

```

37
38     movl    $msgSz, %edx    # message size
39     movl    $msg, %esi     # address of message text string
40     movl    $STDOUT, %edi   # standard out
41     call    write          # invoke write function
42
43     movl    $2, %edx        # 1 character, plus newline
44     leaq    aLetter(%rbp), %rsi # place where character stored
45     movl    $STDOUT, %edi   # standard out
46     call    write          # invoke write function
47
48     movl    $0, %eax        # return 0
49     movq    %rbp, %rsp      # delete local variables
50     popq    %rbp           # restore caller's frame pointer
51     ret                    # back to calling function

```

Listing 8.6: Echoing characters entered from the keyboard (programmer assembly language).

This program introduces another assembler directive (lines 6,7,9,10,15,18):

`.equ name, expression`

The `.equ` directive evaluates the *expression* and sets the *name* equivalent to it. Note that the *expression* is evaluated during assembly, not during program execution. In essence, the *name* and its value are placed on the symbol table during the first pass of the assembler. During the second pass, wherever the programmer has used “*name*” the assembler substitutes the number that the *expression* evaluated to during the first pass.

You see an example on line 9 of Listing 8.6:

```

9     .equ    aLetter, -1

```

In this case the expression is simply -1. Then when the symbol is used on line 34:

```

34     leaq    aLetter(%rbp), %rsi # place to store character

```

the assembler substitutes -1 during the second pass, and it is exactly the same as if the programmer had written:

```

leaq    -1(%rbp), %rsi # place to store character

```

Of course, using `.equ` to provide a symbolic name makes the code much easier to read.

An example of a more complex expression is shown on lines 13 – 15:

```

13 prompt:
14     .string "Enter one character: "
15     .equ    promptSz, .-prompt-1

```

The “.” means “this address”. Recall that the `.string` directive allocates one byte for each character in the text string, plus one for the NUL character. So it has allocated 22 bytes here. The expression computes the difference between the beginning and the end of the memory allocated by `.string`, minus 1. Thus, `promptSz` is entered on the symbol table as being equivalent to 21. And on line 28 the programmer can use this symbolic name,

```

28     movl    $promptSz, %edx # prompt size

```

which is much easier than counting each of the characters by hand and writing: