**Lab 11 – “Your own printf() function”**

### 1. Grading

**1.1. Test Cases**

The zip file that you download contains a Makefile and two test files, main.c and twelvedays.c, that will call your myprintf.c implementation to check its functionality, and mainout and twelvedaysout that contain a copy of the output that should be produced by main.c and twelvedays.c in conjunction with your programs.

### 1.2. Turnin

Create a ~/cs250/lab11-src directory on your Pi and place all of your work in it.

You should have two files:

myprintf.c, which implements the myprintf() function.

printintegers.s, which implements printx() and printd() and is written in ARM assembly.

Before your lab session starts, submit your lab11-src directory electronically on **data.cs.purdue.edu** using the turnin command:

$ turnin -c cs250 -p lab11 lab11-src

$ turnin -c cs250 -p lab11 -v

Remember to submit from data.cs.purdue.edu.

### Introduction

“Print to the terminal” is among the most basic functions that an operating system provides. Before graphical user interfaces, people interacted with computers via the terminal – and only the terminal. In many cases, terminals were able to display only ASCII characters.

Print to the terminal functionality begins with the ability to print a single character. Then, to print multiple characters requires only calling this mechanism multiple times. Such a mechanism is computer specific and part of the operating system.

For this lab we will use the putchar() function every time we have a single character to print to the terminal. Putchar() is the only output function that you may call in this lab. putchar('a') prints the character “a” on the screen.

### 3. Variadic Functions

We often use the function printf() like this:

printf("Elapsed time clocks: %d\n", clock() - time\_start);

Sometimes we may pass many arguments to it:

// It's really hard to debug OS code

// so debug printf's are everywhere.

printf("The process selected for execution is #%d: '%s' with a priority of %d\n", currpid, prptr->prname, prptr→prprio);

Interestingly, printf() can accept an arbitrary number of arguments. This means that printf() is an example of a **variadic function**.

The function prototype for printf() is specified as follows: (you can see it on the man page by typing “man 3 printf” in the terminal):

int printf(const char \*format, ...);

The first argument – format – is always a string literal. The remaining list of arguments, of which there can be zero or more, is specified as “...” These are the values to be substituted for each format specifier contained in the format string literal. Showing three periods as an argument is the way to specify that a function is variadic.

# **3.1 Arguments on the Stack**

The <stdarg.h> header file helps with variadic functions by declaring the va\_list type as well as parsing functions. Do not use it. You are to come up with your own solution.

The first question to answer when implementing printf() is how do you reference the second argument and beyond? There is no explicit variable named for each, so we will have to directly reference their values on the stack. To do so, we must leverage a key aspect of the calling convention: arguments that are passed to a function are pushed onto the stack in reverse order, that is, the last argument is the first to be pushed on the stack.

Thus, the first argument for printf(), the format string liters, will be the last argument pushed onto the stack. Recall also that new arguments are pushed on to the stack at successively smaller addresses. This means the format string will be located at the *lowest numbered address* while the other arguments of printf() will occupy successively higher addresses in the order in which they appear. Put another way, when an item is pushed onto the stack, the address stored in the stack pointer register, sp, decreases in magnitude. When an item is popped, the address in sp increases. Once we know the location of the first argument, we know the starting point for all arguments.

Constant (fixed value) string arguments, such as the format string for the printf() function, are not stored in the stack. Instead, they are stored in the data segment of a program, which lies far from the stack in much lower memory addresses near the text segment. So, what is pushed onto the stack for a static string is not its value, but a pointer to the start of the string in the data segment.

You may have noticed that this argument passing convention disagrees with the ARM convention from an earlier lab. For that ARM convention, the first four arguments were passed via registers r0 to r3. Any additional arguments were passed via the stack. This convention leads to inconsistent handling of arguments for variadic functions, however. Therefore, the C compiler places all arguments for a variadic function on the stack. As an aside, the C compiler will also place all arguments on the stack if at any point the program refers to an argument’s address.

### 3.2 Format Specifiers

The format string used by printf() may contain place holders, called format specifiers, that begin with the character ‘%’:

* **%c** is the place holder for a character in the arguments.
* **%s** represents a character string ending in \0.
* **%x** is a placeholder for an integer to be printed in hexadecimal.
* **%d** represents an integer to be printed in decimal.
* **%f** represents the place for a floating point number.

The standard printf() function supports many more specifiers, including more complicated ones like %5.2f that displays a decimal floating point number (a float) using a minimum field width of a total of 5 characters with two decimal digits to the right of the decimal point and zero digits if needed to pad to the minimum width. We will ignore such complex, compound specifiers for this lab.

For this lab, we will implement the following format specifiers and an escape character:

* **c**, single character
* **s**, character string
* **x**, hexadecimal
* **d**, decimal
* **%**, if the character following the % directive is again a %, print one % only.

You may assume that no “illegal” directives will be present in the format string. A compiler would terminate its work if an illegal format directive for printf() were found, preventing illegal directives in actual use.

**4. Implementation**

You should name your print to the terminal function **myprintf()**, so it does not conflict with the standard printf(). The prototype for your function is:

int myprintf(const char \* format, ...);

Your function should locate the first argument on the stack, iterate through the individual characters in the format string and, using putchar(), output every character that is not a format specifier. When a format specifier is encountered, that is, a percent (%), you should locate the next argument on the stack and, again using putchar() for each character, output the argument after converting it from the representation within your Pi circuitry, as necessary, to a sequence of ASCII characters for display on the terminal according to the format specifier. If two consecutive % symbols are encountered (%%) this means that the % symbol is being escaped from its usual meaning as the start of a format specifier so that the % symbol itself is possible to print to the terminal. In this case, the argument is not on the stack, and you should simply use putchar() to output a single percent sign.

### 4.1 Character and string formats

After %%, %c is the simplest format specifier. For %c, just find the corresponding argument on the stack and pass it to putchar().

Printing a character string – %s – is a slight extension to %c. Find the corresponding pointer argument on the stack, and starting at that address, repeatedly call putchar() on each successive character until you encounter a value of ‘\0’.

**4.2 Hexadecimal format**

# For the %x format specifier, it is expected that the argument will be displayed as a sequence of ASCII characters that represent hexadecimal notation corresponding to the argument bit string. For this part, write a function, printx(), in ARM assembly that takes a 32-bit integer as the argument and generates the appropriate ASCII character sequence. You can then call printx() from your myprintf() C function when you encounter the %x format specifier.

# **Note: W**hen outputting the hexadecimal equivalent for an integer, assume that leading 0s should be omitted. printf() supports padding with leading zeros, but we will not be implementing support for this.

Again, you may use the putchar() function from the standard C library. Do not use use any global variables in your printx() implementation.

Notice that the hexadecimal-represented number may have anywhere between 1 and 8 digits. This means that when parsing, starting with the MSB could result in the output of leading zeros. Instead, start with the LSB.

First, think about which pair of ARM logical operations can be used together to clear (set to 0) all but the rightmost 4 bits of the provided integer. Shifting in zeros is a way to clear bits in a register. Then, think about how to convert the resulting integer, call it Z, into the ASCII character code corresponding to the representation of Z within the 16-symbol hexadecimal representation system, {0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f}. Keep in mind that putchar() will interpret the argument passed to it using ASCII encoding. If you call putchar(9), the terminal will not display a “9” character. Instead, a tab character will be printed, per the ASCII encoding.

Finally, consider where store the ASCII character you generate until it is time to retrieve it for use as an argument for putchar(). Registers are not a good choice for storing these ASCII characters because the integer will have an unknown number of digits. The expandable stack in main memory is a good choice. push and pop instructions will be extremely helpful because the last ASCII character pushed onto the stack will be the first one popped off to be printed.

You could also permit printx() to take a second argument to specify which case to use. In that case you could also implement the %X format specifier.

# **4.3 Decimal integer format and the need to divide by 10**

Write a function, printd(), in ARM assembly to convert to decimal integer format. Again, the function should take a single argument, a 32-bit signed integer, and display this argument in decimal format. You may use the putchar() function from the standard C library. You may not use any global variables in printd().

Printing an integer in decimal form is similar to hexadecimal, but there is another complication: signed integers are stored in 2’s complement format. Unlike hexadecimal, there is no straightforward way to get the decimal representation just by looking at groupings of bits.

Inevitably you will end up performing a division operation on the integer. To obtain the least significant digit of the decimal representation, divide the integer by 10 and get the remainder. If the quotient is larger than zero, divide the quotient by 10 to obtain a new remainder, which represents the next more significant digit. Repeat as long as the quotient is still larger than zero.

There is no native instruction for integer division in ARMv7. There are some pseudo-instructions given in the ARM manual, but they are relatively complicated and represent a set of instructions that unpack the division operation using a loop.

Fortunately, because the divisor for our purposes is always 10, we can use a mathematical relationship discovered long ago for dividing by a constant. There are many articles that discuss this approach. One of them can be found [here](http://thinkingeek.com/2013/08/11/arm-assembler-raspberry-pi-chapter-15/) under “Division by a constant integer.” There is an associated [tool](https://github.com/rofirrim/raspberry-pi-assembler/blob/master/chapter15/magic.py) to generate the instructions to perform the division as well. Copy the python code to a file on your Pi, and invoke it using “python <filename> <arguments>”. Make sure that you understand how to use the code before plugging it in to your assembly code for printintegers.s (see Turnin below).

**2.4 Corner Cases**

You should now have enough information to implement myprintf(). Keep in mind the following corner cases:

* What if the integer to print is 0?
* What if the integer to print is negative? Remember, the most significant bit of the 2’s complement representation will be 1.