2.5 Using C Programs to Explore Data Formats

> *Before writing any programs, I urge you to read Appendix B on writing Makefiles, even if you are familiar with them. Many of the problems I have helped students solve are due to errors in their Makefile. And many of the Makefile errors go undetected due to the default behavior of the `make` program.*

We will use the C programming language to illustrate these concepts because it takes care of the memory allocation problem, yet still allows us to get reasonably close to the hardware. You probably learned to program in the higher-level, object-oriented paradigm using either C++ or Java. C does not support the object-oriented paradigm.

C is a *procedural programming* language. The program is divided into functions. Since there are no classes in C, there is no such thing as a member function. The programmer focuses on the algorithms used in each function, and all data items are explicitly passed to the functions.

We can see how this works by exploring the C Standard Library functions, `printf` and `scanf`, which are used to write to the screen and read from the keyboard. We will develop a program in C using `printf` and `scanf` to illustrate the concepts discussed in the previous sections. The header file required by either of these functions is:

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

which includes the prototype statements for the `printf` and `scanf` functions:

```
int printf(const char *format, ...);
int scanf(const char *format, ...);
```

`printf` is used to display text on the screen. The first argument, `format`, controls the text display. At its simplest, `format` is simply an explicit text string in double quotes.[1] For example,

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

would display

```
Hello, world.
```

If there are additional arguments, the format string must specify how each of these arguments is to be converted for display. This is accomplished by inserting a conversion code within the format string at the point where the argument value is to be displayed. Each conversion code is introduced by the `'%'` character. For example, Listing 2.1 shows how to display both an `int` variable and a `float` variable.

```
1  /*
2   * intAndFloat.c
3   * Using printf to display an integer and a float.
4   * Bob Plantz - 4 June 2009
5   */
6  #include <stdio.h>
7
8  int main(void)
9  {
10     int anInt = 19088743;
11     float aFloat = 19088.743;
```

[1] The text string is a null-terminated array of characters as described in Section 2.7 (page 21). This is not the C++ string class.

```
12
13      printf("The integer is %i and the float is %f\n", anInt, aFloat);
14
15      return 0;
16  }
```

Listing 2.1: Using `printf` to display numbers.

Compiling and running the program in Listing 2.1 on my computer gave (user input is **boldface**):

```
bob$ gcc -Wall -o intAndFloat intAndFloat.c
bob$ ./intAndFloat
The integer is 19088743 and the float is 19088.742188
bob$
```

Yes, the `float` really is that far off. This will be explained in Chapter 14.

> This is not a book about how to use the GNU development environment, so I usually do not show the compile command. I am showing it here to help get you started. You should use the `man gcc` command to learn about the command line options.

Some common conversion codes are `d` or `i` for integer, `f` for float, and `x` for hexadecimal. The conversion codes may include other characters to specify properties like the field width of the display, whether the value is left or right justified within the field, etc. We will not cover the details here. You should read `man` page 3 for `printf` to learn more.

`scanf` is used to read from the keyboard. The format string typically includes only conversion codes that specify how to convert each value as it is entered from the keyboard and stored in the following arguments. Since the values will be stored in variables, it is necessary to pass the address of the variable to `scanf`. For example, we can store keyboard-entered values in `x` (an `int` variable) and `y` (a `float` variable) thusly

```
scanf("%i %f", &x, &y);
```

The use of `printf` and `scanf` are illustrated in the C program in Listing 2.2, which will allow us to explore the mathematical equivalence of the decimal and hexadecimal number systems.

```
1   /*
2    * echoDecHex.c
3    * Asks user to enter a number in decimal and one
4    * in hexadecimal then echoes both in both bases
5    * Bob Plantz - 4 June 2009
6    */
7
8   #include <stdio.h>
9
10  int main(void)
11  {
12      int x;
13      unsigned int y;
14
15      while(1)
16      {
17          printf("Enter a decimal integer (0 to quit): ");
18          scanf("%i", &x);
```

```
19         if (x == 0) break;
20
21         printf("Enter a bit pattern in hexadecimal (0 to quit): ");
22         scanf("%x", &y);
23         if (y == 0) break;
24
25         printf("%i is stored as %#010x, and\n", x, x);
26         printf("%#010x represents the decimal integer %i\n\n", y, y);
27     }
28
29     printf("End of program.\n");
30
31     return 0;
32 }
```

Listing 2.2: C program showing the mathematical equivalence of the decimal and hexadecimal number systems.

Here is an example run of this program (user input is **boldface**):

```
bob$ ./echoDecHex
Enter a decimal integer:  123
Enter a bit pattern in hexadecimal:  7b
123 is stored as 0x0000007b, and
0x0000007b represents the decimal integer 123

Enter a decimal integer:  0
End of program.
bob$
```

Let us walk through the program in Listing 2.2.

- The program declares two `int`s, `x` and `y`.
- The user is prompted to enter an integer in decimal, and the user's response is read from the keyboard and stored in the memory allocated for `x`. The conversion code text string passed to `scanf`, "%i", causes `scanf` to interpret the user's keystrokes as representing a decimal integer. Note that the address of `x`, `&x`, must be passed to `scanf` so that it can store the integer at the memory location named `x`.
- The program next prompts the user to enter a bit pattern in hexadecimal. In this case the conversion code text string passed to `scanf` is "%x", which causes `scanf` to interpret the user's keystrokes as representing hexadecimal digits. Note that the address of `y`, `&y`, must be passed to `scanf` so that it can store the integer at the memory location named `y`.
- Now let us examine the two printf function calls that display the results. The "%i" conversion code is straightforward. The value of the corresponding variable is displayed in decimal at that point in the text string.
- The "`%#010x`" conversion factor is more interesting. (If you are at a computer read section 3 of the `man` page for `printf` as you follow through this description.) The basic conversion is specified by the "`x`" character; it causes the value to be displayed in hexadecimal. The "#" character causes an "alternate form" to be used for the display, which is the C syntax for hexadecimal numbers; that is, the value is prefaced by `0x` when it is displayed. The '`0`' character immediately after the '#'

character causes '`0`' to be used as the fill character. The number "`10`" causes the display to occupy at least ten characters (the field width).

- Look carefully at the output from this program above. The bit patterns used to store the data input by the user, shown in hexadecimal, show that the unsigned `int`s are stored in the binary number system (see Section 2.2, page 8 and Section 2.3, page 9). That is, 123_{10} is stored as $0000007b_{16}$.

The program in Listing 2.2 demonstrates a very important concept — hexadecimal is used as a human convenience for stating bit patterns. A number is not inherently binary, decimal, or hexadecimal. A particular value can be expressed in a precisely equivalent way in each of these three number bases. For that matter, it can be expressed equivalently in any number base.

2.6 Examining Memory With gdb

Now that we have started writing programs, you need to learn how to use the GNU debugger, `gdb`. It may seem premature at this point. The programs are so simple, they hardly require debugging. Well, it is better to learn how to use the debugger on a simple example than on a complicated program that does not work. In other words, tackle one problem at a time.

There is a better reason for learning how to use `gdb` now. You will find that it is a very valuable tool for learning the material in this book, even when you write bug-free programs.

`gdb` has a large number of commands, but the following are the ones that will be used in this section:

- `li` *`lineNumber`* — lists ten lines of the source code, centered at the specified line number.
- `break` *`sourceFilename:lineNumber`* — sets a breakpoint at the specified line in the source file. Control will return to gdb when the line number is encountered.
- `run` — begins execution of a program that has been loaded under control of `gdb`.
- `cont` — continues execution of a program that has been running.
- `print` *`expression`* — evaluate expression and display its value.
- `printf "`*`format`*`",` *`var1`*`,` *`var2`*`,...` — displays the values of the `var`s, using the format specified in the *`format`* `string`.[2]
- `x/`*`nfs memoryAddress`* — displays (examine) *n* values in memory in format *f* of size *s* starting at *memoryAddress*.

We will use the program in Listing 2.1 to see how `gdb` can be used to explore the concepts in more depth. Here is a screen shot of how I compiled the program then used `gdb` to control the execution of the program and observe the memory contents. My typing is **`boldface`** and the session is annotated in *italics*. Note that you will probably see different addresses if you replicate this example on your own (Exercise 2-27).

```
bob$ gcc -g -Wall -o intAndFloat intAndFloat.c
```

[2]Follows the same pattern as the C Standard Library `printf`.

The "-g" option is required. It tells the compiler to include debugger information in the executable program.

```
bob$ gdb ./intAndFloat
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/bob/my_book_working/progs/chap02/intAndFloat...do
(gdb) li
1 /*
2  * intAndFloat.c
3  * Using printf to display an integer and a float.
4  * Bob Plantz - 4 Jun 2009
5  */
6 #include <stdio.h>
7
8 int main(void)
9 {
10      int anInt = 19088743;
(gdb)
11      float aFloat = 19088.743;
12
13      printf("The integer is %i and the float is %f\n", anInt, aFloat);
14
15      return 0;
16 }
(gdb)
```

The `li` command lists ten lines of source code. The display ends with the (gdb) prompt. Pushing the return key will repeat the previous command, and `li` is smart enough to display the next (up to) ten lines.

```
(gdb) br 13
Breakpoint 1 at  0x40050b: file intAndFloat.c, line 13.
```

I set a breakpoint at line 13. When the program is executing, if it ever gets to this statement, execution will pause before the statement is executed, and control will return to gdb.

```
(gdb) run
Starting program: /home/bob/intAndFloat

Breakpoint 1, main () at intAndFloat.c:13
13          printf("The integer is %i and the float is %f\n", anInt, aFloat)
```

The run command causes the program to start execution from the beginning. When it reaches our breakpoint, control returns to gdb.

Each electronic element in a circuit takes time to activate. It is a very short period of time, but it can vary slightly depending upon precisely how the other logic elements are interconnected and the state of each of them when they are activated. The problem here is that the $Control$ input is being used to control the circuit based on the clock signal *level*. The clock level must be maintained for a time long enough to allow all the circuit elements to complete their activity, which can vary depending on what actions are being performed. In essence, the circuit timing is determined by the circuit elements and their actions instead of the clock. This makes it very difficult to achieve a reliable design.

It is much easier to design reliable circuits if the time when an activity can be triggered is made very short. The solution is to use edge-triggered logic elements. The inputs are applied and enough time is allowed for the electronics to settle. Then the next clock *transition* activates the circuit element. This scheme provides concise timing under control of the clock instead of timing determined more or less by the particular circuit design.

5.3.3 Flip-Flops

Although the terminology varies somewhat in the literature, it is generally agreed that (see Figure 5.15.):

- A latch uses a level based clock signal.
- A flip-flop is triggered by a clock signal edge.

At each "tick" of the clock, there are four possible actions that might be taken on a single bit — store 0, store 1, complement the bit (also called *toggle*), or leave it as is.

A *D flip-flop* is a common device for storing a single bit. We can turn the D latch into a D flip-flop by using two D latches connected in a *master/slave* configuration as shown in Figure 5.22. Let us walk through the operation of this circuit.

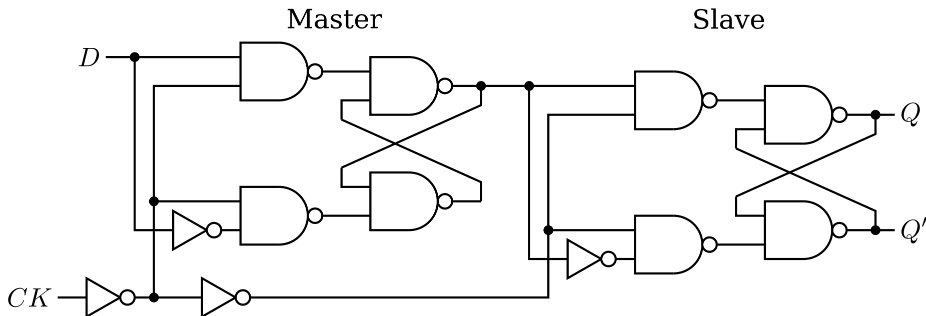


Figure 5.22: D flip-flop, positive-edge triggering.

The bit to be stored, 0 or 1, is applied to the D input of the Master D latch. The clock signal is applied to the CK input. It is normally 0. When the clock signal makes a transition from 0 to 1, the Master D latch will either Reset or Set, following the D input of 0 or 1, respectively.

While the CK input is at the 1 level, the control signal to the Slave D latch is 1, which deactivates this latch. Meanwhile, the output of this flip-flop, the output of the Slave D latch, is probably connected to the input of another circuit, which is activated by the same CK. Since the state of the Slave does not change during this clock half-cycle, the second circuit has enough time to read the current state of the flip-flop connected to its input. Also during this clock half-cycle, the state of the Master D latch has ample time to settle.

When the CK input transitions back to the 0 level, the control signal to the Master D latch becomes 1, deactivating it. At the same time, the control input to the Slave D latch goes to 0, thus activating the Slave D latch to store the appropriate value, 0 or 1. The new input will be applied to the Slave D latch during the second clock half-cycle, after the circuit connected to its output has had sufficient time to read its previous state. Thus, signals travel along a path of logic circuits in lock step with a clock signal.

There are applications where a flip-flop must be set to a known value before the clocking begins. Figure 5.23 shows a D flip-flop with an asynchronous preset input added to it. When a 1 is applied to the PR input, Q becomes 1 and Q' 0, regardless of

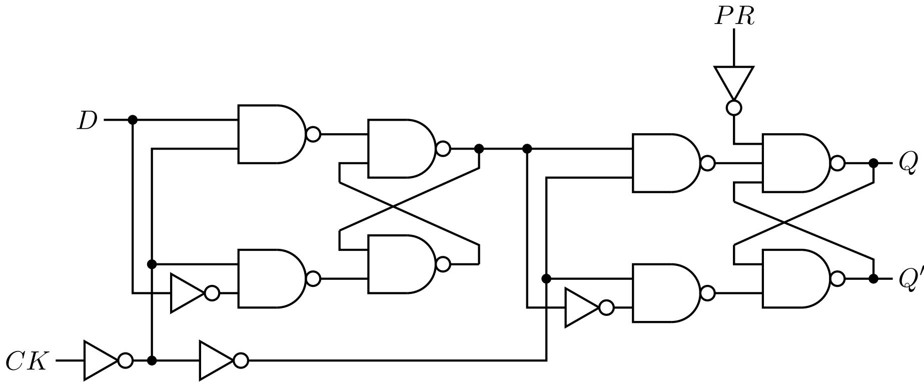


Figure 5.23: D flip-flop, positive-edge triggering with asynchronous preset.

what the other inputs are, even CLK. It is also common to have an asynchronous clear input that sets the state (and output) to 0.

There are more efficient circuits for implementing edge-triggered D flip-flops, but this discussion serves to show that they can be constructed from ordinary logic gates. They are economical and efficient, so are widely used in very large scale integration circuits. Rather than draw the details for each D flip-flop, circuit designers use the symbols shown in Figure 5.24. The various inputs and outputs are labeled in this figure.

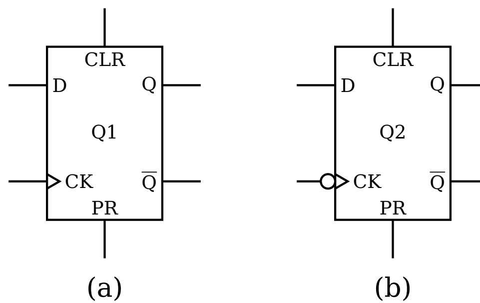
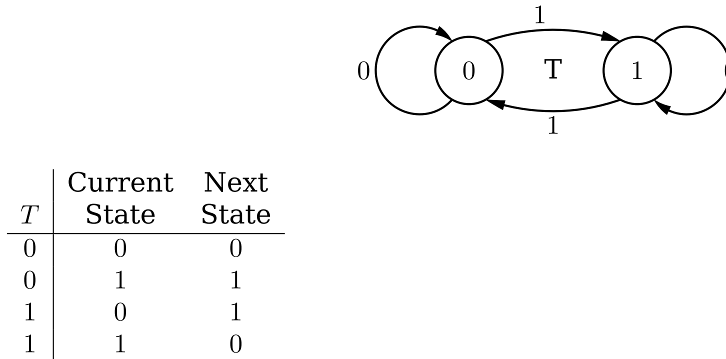


Figure 5.24: Symbols for D flip-flops. Includes asynchronous clear (CLR) and preset (PR). (a) Positive-edge triggering; (b) Negative-edge triggering.

Hardware designers typically use $\overline{Q}$ instead of Q'. It is common to label the circuit as "Q*n*," with *n* = 1, 2,... for identification. The small circle at the clock input in Figure 5.24(b) means that this D flip-flop is triggered by a negative-going clock transition. The D flip-flop circuit in Figure 5.22 can be changed to a negative-going trigger by simply removing the first NOT gate at the CK input.

The flip-flop that simply complements its state, a *T flip-flop*, is easily constructed from a D flip-flop. The state table and state diagram for a T flip-flop are shown in Figure

5.25.



T	Current State	Next State
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5.25: T flip-flop state table and state diagram. Each clock tick causes a state transition, with the next state depending on the current state and the value of the input, T.

To determine the value that must be presented to the D flip-flop in order to implement a T flip-flop, we add a column for D to the state table as shown in Table 5.6. By simply

T	Current State	Next State	D
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

Table 5.6: T flip-flop state table showing the D flip-flop input required to place the T flip-flop in the next state.

looking in the "Next State" column we can see what the input to the D flip-flop must be in order to obtain the correct state. These values are entered in the D column. (We will generalize this design procedure in Section 5.4.)

From Table 5.6 it is easy to write the equation for D:

$$\begin{aligned} D &= T' \cdot Q + T \cdot Q' \\ &= T \oplus Q \end{aligned} \tag{5.16}$$

The resulting design for the T flip-flop is shown in Figure 5.26.

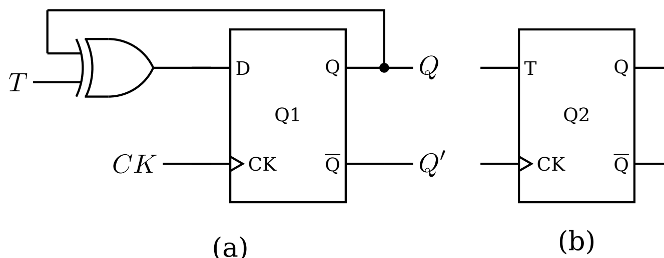
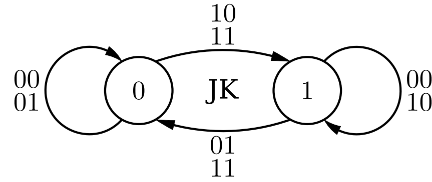


Figure 5.26: T flip-flop. (a) Circuit using a D flip-flop. (b) Symbol for a T flip-flop.

Implementing all four possible actions — set, reset, keep, toggle — requires two inputs, J and K, which leads us to the *JK flip-flop*. The state table and state diagram for a JK flip-flop are shown in Figure 5.27.



J	K	Current State	Next State
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Figure 5.27: JK flip-flop state table and state diagram.

In order to determine the value that must be presented to the D flip-flop we add a column for D to the state table as shown in Table 5.7. shows what values must be input

J	K	Current State	Next State	D
0	0	0	0	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

Table 5.7: JK flip-flop state table showing the D flip-flop input required to place the JK flip-flop in the next state.

to the D flip-flop. From this it is easy to write the equation for D:

$$
\begin{aligned}
D &= J' \cdot K' \cdot Q + J \cdot K' \cdot Q' + J \cdot K' \cdot Q + J \cdot K \cdot Q' \\
&= J \cdot Q' \cdot (K' + K) + K' \cdot Q \cdot (J + J') \\
&= J \cdot Q' + K' \cdot Q
\end{aligned} \tag{5.17}
$$

Thus, a JK flip-flop can be constructed from a D flip-flop as shown in Figure 5.28.

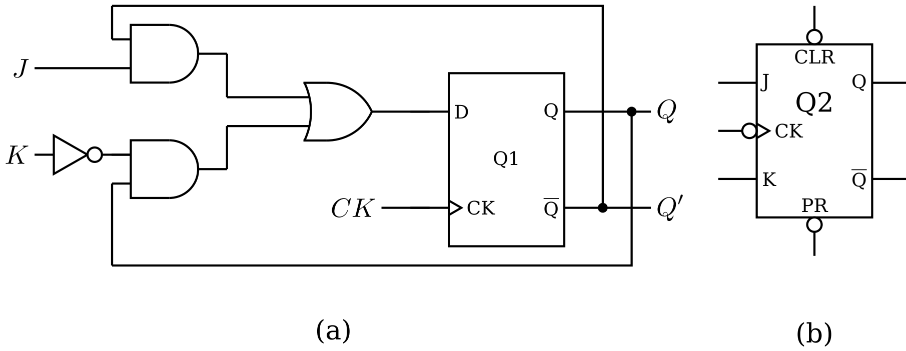


Figure 5.28: JK flip-flop. (a) Circuit using a D flip-flop. (b) Symbol for a JK flip-flop with asynchronous CLR and PR inputs.

5.4 Designing Sequential Circuits

We will now consider a more general set of steps for designing sequential circuits.[1] Design in any field is usually an iterative process, as you have no doubt learned from your programming experience. You start with a design, analyze it, and then refine the design to make it faster, less expensive, etc. After gaining some experience, the design process usually requires fewer iterations.

The following steps form a good method for a first working design:

1. From the word description of the problem, create a state table and/or state diagram showing what the circuit must do. These form the basic technical specifications for the circuit you will be designing.

2. Choose a binary code for the states, and create a binary-coded version of the state table and/or state diagram. For N states, the code will need $\log_2$ bits. Any code will work, but some codes may lead to simpler combinational logic in the circuit.

3. Choose a particular type of flip-flop. This choice is often dictated by the components you have on hand.

4. Add columns to the state table that show the input required to each flip-flop in order to effect each transition that is required.

5. Simplify the input(s) to each flip-flop. Karnaugh maps or algebraic methods are good tools for the simplification process.

6. Draw the circuit.

Example 5-a

Design a counter that has an *Enable* input. When $Enable = 1$ it increments through the sequence 0, 1, 2, 3, 0, 1,... with each clock tick. $Enable = 0$ causes the counter to remain in its current state.

Solution:

[1] I wish to thank Dr. Lynn Stauffer for her valuable suggestions for this section.

8.4 Local Variables on the Call Stack

Now we see that we can store values on the stack by pushing them, and that the push operation *decreases* the value in the stack pointer register, `rsp`. In other words, allocating variables on the call stack involves *subtracting* a value from the stack pointer. Similarly, *deallocating* variables from the call stack involves *adding* a value to the stack pointer.

From this it follows that we can create local variables on the call stack by simply *subtracting the number of bytes required by each variable* from the stack pointer. This does not store any data in the variables, it simply sets aside memory that we can use. (Perhaps you have experienced the error of forgetting to initialize a local variable in C!)

Next, we have to figure out a way to access this reserved data area on the call stack. Notice that there are no labels in this area of memory. So we cannot directly use a name like we did when accessing memory in the `.data` segment.

We could use the `popl` and `pushl` instructions to store data in this area. For example,

```
popl   %eax
movl   $0, %eax
pushl  %eax
```

could be used to store zero in a variable. But this technique would obviously be very tedious, and any changes made to your code would almost certainly lead to a great deal of debugging. For example, can you figure out the reason I had to do a pop before pushing the value onto the stack? (Recall that the four bytes have already been reserved on the stack.)

At first, it may seem tempting to use the stack pointer, `rsp`, as the reference pointer. But this creates complications if we wish to use the stack within the function.

A better technique would be to maintain another pointer to the local variable area on the stack. If we do not change this pointer throughout the function, we can always use the *base register plus offset* addressing mode to directly access any of the local variables. The syntax is:

offset(*register_name*)

Intel® Syntax | [*register_name* + *offset*]

When it is zero, the offset is not required.

base register plus offset: The data value is located in memory. The address of the memory location is the sum of a value in a register plus an offset value, which can be an 8-, 16- or 32-bit signed integer.

syntax: place parentheses around the register name with the offset value immediately before the left parenthesis.

examples: `-8(%rbp)`; `(%rsi)`; `12(%rax)`

Intel® Syntax | `[rbp - 8]`; `[rsi]`; `[rax + 12]`

The appropriate register for implementing this is the frame pointer, `rbp`.

When a function is called, the calling function begins the process of creating an area on the stack, called the *stack frame*. Any arguments that need to be passed on the call stack are first pushed onto it, as described in Section 11.2. Then the `call` instruction pushes the return address onto the call stack (page 173).

The first thing that the called function must do is to complete the creation of the stack frame. The function prologue, first introduced in Section 7.2 (page 148), performs the following actions at the very beginning of each function:

1. Save the caller's value in the frame pointer on the stack.
2. Copy the current value in the stack pointer to the frame pointer.
3. Subtract a value from the stack pointer to allow for the local variables.

Once the function prologue has completed the stack frame, we observe that:

- The local variables are located in an area of the call stack – between the addresses in the `rsp` and `rbp` registers.
- The `rbp` register is a pointer to the bottom (the numerically highest address) of the local variable area.
- The remaining area of the stack can be accessed using the stack pointer (`rsp`) as always.

Notice that each local variable is located at some fixed offset from the base register, `rbp`. In fact, it's a negative offset.

Listing 8.5 is the compiler-generated assembly language for the program in Listing 2.4 (page 25). Comments have been added to explain the parts of the code being discussed here.

```
1          .file   "echoChar1.c"
2          .section        .rodata
3  .LC0:
4          .string "Enter one character: "
5  .LC1:
6          .string "You entered: "
7          .text
8          .globl  main
9          .type   main, @function
10 main:
11         pushq   %rbp              # save caller's frame pointer
12         movq    %rsp, %rbp        # establish our frame pointer
13         subq    $16, %rsp         # space for local variable
14         movl    $21, %edx         # 21 characters
15         movl    $.LC0, %esi       # address of "Enter ... "
16         movl    $1, %edi          # STDOUT_FILENO
17         call    write
18         leaq    -1(%rbp), %rax    # address of aLetter var.
19         movl    $1, %edx          # 1 character
20         movq    %rax, %rsi        # address in correct reg.
21         movl    $0, %edi          # STDIN_FILENO
22         call    read
23         movl    $13, %edx         # 13 characters
24         movl    $.LC1, %esi       # address of "You ... "
25         movl    $1, %edi          # STDOUT_FILENO
26         call    write
27         leaq    -1(%rbp), %rax    # address of aLetter var
28         movl    $1, %edx          # 1 character
29         movq    %rax, %rsi        # address in correct reg.
```

```
30        movl    $1, %edi          # STDOUT_FILENO
31        call    write
32        movl    $0, %eax          # return 0;
33        leave                     # undo stack frame
34        ret                       # back to caller
35        .size   main, .-main
36        .ident  "GCC: (Ubuntu/Linaro 4.7.0-7ubuntu3) 4.7.0"
37        .section        .note.GNU-stack,"",@progbits
```

Listing 8.5: Echoing characters entered from the keyboard (gcc assembly language). Comments added. Refer to Listing 2.4 for the original C version.

The function begins by pushing a copy of the caller's frame pointer (in the rbp register) onto the call stack, thus saving it. Next it sets the frame pointer for this register at the current top of the stack. These two actions establish a reference point to the stack frame for this function.

Next the program allocates sixteen bytes on the stack for the local variable, thus growing the stack frame by sixteen bytes. It may seem wasteful to set aside so much memory since the only variable in this program requires only one byte of memory, but the ABI [25] specifies that the stack pointer (rsp) should be on a sixteen-byte address boundary before calling another function. The easiest way to comply with this specification is to allocate memory for local variables in multiples of sixteen.

Figure 8.5 shows the state of the stack just after the prologue has been executed. The return address to the calling function is safely stored on the stack, followed by the

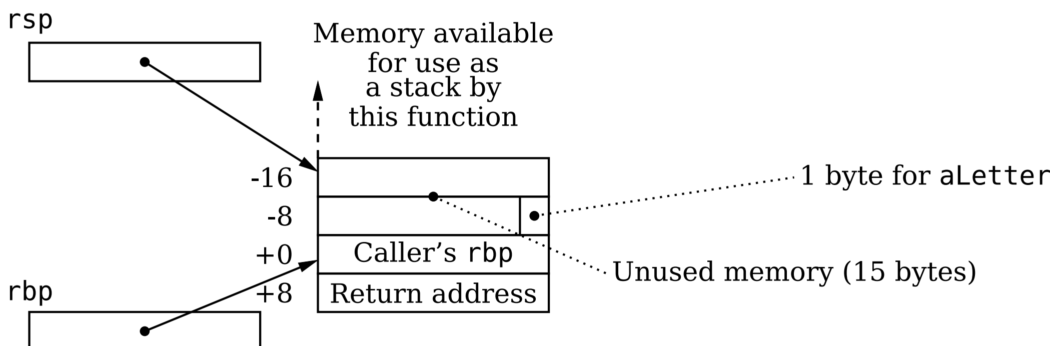


Figure 8.5: Local variables in the program from Listing 8.5 are allocated on the stack. Numbers on the left are offsets from the address in the frame pointer (rbp register).

caller's frame pointer value. The stack pointer (rsp) has been moved up the stack to allow memory for the local variable. If this function needs to push data onto the stack, such activity will not interfere with the local variable, the caller's frame pointer value, nor the return address. The frame pointer (rbp) provides a reference point for accessing the local variable.

> *IMPORTANT:* The space for the local variables must be allocated *immediately after establishing the frame pointer.* Any other use of the stack within the function, e.g., saving registers, must be done *after* allocating space for local variables.

Most of the code in the body of the function is already familiar to you, but the instruction that loads the address of the local variable, aString into the rax register:

```
18        leaq    -1(%rbp), %rax # address of aLetter var.
```

is new. It uses the base register plus offset addressing mode for the source. We can see from the instruction on line 18 that the `aString` variable is located one byte negative from the address in the `rbp` register. Since the call stack grows toward negative addresses, this is the next available byte in this function's stack frame.

As with the `write` function, the second argument to the `read` function must be the address of a variable. However, the address of `aString` cannot be known when the program is compiled and linked because it is the address of a variable that exists in the stack frame. There is no way for the compiler or linker to know where this function's stack frame will be in memory when it is called. The address of the variable must be computed at run time.

Each instruction that accesses a stack frame variable must compute the variable's address, which is called the *effective address*. The instruction for computing addresses is *load effective address* — `leal` for 32-bit and `leaq` for 64-bit addresses. The syntax of the `lea` instruction is

`lea`*w* *source*, *%register*

where *w* = `l` for 32-bit, `q` for 64-bit.

Intel® Syntax | `lea` *register*, *source*

The source operand must be a memory location. The `lea` instruction computes the effective address of the source operand and stores that address in the destination register. So the instruction

```
leaq    -1(%rbp), %rax
```

takes the value in `rbp` (the base address of this function's stack frame), adds `-1` to it, and stores this sum in `rax`. Now `rax` contains the address of the variable `aLetter`. (The address still needs to be moved to `rsi` for the call to the `read` function.)

So the following code sequence:

```
18        leaq    -1(%rbp), %rax  # address of aLetter var.
19        movl    $1, %edx        # 1 character
20        movq    %rax, %rsi      # address in correct reg.
21        movl    $0, %edi        # STDIN_FILENO
22        call    read
```

implements the C statement

```
14    read(STDIN_FILENO, &aLetter, 1);                   // one character
```

in the original C program (Listing 2.4, page 25). (Yes, it would have been more efficient to use `rsi` as the destination for the `leaq` instruction. Recall that this program was compiled with the `-O0` option, no optimization. You can also expect this to vary across different versions of the compiler.)

Some notes about the `read` function call:

- The characters read from the keyboard must be stored in memory. You cannot pass the name of a CPU register to the `read` function.
- The number of bytes actually read from the keyboard is returned in the eax register. So if the current function is using `eax`, the value will be changed by the call to `read`.
- The `read` function is a C wrapper that sets up the registers for the `syscall` instruction. Unfortunately, there is no guarantee that it restores the values that were in the registers when it was called.

> *IMPORTANT:* Since neither the `write` nor the `read` system call functions are guaranteed to restore the values in the registers, your program must save any required register values before calling either of these functions.

There is also a new instruction on line 33:

```
33         leave                        # undo stack frame
```

Just before this function exits the portion of the stack frame allocated by this function must be released and the value in the `rbp` register restored. The `leave` instruction performs the actions:

```
        movq    %rbp, %rsp
        popq    %rbp
```

which effectively

1. deletes the local variables
2. restores the caller's frame pointer value

After the epilogue has been executed, the stack is in the state shown in Figure 8.6. The stack pointer (`rsp`) points to the address that will return program flow back to the

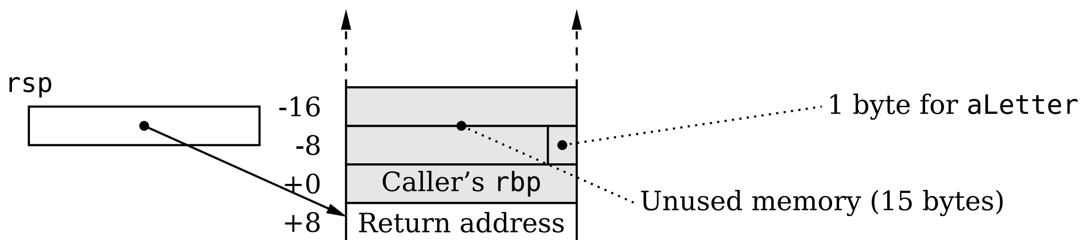


Figure 8.6: Local variable stack area in the program from Listing 8.5. Although the values in the gray area may remain they are invalid; using them at this point is a programming error.

instruction immediately after the `call` instruction that called this function. Although the data that was stored in the memory which is now above the stack pointer is still there, it is a violation of stack protocol to access it.

One more step remains in completing execution of this function — returning to the calling function. Since the return address is at the top of the call stack, this is a simple matter of popping the address from the top of the stack into the `rip` register. This requires a special instruction,

```
        ret
```

which does not require any arguments.

Recall that there are two classes of local variables in C:

Automatic variables are created when the function is first entered. They are deleted upon exit from the function, so any value stored in them during execution of the function is lost.

Static variables are created when the program is first started. Any values stored in them persist throughout the lifetime of the program.

Most local variables in a function are automatic variables. General purpose registers are used for local variables whenever possible. Since there is only one set of general purpose registers, a function that is using one for a variable must be careful to save the value in the register before calling another function. Register usage is specified by the ABI [25] as shown in Table 6.4 on page 134. But you should not write code that depends upon everyone else following these recommendations, and there are only a small number of registers available for use as variables. In C/C++, most of the automatic variables are typically allocated on the call stack. As you have seen in the discussion above, they are created (automatically) in the prologue when the function first starts and are deleted in the epilogue just as it ends. Static variables must be stored in the data segment.

We are now in a position to write the echoChar program in assembly language. The program is shown in Listing 8.6.

```
1  # echoChar2.s
2  # Prompts user to enter a character, then echoes the response
3  # Bob Plantz - 8 June 2009
4
5  # Useful constants
6          .equ    STDIN,0
7          .equ    STDOUT,1
8  # Stack frame
9          .equ    aLetter,-1
10         .equ    localSize,-16
11 # Read only data
12         .section  .rodata
13 prompt:
14         .string "Enter one character: "
15         .equ    promptSz,.-prompt-1
16 msg:
17         .string "You entered: "
18         .equ    msgSz,.-msg-1
19 # Code
20         .text                   # switch to text section
21         .globl  main
22         .type   main, @function
23 main:
24         pushq   %rbp            # save caller’s frame pointer
25         movq    %rsp, %rbp      # establish our frame pointer
26         addq    $localSize, %rsp  # for local variable
27
28         movl    $promptSz, %edx # prompt size
29         movl    $prompt, %esi   # address of prompt text string
30         movl    $STDOUT, %edi   # standard out
31         call    write           # invoke write function
32
33         movl    $2, %edx        # 1 character, plus newline
34         leaq    aLetter(%rbp), %rsi # place to store character
35         movl    $STDIN, %edi    # standard in
36         call    read            # invoke read function
37
38         movl    $msgSz, %edx    # message size
39         movl    $msg, %esi      # address of message text string
40         movl    $STDOUT, %edi   # standard out
```