CS220 Library User's Manual

May 17, 2007

Introduction

The CS220 library contains several routines that simplify input and output from assembly language programs. The included routines are listed in Table 1.

Examples

Assume we have the following definitions in the data section:

```
.section .data
msg1: .asciz "Hello world!"
msg2: .asciz "Enter an integer: "
msg3: .asciz "Enter a line of text"
buff: .fill 80
xval: .int 12345
yval: .int 0
```

The following code illustrates the use of several of the routines in the library.

```
# Display "Hello world!" message
movl    $msg1, %eax
call    print_str
call    print_nl

# Display the value of the xval variable
movl    xval, %eax
call    print_int
call    print_nl

# Prompt for an integer, store in yval variable
movl    $msg2, %eax
call    print_str
call    read_int
movl    %eax, yval
call    read_nl

# Prompt for a line of text, store at location buff
# Display the line of text.
movl    $msg3, %eax
call    print_str
call    print_nl
movl    $buff, %eax
call    read_line
movl    $buff, %eax
call    print_str
call    print_nl
```

Note that, in AT&T syntax, **$label** is an address while **label** is a memory reference, so “movl xval, %eax” would move the value stored at memory location **xval** into register **EAX**, while “movl $xval, %eax” would move the address of **xval** into register **EAX**.

Notes

1. All of the routines use the C stdio routines for input/output.

2. All of the output routines (**print_char**, **print_int**, **print_uint**, **print_int**, **print_single**, **print_double**, **print_str**, **print_nl**) flush the stdout buffer after calling the stdio output routine. Although this is not efficient, it ensures that output appears on the screen immediately. Beginning programmers usually expect output to appear immediately when an output routine is used and troubleshooting efforts via instrumentation (using print statements) can be stymied if this is not the case.

3. The **read_int**, **read_uint**, **read_single**, **read_double**, and **read_str** routines skip over any leading whitespace (space, tabs, line feeds, carriage returns, etc) while the **read_char** and **read_line** routines do not. However, **read_int**, **read_uint**, **read_single**, **read_double**, and **read_str** leave trailing whitespace in the input stream. This can cause unexpected behavior when mixing calls that skip whitespace with those that do not. For example, using **read_int** to read an integer at the end of a line and then calling **read_line** will actually read what is left in the line after the integer (perhaps nothing) and not the next line. Use **read_nl** after the call to **read_int** and before the call to **read_line** to read the whitespace (including the line feed) after the integer. The call to **read_line** will then read the next line.

Usage

An assembly program to be linked with the library should have the following general form:

```
.globl _asm_main

# Initialized data goes in the data segment
.section .data

# Uninitialized data goes in the bss segment
.section .bss

# Code goes in the text segment
.section .text
_asm_main:
     enter     $0, $0          # set up stack frame
     pusha                     # save all registers

     # User-written code goes here.

     popa                      # restore all registers
     movl      $0, %eax        # return program status in eax
     leave                     # restore stack frame
     ret
```

The name of the entry point must be **_asm_main** and **_asm_main** must be declared global as shown. This code is in the **skeleton.s** file that is in the library archive.

To link assembly file **foobar.s** with the library (**libcs220.a**) invoke **g++** as follows:

```
g++ -o foobar foobar.s -LLIBDIR -lcs220
```

where **LIBDIR** should be replaced by the path to the directory containing the library (use a single period to represent the current directory).

The library includes a C++ **main** program that calls **_asm_main**. If you want to link your own C++ **main** programs with an assembly routine that calls I/O routines in the CS220 library you will need to use the following compilation options:

```
g++ -o foobar -Wl,--defsym -Wl,_asm_main=0  \
    -Wl,--allow-multiple-definition foo.cpp bar.s \
    -LLIBDIR -lcs220
```

The options after "-Wl," (that's W-ell, not W-one) are passed to the linker and are necessary to prevent link errors due to the multiple definitions of **main** (and presumably an undefined **_asm_main** routine). In a **Makefile** this can be accomplished more neatly using:

```
LDOPTSA=-Wl,--defsym -Wl,_asm_main=0
LDOPTS=$(LDOPTSA) -Wl,--allow-multiple-definition

foobar: foo.cpp bar.s
     g++ -o foobar $(LDOPTS) foo.cpp bar.s -LLIBDIR -lcs220
```

Name	*Input Register*	*Output Register*	*Description*
print_char	%eax		Sends an ASCII character stored in register EAX to standard output. The character should be stored in the lowest byte of the EAX register. The upper three bytes should contain zeros.
print_int	%eax		Converts a 32 bit signed integer stored in register EAX to ASCII and sends the result to standard output.
print_uint	%eax		Converts a 32 bit unsigned integer stored in register EAX to ASCII and sends the result to the standard output.
print_single print_double	%st(0)		Converts a 32/64 bit float stored in register ST(0) to ASCII and sends the result to the standard output.
print_singleptr print_doubleptr	%eax		Converts a 32/64 bit float stored at address in EAX to ASCII and sends the result to the standard output.
print_str	%eax		Sends a null-terminated C string (ASCII char array) whose address is in EAX to standard output.
print_nl			Sends an ASCII line feed character to standard output.
read_char		%eax	Returns a character read from standard input in register EAX. The character is returned in the lowest byte of EAX, the upper three bytes are set to zero.
read_int		%eax	Converts an ASCII representation of an integer read from standard input to a signed 32 bit integer and returns the result in register EAX. Leading whitespace is skipped.
read_uint		%eax	Converts an ASCII representation of a positive integer read from standard input to an unsigned 32 bit integer and returns the result in register EAX. Leading whitespace is skipped.
read_single read_double		%st(0)	Converts an ASCII representation of a float read from standard input to a 32/64 bit float and returns the result in ST(0). Leading whitespace is skipped.
read_singleptr read_doubleptr	%eax		Converts an ASCII representation of a float read from standard input to a 32/64 bit and stores the result at the address in register EAX. Leading whitespace is skipped.
read_str	%eax		Reads a white-space delimited string (a word) from standard input and stores it at the address in register EAX. The string is null terminated. Leading whitespace is skipped.
read_line	%eax		Reads a linefeed terminated line from standard input and stores it at the address in register EAX. The string is null terminated. The linefeed is removed from the input buffer. Reads in a maximum of 255 characters. The buffer should be 256 bytes in length (255 characters plus the trailing null character). The buffer may be shorter, but then buffer overflow may occur if the input line contains more characters than the buffer can hold.
read_nl			Reads and discards all characters up to and including the next linefeed.
str_length	%eax	%eax	Determines the length of a null-terminated C string (ASCII char array) whose address is in EAX. The null-terminating byte is not included in the length value. The length is returned in the EAX register.

Table 1: Routines in the CS220 Assembly Library

Preprocessing and Debugging
Assembly source code files that have a .S (uppercase s) suffix are automatically run through the C preprocessor when assembled using either **gcc** or **g++**. Source code files that have a .s (lowercase s) are assumed to be pure assembly and may not contain preprocessor directives. The use of preprocessor definitions in assembly can result in programs that are easier to read and maintain. They are particularly useful when using local variables in an assembly subprogram. For example:

```
#define   X_INT     -4(%ebp)
#define   Y_DBL     -12(%ebp)
```

The X_INT and Y_DBL labels can then be used throughout the assembly source code file.

There is a slight problem when trying to debug assembly-with-preprocessor source code. The preprocessor creates a pure assembly source code file in the /tmp directory that is then run through the assembler. When the debugger is started it looks for an assembly source file in the /tmp directory (which has since been deleted) and not the original assembly-with-preprocessor source file. The solution is to either use pure assembly when wanting to use the debugger (use a lowercase s suffix for the assembly source code file) or to break down preprocessing and assembly into two separate stages. I prefer to use the asm suffix on files that contain assembly-with-preprocessor source code. (This prevents problems on case insensitive file systems.) To create an object file with debugging information from a source file named **foo.asm** then do the following:

```
cpp -P -x assembler-with-cpp foo.asm foo.s
g++ -g -c foo.s
```

The file **foo.s** will then serve as the source code file for the debugger. It should be similar enough to the original **foo.asm** file so that debugging is not too confusing.