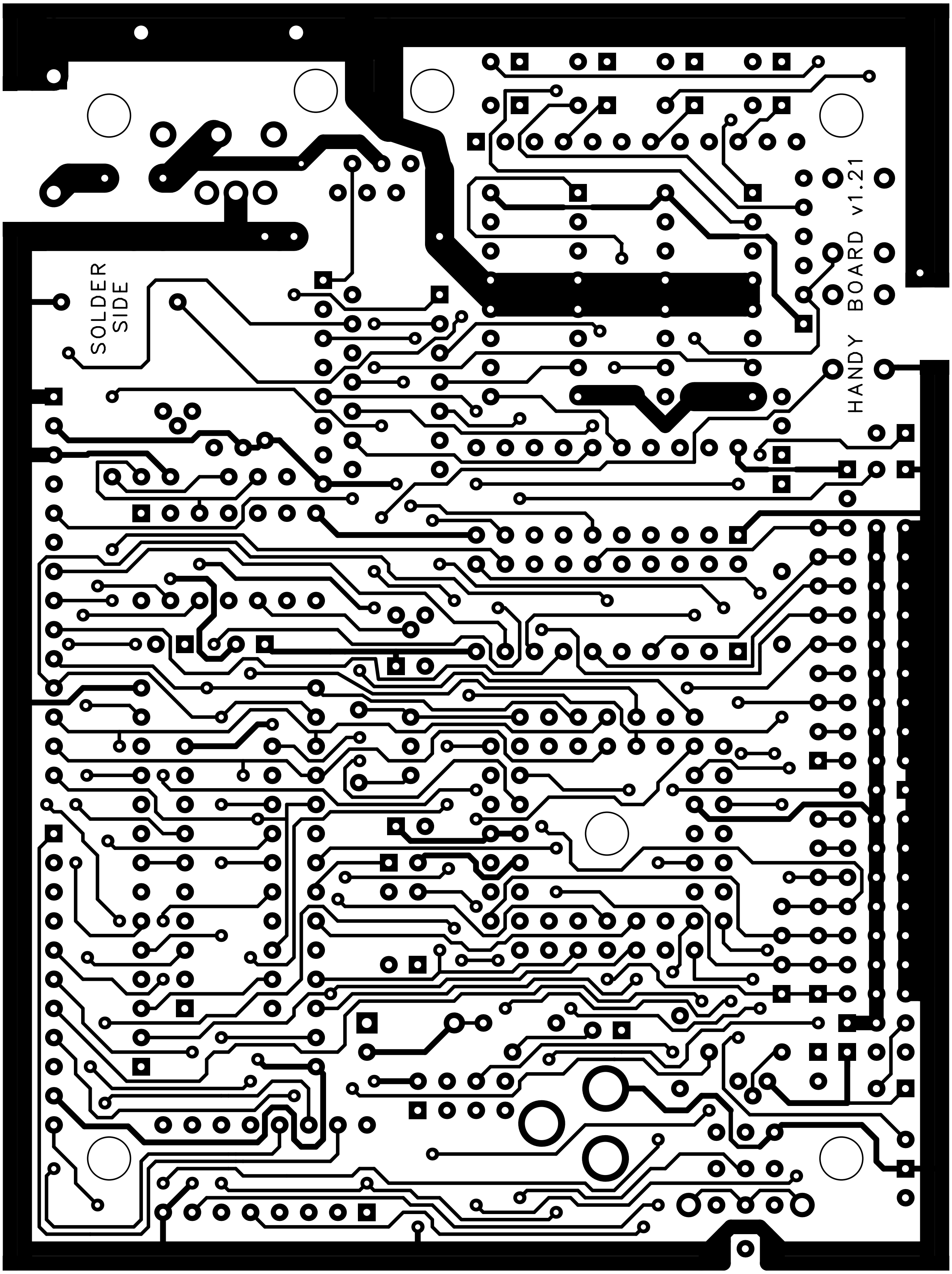


10 Printed Circuit Board Layouts

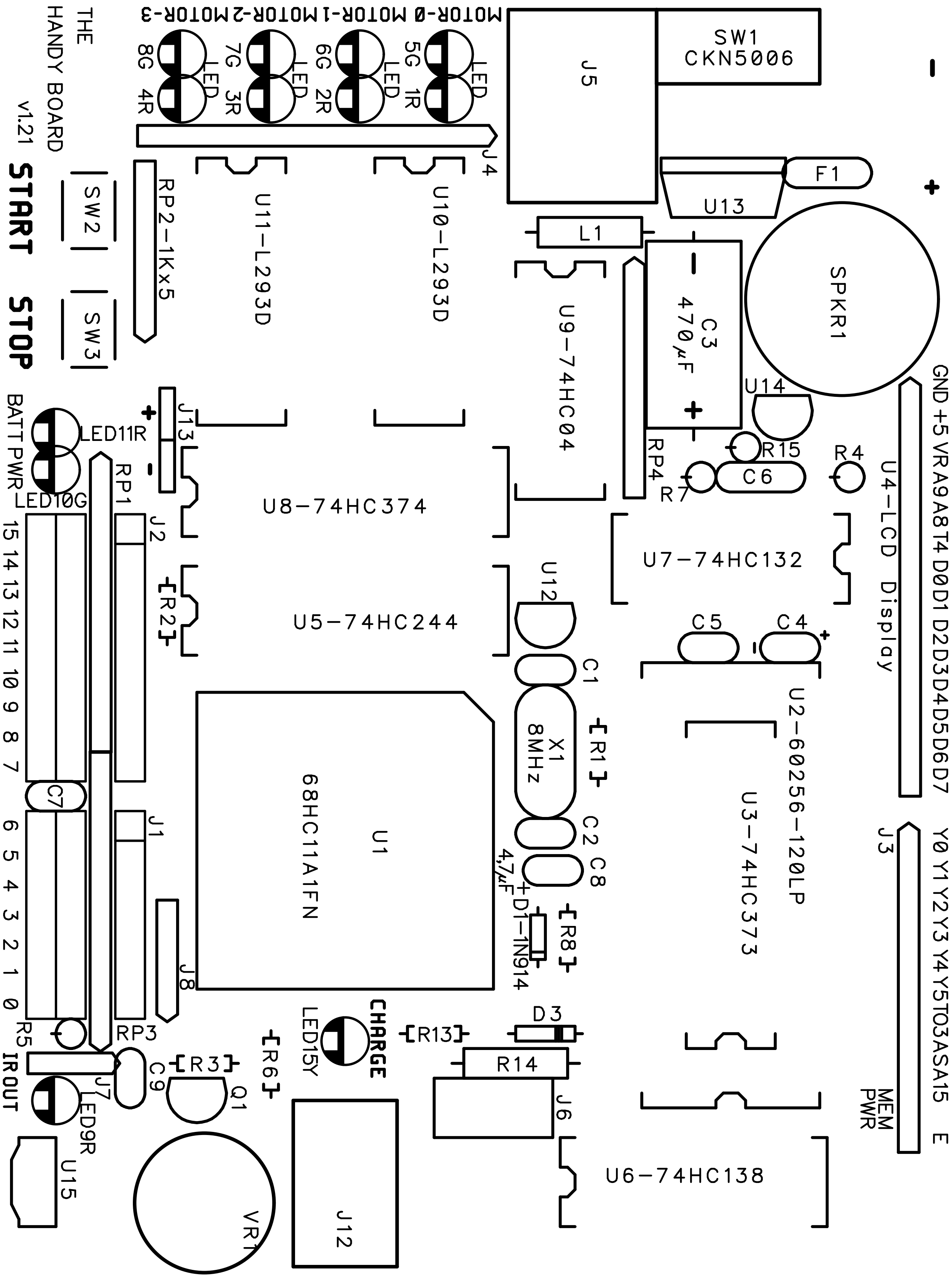
10.1 Handy Board Component Side



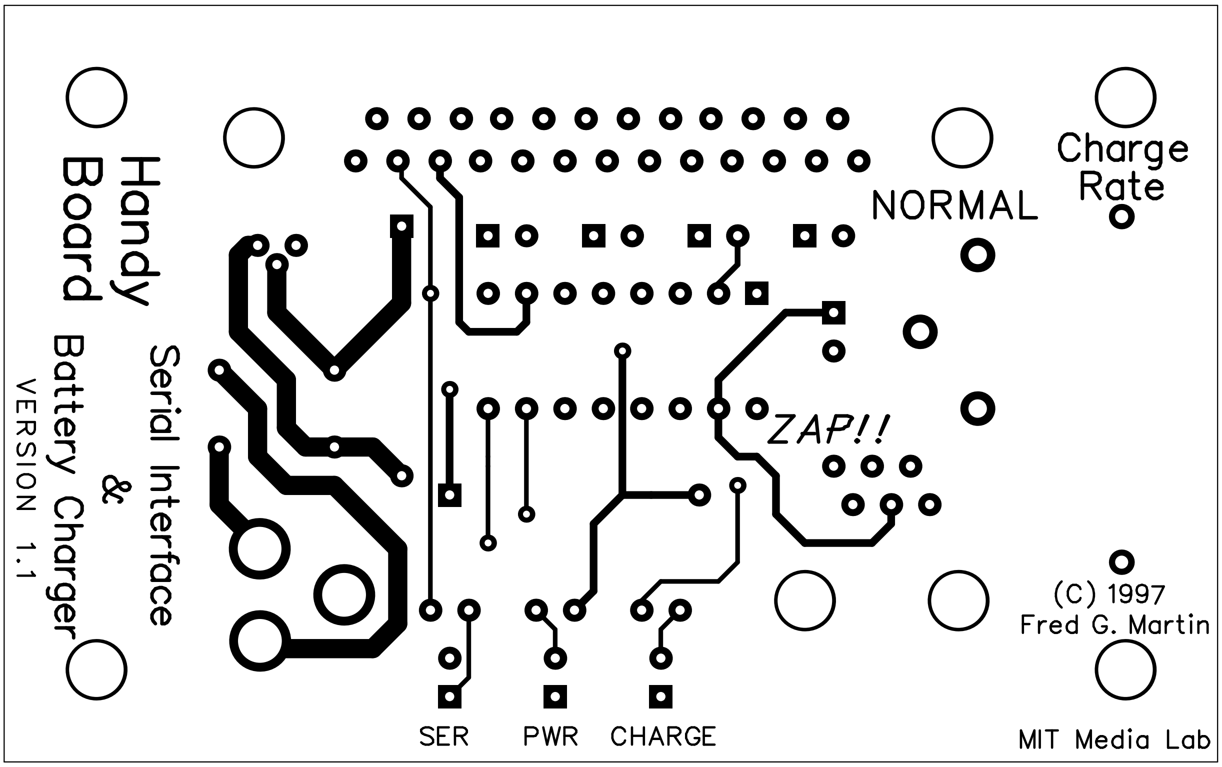
10.2 Handy Board Solder Side



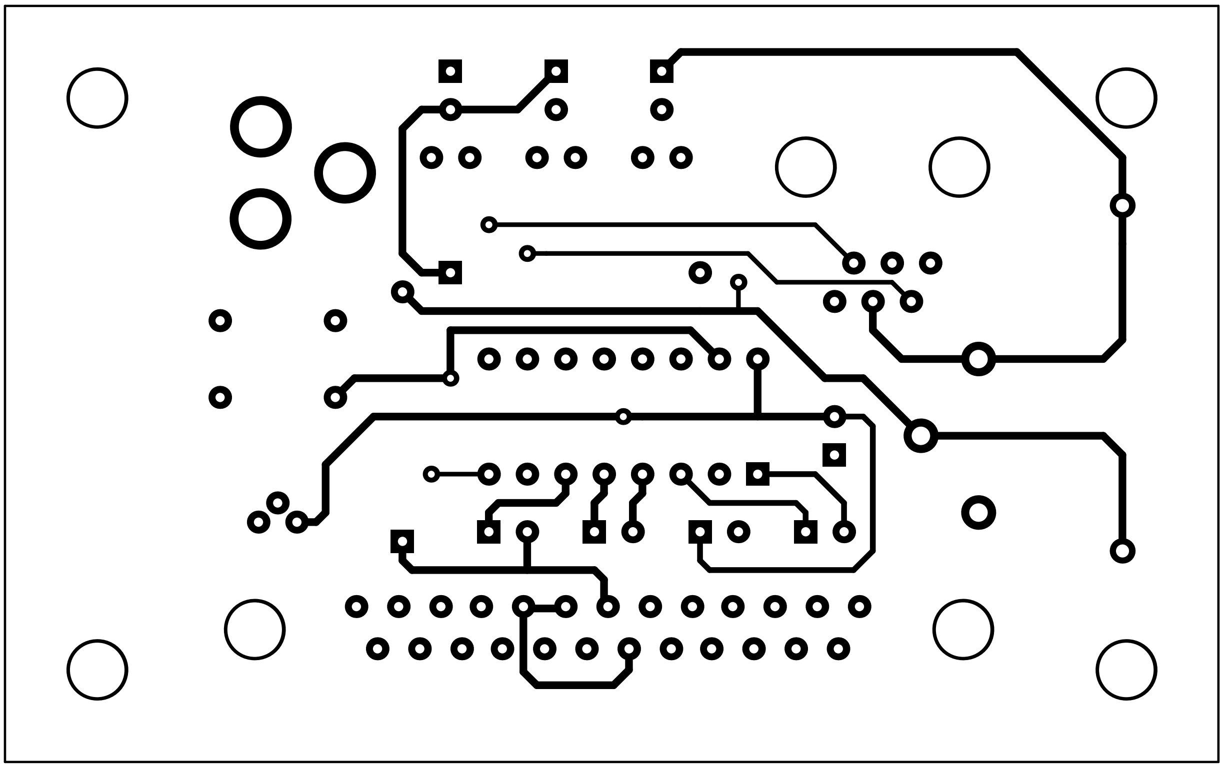
10.3 Handy Board Silkscreen



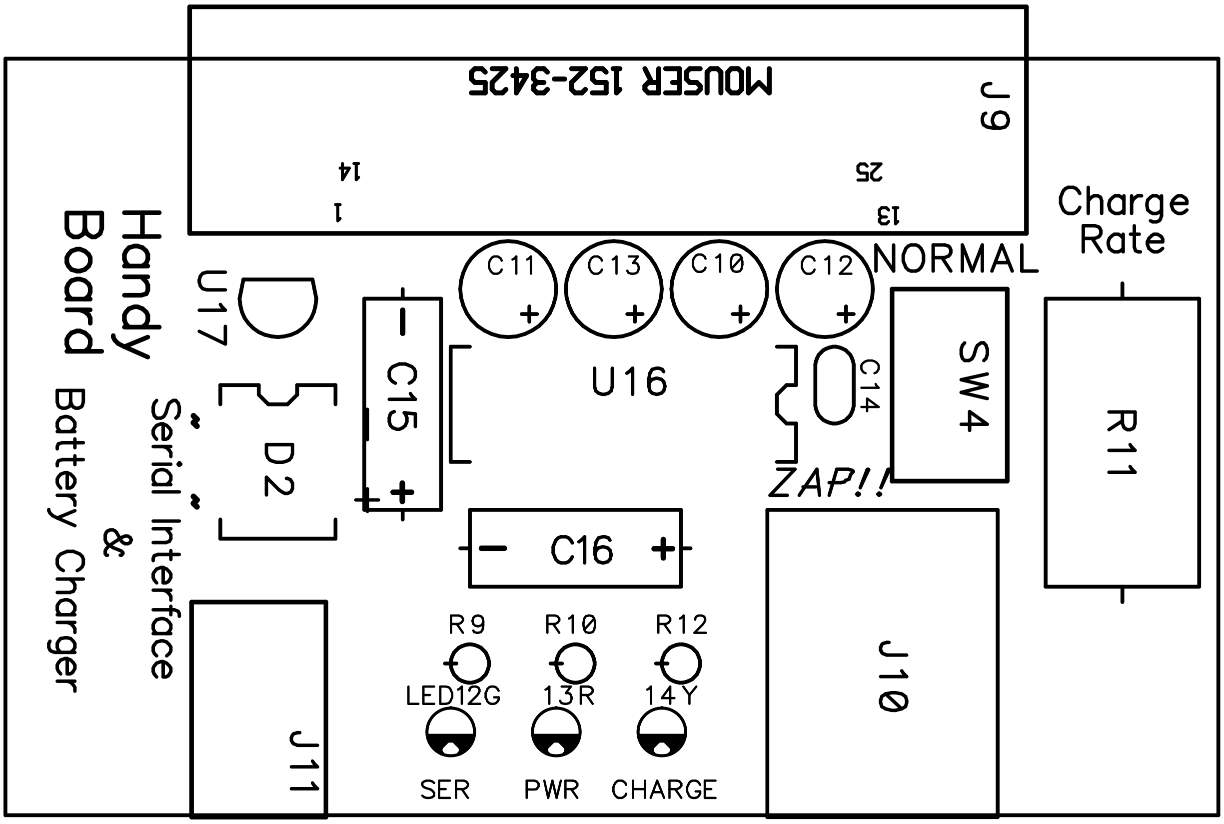
10.4 Interface/Charger Board Component Side

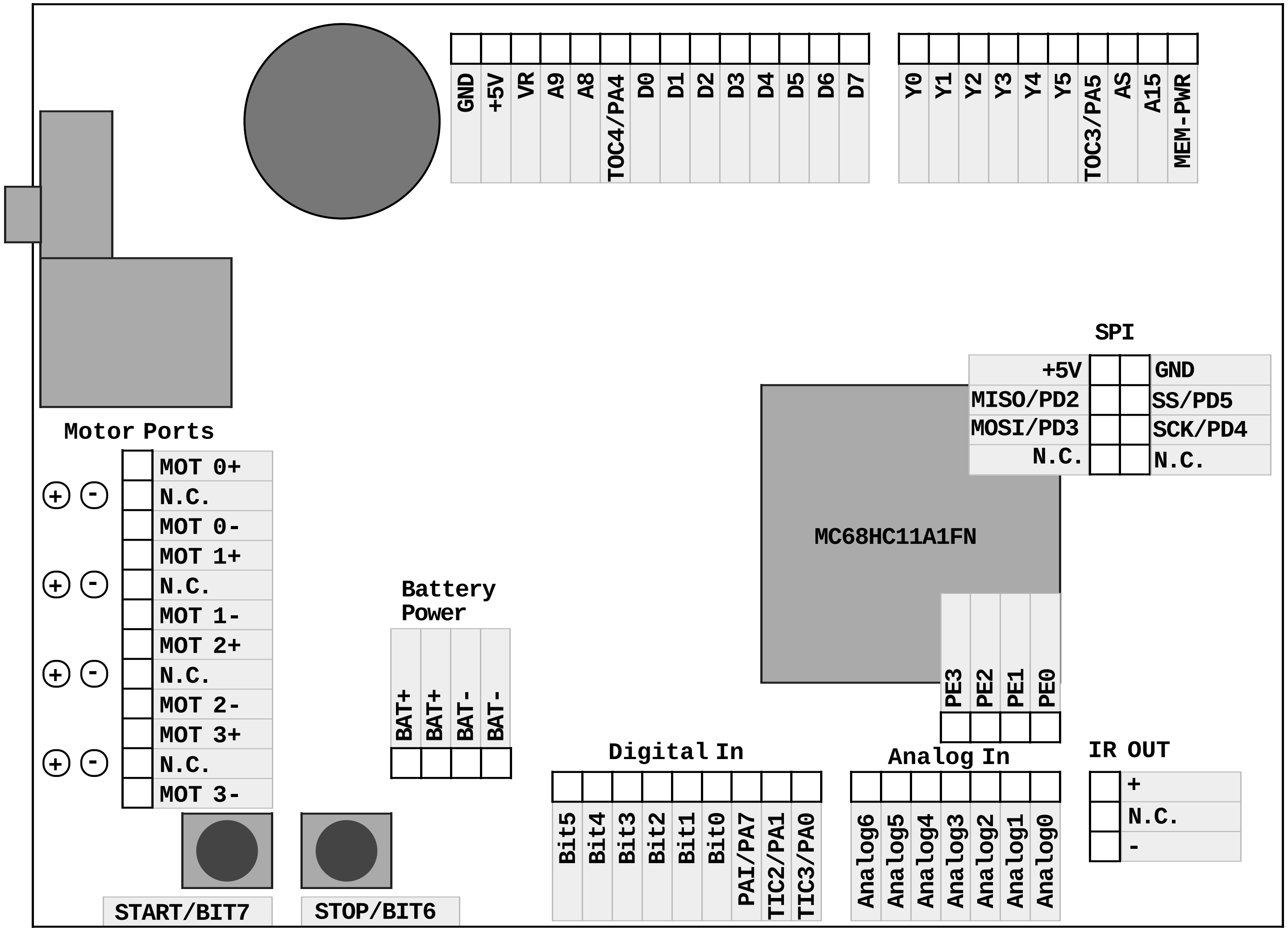


10.5 Interface/Charger Board Solder Side



10.6 Interface/Charger Board Silkscreen





Notes:
The 'Bit X' on the buttons and Digital In are bits of a read to 0x7FFF.

Y0 through Y5 are active lows. This table shows what to do to force them low for a cycle.

```
Write 0x4000 = Y0
Read  0x4000 = Y1
Write 0x5000 = Y2
Read  0x5000 = Y3
Write 0x6000 = Y4
Read  0x6000 = Y5
```

This diagram was contributed by Brian Schmalz.

12 Frequently Asked Questions

This section answers some common questions and problems regarding the Handy Board. Please note that the information here is only a subset of the full FAQ, which is on-line. For more questions and answers, please refer to the on-line FAQ at `http://handyboard.com/faq/` .

12.1 Hardware

12.1.1 Motor Voltage

How can I use motors other than 9 volts with the Handy Board?

The Handy Board's internal battery is rated for 9.6 volts; this is generally adequate for running motors rated between 6 and 12 volts. However, some 6 volt motors aren't happy with the extra voltage, and some 12 volt motors will run too slowly.

Instructions are available on the Handy Board web site for modifying the Handy Board to accept an external motor battery that can be any voltage from about 6 volts to 36 volts. See `http://handyboard.com/mods/hbmcut.html` .

Lyle Hazelwood implemented a high-current H-bridge circuit described by Chuck McManis and tested by Jeffrey Keyzer, and posted the schematic and circuit notes. See `ftp://cherupakha.media.mit.edu/pub/contrib/lylehaze/hbridge/` .

12.1.2 Digital Outputs

Does the Handy Board have any digital outputs?

The SPI pins on the connector on the middle right edge of the board (**J6**) can be configured as digital outputs. Do a `poke(0x1009, 0x3c)` to make them outputs; then they are mapped to the middle 4 bits of address 0x1008 (SS= bit 5, SCK= bit 4, MOSI= bit 3, MISO= bit 2). Poke to that address (0x1008) to set them.

d0..d7 is the data bus and stuff is flying around on those pins all the time, so they cannot be used as outputs. If you hook an 'hc374 chip to the board, in the same fashion as the one driving the motor chips, you get 8 more digital outs. Connect the 'hc374's clock line to any of the three unused output latch selects of the 'hc138 (**Y0**, **Y2**, or **Y4**). All of these signals are present on the Expansion Bus.

Also, digital input #9 can be reconfigured as an output. Do a `bit_set(0x1026, 0x80)` to make it an output, and then use `bit_set(0x1000, 0x80)` to turn the bit on and `bit_clear(0x1000, 0x80)` to turn it off.

Finally, **TO3** is an uncommitted timer output brought out to the Expansion Bus. This pin is bit 5 of PORTA; i.e., set it with `bit_set(0x1000, 0x20)` and clear it with `bit_clear(0x1000, 0x20)`.

12.1.3 High Adapter Voltage

I'm measuring the voltage on my adapter, and it says 18 volts. Is this normal?

This is correct. Normal DC adapters are unregulated and there is an inverse relationship between voltage and current.

Here is how to interpret the rating on an adapter. Let's use the Handy Board's 12 volt, 500 mA (milliamp) DC adapter standard as an example.

This rating means that when a load is drawing 500 mA of current, the adapter voltage will be 12 volts.

If the adapter is plugged into the wall but its output is not connected to anything—in other words, there is no load—then the current is zero and the voltage measured will higher than the adapter's specification. For the Handy Board's "12 volt" adapter, a reading of 18 volts is normal if there is no load.

If there is load that draws *more* than 500 mA, then the output voltage would be less than 12 volts. Note that it *is possible* to draw more than 500 mA even though an adapter might only be rated for 500 mA. The effect is that the output voltage will be less than the adapter's specified voltage, and also this will overtax the adapter and potentially cause it to fail.

12.2 Software

12.2.1 ICB Files

IC won't load my ICB files.

Please note an important bug related to ICB files. On the MS-DOS platform (with both the freeware v2.853 and commercial 3.1 beta 4) version of Interactive C, the ICB files must have Unix-style line termination.

Here is the explanation. ICB files are text files, and in a text file, the Mac, Unix, and MS-DOS file systems each have a different way of specifying the end of each text line. On the Mac, a ctrl-M indicates the end of line. On Unix, it's a ctrl-J. On the PC, it's a ctrl-M followed by a ctrl-J.

On the MS-DOS computer platform, if you use Newton Lab's web-based ICB assembler (located at `href=http://www.newtonlabs.com/ic/icb.html`), or if you download an ICB file from an FTP server, when you save the resulting ICB file it will undoubtedly create a normal MS-DOS text file, with ctrl-M + ctrl-J linefeeds. You must edit this file and remove all of the ctrl-M's.

Originally, ctrl-J meant line feed and ctrl-M meant carriage return (think of a TeleType machine). So on MS-DOS, when you remove the ctrl-M's you get files where each new line starts where the last one ended in terms of screen column. This will look wrong but it is what the MS-DOS Interactive C versions require.

On the Mac and Unix platforms, the IC accepts the corresponding native text file format. But the MS-DOS and Windows versions of IC require the CTRL-M's to be edited out of ICB files.

Remi Desrosiers (`silverwa@odyssee.net`) contributed a DOS utility to automatically strip out the CTRL-M's. It is available at `http://handyboard.com/software/i2u.exe` .

Please make sure to save as source so it gets downloaded as a binary file.

12.2.2 Power Glitch

I keep getting a message that says "`-POWER GLITCH-`*" on the Handy Board LCD screen.*

This is caused when the incorrect pcode file is downloaded to the Handy Board. Make sure you are downloading `pcode_hb.s19`, *not* `pcoder22.s19`.

It may be necessary to reconfigure your downloader to send the proper file. If you are using *Initialize Board* on the Macintosh, use ResEdit to change the `STR` resource, naming "`pcode_hb.s19`" as the file to download.

12.2.3 I can't get any of the downloaders to work on my fast Windows 95 machine. What is wrong?

For presently unknown reasons, some fast Pentiums have trouble running the downloaders properly. Here are some suggestions that many Handy Board users have found helpful:

- When using `dl.exe`, run it in a full-screen DOS mode. (This may also be necessary when using the freeware DOS version of Interactive C.)
- In the advanced serial port options, change the setting for receive buffer and transmit buffer to the lowest value possible (min).

13 Vendors

The Handy Board is commercially available from the following companies:

Gleason Research. Gleason Research supplies assembled Handy Board systems, Expansion Boards, and accessories. Gleason Research, P.O. Box 1494, Concord, MA 01742. Phone (510) 665–2302; fax (978) 287–4170; E-mail `info@gleasonresearch.com`; URL `http://gleasonresearch.com/`.

Robot Store HK. Robot Store HK, located in Hong Kong, supplies Handy Boards, Expansion Boards, and accessories, assembled or in kit form. Mr Hui, Pak Ki, Robot Store (HK), 7th Floor, Fok Wa Mansion, No. 19 Kin Wah Street, North Point, Hong Kong. Telephone (852) 2563–8511; fax (852) 2887–2519; E-mail `huip@hkstar.com`; URL `http://www.robotstorehk.com/`.

Douglas Electronics. Douglas Electronics supplies blank Handy Board printed circuit boards, blank Expansion Boards, and Expansion Board parts kits. Douglas Electronics, Inc., 2777 Alvarado Street, San Leandro, CA 94577 USA. Phone (510) 483–8770; fax (510) 483–6453; BBS (FirstClass) (510) 483–6548; E-mail `info@douglas.com`.

Acroname. Acroname sells unassembled Handy Boards, Expansion Boards, and accessories. Acroname, 5621 Arapahoe Ave, Suite C, Boulder CO, 80303. Phone (720) 564–0373; fax (720) 564–0376; E-mail `sales@acroname.com`; URL `http://www.acroname.com/`.

14 Handy Board Community

In order to support everyone who wants to use a Handy Board, it is crucial that all users help each other in troubleshooting problems, exchanging ideas and techniques, and sharing code. Neither I (Fred Martin) nor the Handy Board vendors can do it alone.

The Handy Board discussion group is the main way that Handy Board users communicate with each other. *All active Handy Board users are encouraged to seek technical support, advise, and ideas from the discussion group community.*

For information, see `http://news.lugnet.com/robotics/handyboard/`.

15 Licensing

The Handy Board technology, including the printed circuit board layout and supplied code libraries, is distributed under a free licensing policy. This agreement allows any party to use the Handy Board technology for any purpose without having to pay a licensing fee.

The technology is *not* public domain. The Massachusetts Institute of Technology reserves the copyright to the artwork and code. Any commercial use of the technology must include a reproduction of the copyright notice on the board itself, and must acknowledge the institutional source (MIT) and author (Fred Martin) of the technology in an appropriate fashion in any accompanying product documentation.

A copy of the documentation authorizing this usage is available from the Handy Board web site `http://handyboard.com/howtoget/license.html`.