Chapter 7

IC Manual

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.

7.1 Getting Started

This section describes how to boot IC on the 6.270 board using the MIT Athena computer network. Commands that are typed to the computer are shown underlined for visibility.

1. **Add the `6.270` directory to the execution path.** Type the following command at the Unix prompt:

   `add 6.270`

2. **Plug the board into the computer.** Using the modular phone cable, plug the modified end into the DEC or VAX's printer port (indicated by an icon of a computer printer). *Make sure that the exposed wires of the plug are facing up.* If using the VAX2000 computer, use the 9-pin plug adapter.

   Plug the other end into the modular jack on the 6811 board. *Before the board is turned on*, check that the board's green LED (labelled SER RCV) is lit. If it is not lit, there is a problem with the connection.

3. **Initialize the board.** The first step is using IC is to load the run-time module (called the "p-code program") into the board. If the p-code is already loaded, this step may be skipped. If not:

   - Switch the board on. *Do not* hit the reset button at this time. When the board is switched on, the yellow LED (labelled SER XMIT) should flash briefly and then stay off. *If the yellow LED is lit, the board is not ready to be initialized. Turn the board off and on and try again.*
   - From the Unix prompt, type

     `init_bd`

     A process should begin that downloads the p-code program to the board. This will take about 15 to 30 seconds to complete. If the program exits with an error message, check the connection and try again.

4. **Reset the board.** Press the reset button on the board to reset it. The following should happen:

   (a) The board will emit a brief beep;

   (b) A version message will be printed on the LCD screen (e.g., "`IC vX.XX`");

(c) The yellow LED will turn on brightly.

If these things do not happen, repeat step 3 to initialize the board.

5. **Begin IC.** From the Unix prompt, type:

   IC

At this point, IC will boot, ready to load a C program or evaluate expressions typed to the IC prompt.

7.2 Using IC

IC is started from the Unix shell by typing `ic` at the prompt. Some Unix systems (in particular, MIT Athena DECstations) have an unrelated application named `ic`. If this application is first in the execution path, it will be invoked rather than the IC compiler. This situation may be remedied by reordering the execution path to include the path to the IC compiler first, or by using the program name `icc`, which will also invoke IC.

IC can be started with the name (or names) of a C file to compile.

When running and attached to a 6811 system, C expressions, function calls, and IC commands may be typed at the "`C>`" prompt.

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

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

When this expression is typed, it is compiled by the console computer and then downloaded to the 6811 system for evaluation. The 6811 then evaluates the compiled form and returns the result, which is printed on the console computer's screen.

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 6811's LCD screen:

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

7.2.1 IC Commands

IC responds to the following commands:

- **Load file.** The command `load` <*filename*> compiles and loads the named file. The 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.

- **Edit a file.** The command `edit` <*filename*> brings up a system editor to allow editing of a file. This command is most useful on single-tasking operating systems, like MS-DOS.

- **Run an inferior shell.** If IC is running on a MS-DOS system, this command opens a shell to execute MS-DOS functions.

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

- **Quit.** The command `quit` exits IC. CTRL-C can also be used.

7.2.2 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.

An earlier line in the command history can be retrieved by typing the exclamation point followed by the first few characters of the line to retrieve, and then the space bar.

Figure 7.1 shows the keystroke mappings understood by IC.

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

Keystroke	**Function**
DEL	backward-delete-char
CTRL-A	beginning-of-line
CTRL-B	backward-char
←	backward-char
CTRL-D	delete-char
CTRL-E	end-of-line
CTRL-F	forward-char
→	forward-char
CTRL-K	kill-line
CTRL-U	universal-argument
ESC D	kill-word
ESC DEL	backward-kill-word

Figure 7.1: IC Command-Line Keystroke Mappings

7.2.3 The `main()` Function

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

To reset with board *without* running the `main()` function (for instance, when hooking the board back to the computer), hold down one of the two user input buttons on the board while pressing reset. The board will reset without running `main()`.

7.3 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.

7.4 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.

7.4.1 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.

7.4.2 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.

7.4.3 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 robot 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.

7.4.4 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."`.

7.4.5 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 7.10 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).

7.4.6 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:

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

7.4.7 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.

7.4.8 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.

<table>
<tr><th>Operator</th><th>Associativity</th></tr>
<tr><td><code>() []</code></td><td>left to right</td></tr>
<tr><td><code>! ~ ++ -- -</code> (type)</td><td>right to left</td></tr>
<tr><td><code>* / %</code></td><td>left to right</td></tr>
<tr><td><code>+ -</code></td><td>left to right</td></tr>
<tr><td><code><< >></code></td><td>left to right</td></tr>
<tr><td><code>< <= > >=</code></td><td>left to right</td></tr>
<tr><td><code>== !=</code></td><td>left to right</td></tr>
<tr><td><code>&</code></td><td>left to right</td></tr>
<tr><td><code>^</code></td><td>left to right</td></tr>
<tr><td><code>|</code></td><td>left to right</td></tr>
<tr><td><code>&&</code></td><td>left to right</td></tr>
<tr><td><code>||</code></td><td>right to left</td></tr>
<tr><td><code>= += -=</code> etc.</td><td>right to left</td></tr>
<tr><td><code>,</code></td><td>left to right</td></tr>
</table>

7.5 Control Flow

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

7.5.1 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.

7.5.2 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.

7.5.3 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

7.5.4 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);
```

7.5.5 Break

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

7.6 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.

7.6.1 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.

7.6.2 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)

7.6.3 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 board has failed.
- Characters that would be printed beyond the final character position are truncated.
- When using a two-line display, the `printf()` command treats the display as a single longer line.
- Printing of long integers is not presently supported.

7.7 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.

7.7.1 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.

7.7.2 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);
}
```

7.7.3 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.

7.7.4 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`.

7.8 The IC Library File

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

IC automatically loads the library file every time it is invoked. Depending on which 6811 board is being used, a different library file will be required. IC may be configured to load different library files as its default; for the purpose of the 6.270 contest, the on-line version of IC will be configured appropriately for the board that is in use.

> *As of this writing, there are three related 6811 systems in use: the 1991 6.270 Board (the "Revision 2" board), the 1991 Sensor Robot, and the 1992 6.270 Board (the "Revision 2.1" board). This writing covers the 1992 board only; documentation for the other two systems is available elsewhere.*

On the MIT Athena system, IC library files are located in the directory `/mit/-6.270/lib/ic`. (To understand better how the library functions work, study of the library file source code is recommended.) The main library file for the 1992 6.270 Board is named `lib_r21.lis`.

7.8.1 Output Control

DC Motors

Motor ports are numbered from 0 to 5; ports for motors 0 to 3 are located on the Microprocessor Board while motors 4 and 5 are located on the Expansion Board.

Motor 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 the 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. The servo motor has a three-wire connection: power, ground, and control. The power and ground of the servo motor connect to the power port labelled PWR OUT on the Microprocessor Board; the control wire plugs into the pin labelled D2 of the Port D I/O connector.

The position of the servo motor shaft is controlled by a rectangular waveform that is generated on the D2 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 determined by the value of a library variable.

`int servo_pulse_wavetime`
Library variable. Value is the time in half-microseconds of the positive portion of a rectangular wave that is generated on the Port D2 pin for use in controlling a servo motor.

Initial value is 2000, yielding a pulse time of one millisecond. Values between 1400 and 4000 (.7 to 2 milliseconds) are boundary settings for typical servo motors.

Unidirectional Drivers

LED Drivers There are two output ports located on the Expansion Board that are suitable for driving LEDs or other small loads. These ports draw their power from the motor battery and hence will only work when that battery is connected.

The following commands are used to control the LED ports:

`void led_out0(int s)`
Turns on LED0 port if `s` is non-zero; turns it off otherwise.

`void led_out1(int s)`
Turns on LED1 port if `s` is non-zero; turns it off otherwise.

Expansion Board Motor Ports Motor ports 4 and 5, located on the Expansion Board, may also be used to control unidirectional devices, such as a solenoid, lamp, or a motor that needs to be driven in one direction only. Each of the two motor ports, when used in this fashion, can independently control two such devices.

To use the ports unidirectionally, the two-pin header directly beneath the motor 4 and 5 LEDs is used.

`void motor4_left(int s)`
Turns on left side of motor 4 port if `s` is non-zero; turns it off otherwise.

`void motor4_right(int s)`
Turns on right side of motor 4 port if `s` is non-zero; turns it off otherwise.

`void motor5_left(int s)`
Turns on left side of motor 5 port if `s` is non-zero; turns it off otherwise.

`void motor5_right(int s)`
Turns on right side of motor 5 port if `s` is non-zero; turns it off otherwise.

7.8.2 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 Microprocessor Board and Expansion 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 Microprocessor Board and Expansion Board.

`int motor_force(int m)`

Returns value of analog input sensing current level through motor `m`. Result is integer between 0 and 255, but typical readings range from about 40 (low force) to 100 (high force).

The force-sensing circuitry functions properly only when motors are operated at full speed. The circuit returns invalid results when motors are pulse-width modulated because of spikes that occur in the feedback path.

The force-sensing circuitry is implemented for motors 0 through 3.

`int dip_switch(int sw)`

Returns value of DIP switch `sw` on interface board. Switches are numbered from 1 to 4 as per labelling on actual switch. Result is 1 if the switch is in the position labelled "on," and 0 if not.

`int dip_switches()`

Returns value on DIP switches as a four-bit binary number. Left-most switch is most significant binary digit. "On" position is binary one.

`int choose_button()`

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

Example:

```
/*  wait until choose button pressed  */
while (!left_button()) {}
```

`int escape_button()`

Returns value of button labelled ESCAPE.
Example:

```
/*  wait for button to be pressed; then
    wait for it to be released so that
    button press is debounced          */
while (!escape_button()) {}
while (escape_button()) {}
```

Infrared Subsystem

The infrared subsystem is composed of two parts: an infrared transmitter, and infrared receivers. Software is provided to control transmission frequency and detection of infrared light at two frequencies.

Infrared Transmission

`void ir_transmit_on()`

Enables transmission of infrared light through IR OUT port.

`void ir_transmit_off()`

Disables transmission of infrared light through IR OUT port.

`void set_ir_transmit_frequency(int period)`

Sets infrared transmission frequency. `period` determines the delay in half-microseconds between transitions of the infrared waveform. If `period` is set to 10,000, a frequency of 100 Hz. will be generated. If `period` is set to 8,000, a frequency of 125 Hz. will be generated. The decoding software is capable of detecting transmissions on either of these two frequencies only.

Upon a reset condition, the infrared transmission frequency is set for 100 Hz. and is disabled.

Infrared Reception In a typical 6.270 application, one robot will be broadcasting infrared at 100 Hz. and will set its detection system for 125 Hz. The other robot will do the opposite. Each robot must physically shield its IR sensors from its own light; then each robot can detect the emissions of the other.

The infrared reception software employs a *phase-locked loop* to detect infrared signals modulated at a particular frequency. This program generates an internal squarewave at the desired reception frequency and attempts to lock this squarewave into synchronization with a waveform received by an infrared sensor. If the error between the internal wave and the external wave is below some threshold, the external wave is considered "detected." The software returns as a result the number of consecutive detections for each of the infrared sensor inputs.

Up to four infrared sensors may be used. These are plugged into positions 0 through 3 of the digital input port. These ports and the remainder of the digital input port may be used without conflict for standard digital input while the infrared detection software is operating.

The following library functions control the infrared detection system:

`void ir_receive_on()`

Enables the infrared reception software. The default is disabled. When the software is enabled, between 20% and 30% of the 6811 processor time will be spent performing the detection function; therefore it should only be enabled if it is being used.

`void ir_receive_off()`

Disables the infrared reception software.

`void set_ir_receive_frequency(int f)`

Sets the operating frequency for the infrared reception software. `f` should be 100 for 100 Hz. or 125 for 125 Hz. Default is 100.

`int ir_counts(int p)`

Returns number of consecutive squarewaves at operating frequency detected from port `p` of the digital input port. Result is number from 0 to 255. `p` must be 0, 1, 2, or 3.

Random noise can cause spurious readings of 1 or 2 detections. The return value of `ir_counts()` should be greater than three before it is considered the result of a valid detection.

Shaft Encoders

Machine language drivers are provided to keep count of rapid transitions, as might occur on a shaft encoder sensor. Two types of shaft encoders are *optical*, in which a slotted wheel or black-and-white disk provides visual cues to an optosensor, and *magnetic*, in which a small magnet rotates past a magnetic sensor.

In either case, the task of the software consists of counting pulses. To count accurately, the software uses different thresholds for the rising and falling edge of a pulse. Hence the signal must rise above an upper threshold before being detecting as a valid "logic high" of the pulse, and must fall beneath a lower threshold before being detected as a logic low. This method prevents the possibility that the signal might oscillate rapidly about a single threshold point. (The thresholds will probably need to be calibrated for the particular performance of each sensor arrangment.)

The software returns a total count of pulses, which may be reset by the user, and a velocity, consisting of the number of pulses recorded in the most recent 64 milliseconds. The software does *not* keep track of the direction of rotation of the shaft.

The software samples the sensor at the rate of 1000 Hz. Therefore the software cannot detect pulses more rapid that that frequency.

Software Driver Files The library functions for shaft encoders differ from most other library functions discussed in that they are *not* automatically loaded with the system library. These functions are stored in distinct files and must be explicitly loaded by the user when needed.

A separate driver program is implemented for ports 12 through 17 of the Expansion Board. The files containing the drivers are named `sencdr12.icb` through `sencdr17.icb` (for ports 12 through 17, respectively) and are located in the IC system library. These `icb` files must be loaded at the IC command line or from within a C program's `lis` file (as explained in Section 7.14) when needed.

Although the software drivers run in the background, each uses processing time, and only those which are needed should be loaded. For example, if shaft encoders are to be used on ports 12 and 13, then the files `sencdr12.icb` and `sencdr13.icb` only should loaded.

Shaft Encoder Routines The operation of the shaft encoders is controlled mostly through global variables that are defined when the `icb` files are loaded. Instead of calling a library *function*, getting a measurement from a shaft encoder consists simply of examining a library *variable*.

The library routines and variables are named according to the port number they are designed for and are located in the correspondingly named file. The following explanation assumes the file `sencdr12.icb` has been loaded.

`int encoder12_low_threshold`

Library variable. Its value determines the logic low level of the pulsetrain being measured. Should be calibrated for the performance of actual sensor that is used. Default value is 10; must be between 0 and 255.

`int encoder12_high_threshold`

Library variable. Its value determines the logic high level of the pulsetrain being measured. Should be calibrated for the performance of actual sensor that is used. Default value is 240; must be between 0 and 255.

Example:

```
/*  set threshold points for shaft encoder
    on port 12 */
encoder12_low_threshold= 20;
encoder12_high_threshold= 40;
```

`int encoder12_counts`

Library variable. Continuously updated count of pulses. Each transition from high to low and low to high is registered as one count.

May be reset to zero simply by setting the variable to zero:

```
/* reset port 12 encoder counts to zero */
encoder12_counts= 0;
```

`int encoder12_velocity`

Library variable. Updated every 64 milliseconds as the number of counts recorded over the last 64 millisecond period.

7.8.3 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., pressing reset switch on board) 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);
```

7.8.4 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.

7.9 Multi-Tasking

7.9.1 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.

7.9.2 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).

7.9.3 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);
}
```

7.9.4 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.

7.9.5 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.

7.10 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:

`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.

7.11 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`.

7.12 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.

7.12.1 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.

7.12.2 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

7.13 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.

7.13.1 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 7.4.3, 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.

To explain how these features work, let's look at a sample IC binary source program, listed in Figure 7.2.

```
/* 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 7.2: Sample IC Binary Source File: `testicb.asm`

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

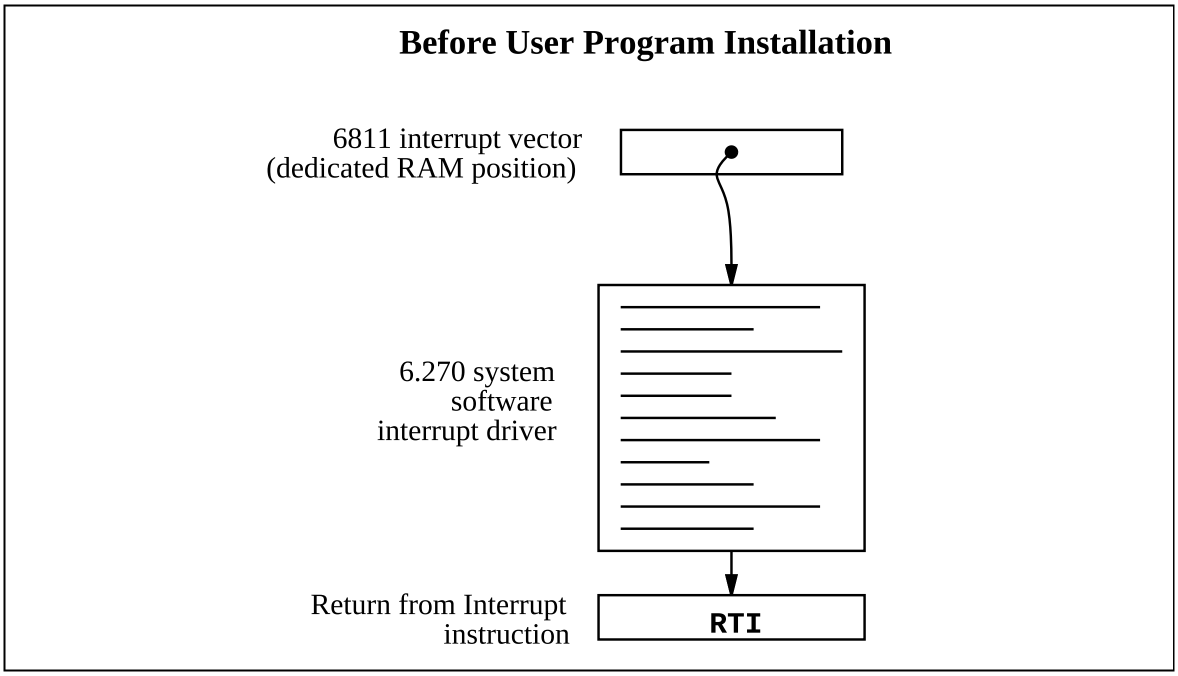

Figure 7.3: Interrupt Structure Before User Program Installation

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.

7.13.2 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 6.270 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 6.270 interrupt code. This vector is made to point at the user program. When user program finishes, it jumps to the start of the 6.270 interrupt code.

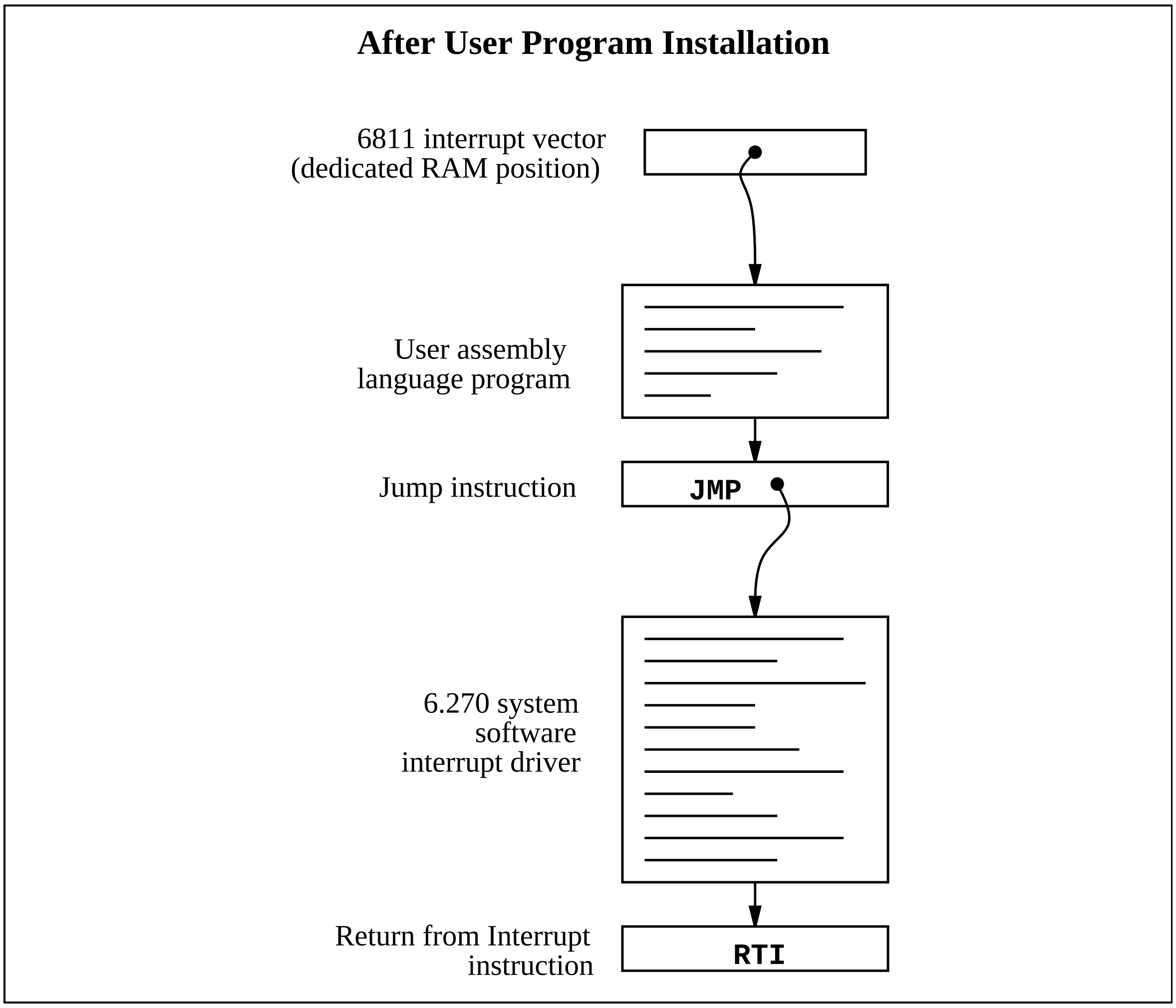

Figure 7.4: Interrupt Structure After User Program Installation

Figure 7.3 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 7.4 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.

Multiple user programs could be installed in this fashion. Each one would install itself ahead of the previous one. Some standard 6.270 library functions, such as the shaft encoder software, is implemented in this fashion.

Figure 7.5 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. (This and other files included by the `as11` assembler are located in the assembler's default library directory, which is `/mit/6.270/lib/as11/` on the MIT Athena system.)

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 7.4.

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 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.

```
* 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 7.5: `sysibeep.asm`: Binary Program that Installs into System Interrupt

7.13.3 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 7.6: 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 7.6 shows the binary object file that is created from the `testicb.asm` example file.

7.13.4 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.

7.13.5 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.

7.14 IC File Formats and Management

This section explains how IC deals with multiple source files.

7.14.1 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.

7.14.2 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.

7.14.3 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
```

7.15 Configuring IC

IC has a multitude of command-line switches that allow control of a number of things. 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.