Computer Organization
Input/Output and linking to C libraries

Prof. Charles W. Kann

Overview of Class

- **Why use C libraries for I/O**
- **Writing HelloWorld**
- **Prompt for, retrieve,and print a string**
- **Prompt for, retrieve, and print an int**
  - With a static variable
  - Using a stack variable
- **Makefile and touch**

Why use a C libraries for I/O

- System Call using R7 only allows binary read/write. Generic data type reads and writes are needed.
- C has a library of I/O including the scanf and printf functions that allow reading and writing of most data types needed for programming.
- Learning how to access C libraries from assembly (and assembly libraries from C) is a useful skill to most programmers.
- Libraries that will be used in this module for I/O are scanf and printf.

printf

- **The printf function takes two parameters**
  - A format string, prints string
  - A pointer to a va_list. Don't worry about this for now, as printf will only be used for a single output parameter, and the format can be copied.
  - http://www.cplusplus.com/reference/cstdio/printf/
  - Examples:
    - printf("Is %d your number?\n", 5) – Prints "Is 5 your number?"
    - printf("Is %s your name?\n", "Chuck") – Prints "Is Chuck your name?

Format specifiers

A *format specifier* follows this prototype: [see compatibility note below]
`%[flags][width][.precision][length]specifier`

Where the *specifier character* at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

specifier	**Output**	**Example**
`d` *or* `i`	Signed decimal integer	`392`
`u`	Unsigned decimal integer	`7235`
`o`	Unsigned octal	`610`
`x`	Unsigned hexadecimal integer	`7fa`
`X`	Unsigned hexadecimal integer (uppercase)	`7FA`
`f`	Decimal floating point, lowercase	`392.65`
`F`	Decimal floating point, uppercase	`392.65`
`e`	Scientific notation (mantissa/exponent), lowercase	`3.9265e+2`
`E`	Scientific notation (mantissa/exponent), uppercase	`3.9265E+2`
`g`	Use the shortest representation: `%e` or `%f`	`392.65`
`G`	Use the shortest representation: `%E` or `%F`	`392.65`
`a`	Hexadecimal floating point, lowercase	`-0xc.90fep-2`
`A`	Hexadecimal floating point, uppercase	`-0XC.90FEP-2`
`c`	Character	`a`
`s`	String of characters	`sample`
`p`	Pointer address	`b8000000`
`n`	Nothing printed. The corresponding argument must be a pointer to a `signed int`. The number of characters written so far is stored in the pointed location.	
`%`	A `%` followed by another `%` character will write a single `%` to the stream.	`%`

Control Characters

Escape sequence	Hex value in ASCII	Character represented
\a	07	Alert (Beep, Bell) (added in C89)[1]
\b	08	Backspace
\e[note 1]	1B	Escape character
\f	0C	Formfeed Page Break
\n	0A	Newline (Line Feed); see notes below
\r	0D	Carriage Return
\t	09	Horizontal Tab
\v	0B	Vertical Tab
\\	5C	Backslash
\'	27	Apostrophe or single quotation mark
\"	22	Double quotation mark
\?	3F	Question mark (used to avoid trigraphs)
nnn[note 2]	any	The byte whose numerical value is given by *nnn* interpreted as an octal number
\x*hh*...	any	The byte whose numerical value is given by *hh*... interpreted as a hexadecimal number
\u*hhhh*[note 3]	none	Unicode code point below 10000 hexadecimal
\U*hhhhhhhh*[note 4]	none	Unicode code point where *h* is a hexadecimal digit

scanf

- **Two parameters are passed to the scanf function**
  - The address of an input format string
  - The address of where to store the value that is entered by the user
- **Valid input types for scanf are the same as for the printf**

Writing HelloWorld

- To use the C functions, the gcc (GNU C) compiler will be used.
- The gcc linker will look for a program beginning at the global symbol *main*, not *_start*.
- When writing a program using gcc, the *main* is called as a method, not as the starting point in the program. So there are 2 lines at the beginning and 2 lines at the end of the program that are used to interface with the program stack. For now, just include these lines, they will be explained in a later module.
- The file *Template.s* shows a template for a program. This program is a template, and does nothing.

Template.s

```
# Template.s
   .text
   .global main

main:
# Save return to os on stack
   sub sp, sp, #4
   str lr, [sp, #0]

# Return to the OS
   ldr lr, [sp, #0]
   add sp, sp, #4
   mov pc, lr

.data
```

Compiling and linking Template

- To compile and link the Template.s source program into the executable program Template, only one call to gcc needs to be used. This call still compiles to an object file and calls the linker to create the executable. But this is hidden.
- We will do the individual steps later to examine the machine code produced, but for now just compile and link in one step.

```
All: Template

Template: Template.s
	gcc $@.s -g -o $@   # this line must start with a tab
```

Using C IO functions in ARM assembly

- **How to use the CIO functions in ARM assembly will be covered in the next four slides. The slides are:**
  - IOExample_1.s - Implements HelloWorld program using printf.
  - IOExample_2.s – Implements a program to read/write a string using scanf and printf.
  - IOExample_3.s – Implements a program to read/write an int using scanf and printf. The program uses data memory to store the value
  - IOExample_4.s – Implements a program to read/write an int using scanf and printf. The program uses stack memory to store the value.

The final Makefile for all of the programs

```
All: Template IOExample_1 IOExample_2 IOExample_3 IOExample_4

Template: Template.s
	gcc $@.s -g -o $@     # Note that all lines with gcc must start with a tab.

IOExample_1: IOExample_1.s
	gcc $@.s -g -o $@

IOExample_2: IOExample_2.s
	gcc $@.s -g -o $@

IOExample_3: IOExample_3.s
	gcc $@.s -g -o $@

IOExample_4: IOExample_4.s
	gcc $@.s -g -o $@
```

The final Makefile for all of the programs

```
All: Template IOExample_1 IOExample_2 IOExample_3 IOExample_4

Template: Template.s
	gcc $@.s -g -o $@     # Note that all lines with gcc must start with a tab.

IOExample_1: IOExample_1.s
	gcc $@.s -g -o $@

IOExample_2: IOExample_2.s
	gcc $@.s -g -o $@

IOExample_3: IOExample_3.s
	gcc $@.s -g -o $@

IOExample_4: IOExample_4.s
	gcc $@.s -g -o $@
```

IOExample_1.s – printf function

```
# IOExample_1.sp
# Example of using printf
   .text
   .global main
main:
# Save return to os on stack
   sub sp, sp, #4
   str lr, [sp, #0]

# Printing The Message
   ldr r0, =HelloWorld
   bl  printf

# Return to the OS
   ldr lr, [sp, #0]
   add sp, sp, #4
   mov pc, lr

.data
HelloWorld:
   .asciz "Hello World\n"
```

IOExample_1.s explanation

- **The only code in this program not previously explained is at the bottom of this slide.**
- **In this code:**
  - r0 is loaded with the address of the string at label HelloWorld
  - The *bl* operator executes a *branch-and-link* operation, executing the printf function.
  - Do not worry about the addressing for the string parameter or specifics of the bl instruction for now, they will be covered later.

```
# Printing The Message
   ldr r0, =HelloWorld
   bl  printf
```

IOExample_2.s – scanf function with a string

```
#IOExample_2.s – Shows scanf function
   .text
   .global main
main:
# Save return to os on stack
   sub sp, sp, #4
   str lr, [sp, #0]
# Prompt For An Input
   ldr r0, =prompt1
   bl  printf
#Scanf
   ldr r0, =input1
   ldr r1, =name
   bl scanf
# Printing The Message
   ldr r0, =format1
   ldr r1, =name
   bl  printf
# Return to the OS
   ldr lr, [sp, #0]
   add sp, sp, #4
   mov pc, lr
.data
   prompt1: .asciz "Enter your name: "
   input1:  .asciz "%s"
   format1: .asciz "Hello %s, how are you today? \n"
   name:    .space 40
```

IOExample_1.2 explanation

- **The only code in this program not previously explained is at the bottom of this slide.**
- **The scanf function uses:**
  - r0 as the address of a string containing formatting information on what data to read (%s for string).
  - r1 as the address of memory into which to store the input.

```
#Scanf
   ldr r0, =input1
   ldr r1, =name
   bl scanf
   …
input1:  .asciz "%s"
```

IOExample_3.s – scanf function using data memory

```
#IOExample_3.s – scanf example
   .text
   .global main
main:
# Save return to os on stack
   sub sp, sp, #4
   str lr, [sp, #0]
# Prompt For An Input
   ldr r0, =prompt1
   bl  printf
#Scanf
   ldr r0, =input1
   ldr r1, =num1
   bl scanf
# Printing The Message
   ldr r0, =format1
   ldr r1, =num1
   ldr r1, [r1, #0]
   bl  printf
# Return to the OS
   ldr lr, [sp, #0]
   add sp, sp, #4
   mov pc, lr
.data
   num1: .word 0
   input1: .asciz "%d"
   prompt1: .asciz "Enter A Number\n"
   format1: .asciz "Your Number Is %d \n"
```

IOExample_1.3 explanation

This example only differs from the previous example in that: :

- r0 as the address of a string containing formatting information on what data to read, %d for a decimal value 1 byte big.
- Note that the memory for num1 is statically allocated, e.g. it is permanent and is never released. The next slide addresses this issue using a slight-of-hand trick on the stack.
- The stack trip is possible because the size of the input data is known as 4 bytes.
- It is a common trick so it is shown here, but will not be explained until later in the class. Do not be concerned if you cannot understand it yet.

IOExample_4.s – scanf function using stack memory

```
#IOExample_4.s - scanf example
   .text
   .global main
main:
# Save return to os on stack
   sub sp, sp, #4
   str lr, [sp, #0]
# Prompt For An Input
   ldr r0, =prompt1
   bl  printf
#Scanf
   ldr r0, =input1
   sub sp, sp, #4
   mov r1, sp
   bl  scanf
   ldr r2, [sp, #0]
   add sp, sp, #4
# Printing The Message
   ldr r0, =format1
   mov r1, r2
   bl  printf

# Return to the OS
   ldr lr, [sp, #0]
   add sp, sp, #4
   mov pc, lr

.data

num1: .word 0
format1: .asciz "Your Number Is %d \n"
prompt1: .asciz "Enter A Number\n"
input1: .asciz "%d"
```