CprE 288 – Introduction to Embedded Systems

Instructors:
Dr. Phillip Jones

Overview

- Announcements
- Interrupts
  - Textbook reading:
    - Section 2.4
    - Chapter 5 (pages 261 – 306)

Announcement

- HW 5: Not graded
  - **CPRE 288: Datasheet Trainer**
  - **HW 5 with solutions posted this evening**
- Lab 5: UART
- Exam 1: In class October 5 (Thursday, 10/5).
- Formal in class activity next Tuesday (9/26)
- Lab 6: ADC
  - Textbook: Chapter 7.5: pages 446-479 (~35 pages, but quite a bit of redundancy)

ISR (INTERRUPT SERVICE ROUTINES)

Interrupts and Interrupt Service Routines (ISRs)

- **Interrupt**: A mechanism that allows hardware to inform the CPU that an event has occurred. Example hardware events:
  - Button connected to an GPIO port pressed
  - Data arrives to the system. e.g. new byte on UART interface, new sample from analog to digital converter (ADC)
  - Timer expires or Timer becomes equal to a given value
  - CPU tries to execute an invalid assembly instruction

- **Interrupt Service Routine (ISR)**: code to be executed to deal with the hardware event that has occurred. Also referred to as a Interrupt handler.

General Interrupt to ISR flow

1. Hardware event causes an interrupt to occur
2. Interrupt notifies the CPU
3. CPU pauses the program
4. CPU disables interrupts and saves its "state"
   - The CPU state is the information the CPU needs to un-pause the program properly. e.g. location of the instruction when the paused occurred, current CPU register values
5. CPU re-enables interrupts and executes the proper ISR
6. Once the ISR completes, the CPU disables interrupts and restores its state to what it was before the interrupt occurred
7. CPU re-enables interrupts, and continues the program from where it paused.

Nested Vector Interrupt Controller (NVIC)

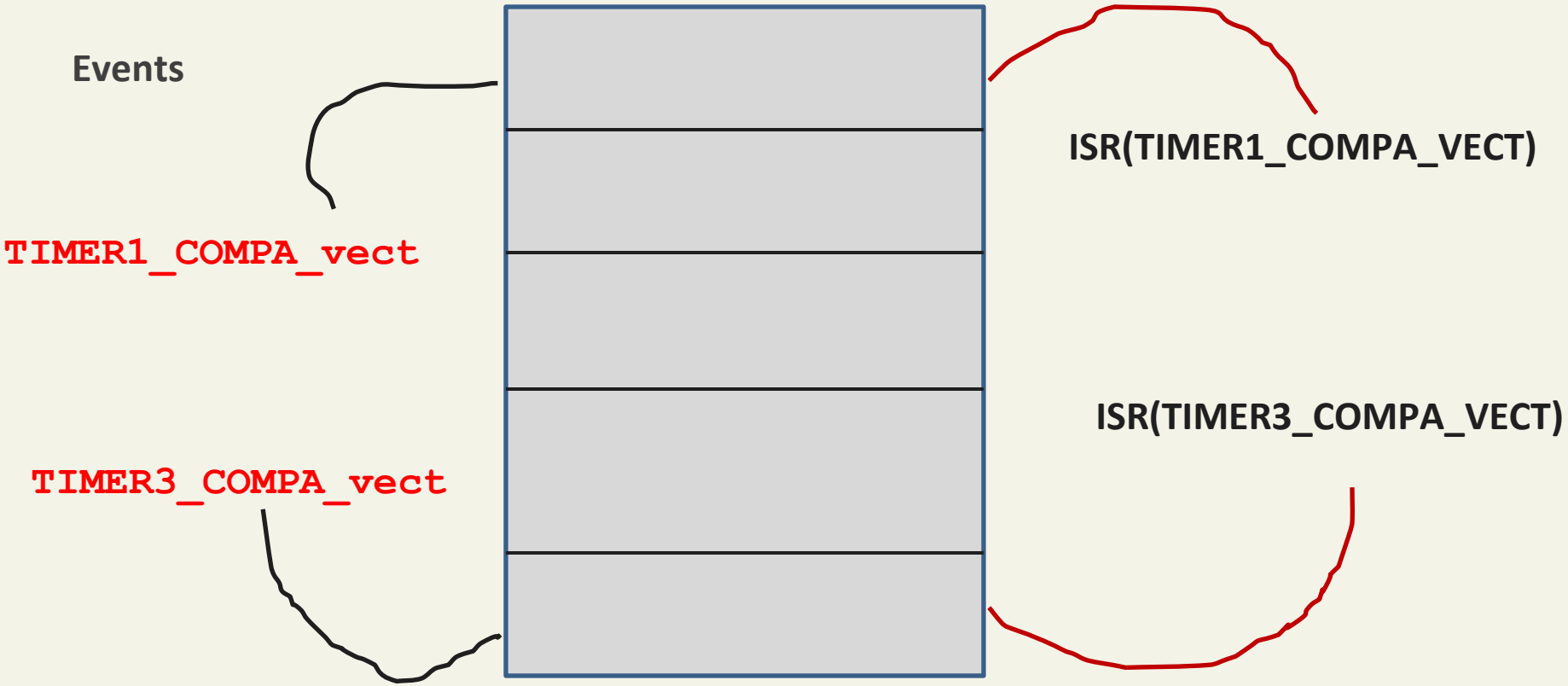
- **NVIC**: the name of the hardware on the CPRE 288 microcontroller chip that manages interrupts
  - Notifies CPU when an interrupt occurs
  - Programmer configures to enable/disable specific interrupts
  - Programmer configures to give interrupts priorities
  - Provides the CPU with information for accessing an *Interrupt Vector Table*, which stores the starting address (i.e. entry point) of each ISR.
- **Interrupt Vector Table**: Each row in this table (located in memory) contains the address of the starting instruction for each ISR. The CPU uses this information to start execution of the ISR that has been "triggered" by a corresponding Interrupt (i.e. hardware) event)

Interrupt Service Routine (ISR) Setup

1. **Enable the interrupt:** (every interrupt has an enable bit): Use the datasheet to find the register name and bit position you need to set.
    - Find the **interrupt number** on page 104 of the datasheet and set a 1 to its bit in the NVIC_ENn_R register. Where n is 0-4 and indicates a group of 32 interrupts (i.e. 0 -> 0-31, 1-> 32-63, etc). The bit you set is the (interrupt number - (32*n))th bit.
2. **Bind the handler** (i.e. indicate where to go when an interrupt event occurs)
    - Find the corresponding Interrupt **vector number** for your interrupt. (datasheet, page 104). Alternatively the timer interrupt vectors are defined as INT_TIMERxn where x is 0-5 and n is A or B.
    - Call IntRegister(interrupt vector number, handler name) to bind the interrupt(s) to your handler (i.e. ISR name). IntRegister can be found in interrupt.h
3. **Write the ISR** (Interrupt Service Routine)
    - The ISR is a function, or block of code, the CPU will call for you whenever the interrupt event occurs. You define/program what type of processing should occur for a given type of interrupt event.

Interrupt in Embedded Systems (Extra)

- A low-level simplified hardware figure to show how an event is mapped to a ISR vector: **Interrupt Vector Table**



ISR Example: Lab 4

```
int main()
{
	lcd_init();
	clock_timer_init();  // enable interrupt and create handlers
	while (1) {
	    // do nothing
	}
}
//interrupts are enabled by default, but you may want to
//call IntMasterEnable() in "driverlib/interrupt.h" in case
//any of the preceding code disabled interrupts.
```

ISR Example: Lab 4

```
/* Timer interrupt source 1: the function will be
   called every one second to update clock */
void TIMER4A_Handler()
{
   // YOUR CODE
}

/* Timer interrupt source 2: for checking push
 button five times per second*/
void TIMER4B_Handler()
{
   // YOUR CODE
}
```

Volatile Varaibles

Volatile variable: The memory content may change even if the running code doesn't change it.

```
volatile unsigned char pushbutton_reading;

  void TIMER4B_Handler()
{
  … // read PORT for push button
  pushbutton_reading = …;
}

main()
{
  while (!pushbutton_reading)
    {}
  … // other code
}
```

UART / ADC

OPERATOR PRECEDENCE

Operator Precedence Chart

Operator Type	Operator	Associativity
Primary Expression Operators	`() [] . -> expr++ expr--`	left-to-right
Unary Operators	`* & + - ! ~ ++expr --expr (typecast) sizeof`	right-to-left
Binary Operators	`* / %`	left-to-right
	`+ -`	
	`>> <<`	
	`< > <= >=`	
	`== !=`	
	`&`	
	`^`	
	`|`	
	`&&`	
	`||`	
Ternary Operator	`?:`	right-to-left
Assignment Operators	`= += -= *= /= %= >>= <<= &= ^= |=`	right-to-left
Comma	`,`	left-to-right

Exercise: Operation Precedence

a*b + c*d same as (a*b) + (c*d)

How about the following expression and condition?

x + y * z + k	x + (y * z) + k
*str++	*(str) str = str + 1;
if (a == 10 && b == 20)	if ((a == 10) && (b == 20))
if (a & 0x0F == b & 0x0F)	if (a & (0x0F == b) & 0x0F)
if ((a & 1) == 0)	

Are ()'s required?

x & (0x10 == 0x10) No

x & (!y) No

(x == 23) && (y < 12) No

```
// Increase each element by 1, exit if an element increases to 0
int my_array[50] = {1, 2, 3, 4, -1};
int *array = my_array;
do {
   (*array)++;
} while (*array++);
```

Yes

SCOPE

Variable scope

Global vs. Local

Global variable

- Declared outside of all functions
- May be initialized upon program startup
- Visible and usable everywhere from .c file

What happens when local/global have the same name?

- Local takes precedence

Summary

- Local – declared inside of a function, visible only to function
- Global – declared outside all functions, visible to all functions

Variable scope

What happens when you want a local variable to stick around but do not want to use a global variable?

Create a *static* variable

Syntax:

static Type Name;

Static variables are initialized once

- Think of static variables as a **"local" global**
- Sticks around (has persistence) but only the function can access it

Variable scope

C global variable (visible to all program files)

```
int global_var;
```

C file-wide static variables (visible only in this file)

```
static int static_var;
```

Local static variables

```
any_func()
{
  static int static_var;
  …
}
```

Variable scope

Example: How to define and use global variables

In header file myvar.h

```
extern int global_var;
```

In program file myvar.c

```
#include “myvar.h”
int global_var;
```

In program file usevar.c

```
#include “myvar.h”
… /* use myvar */
```

Visibility Scope Across Multiple Files

File1.c

```
// global variable
int count = 0;
```

This instance of “count” is visible in all files in the same project.

File2.c

```
extern int count;
int x = count;
```

This is how to use the global variable “count” declared in file1.c.

“extern” declaration is usually put in a header file.

Visibility Scope Across Multiple Files

File1.c

```
// global variable

int count = 0;
```

Another scenario: We want to use the same name “count” in multiple program files, each as a unique variable instance.

File2.c

```
// another global variable
// with the same name
int count = 100;
```

Bad use. The compiler/linker will report conflicting use of name “count”.

Some complier may tolerate it – still bad practice.

Visibility Scope Across Multiple Files

File1.c

```
// static global variable
static int count = 0;
```

Outside the functions, “static” means to limit the visibility of “count” to this program file only.

“static” is a also a storage class modifier (see later).

File2.c

```
// count for file2.c
static int count = 100;
```

“file2.c” gets its own “count”. There is no conflict.

Each instance of “count” is visible in its own file, not visible in any other file.

Variable scope

Visibility scope: Where a variable is visible

```
int m=0;
int n;

int any_func()
{
    int m;
    m = n = 5;
}
main()
{
  printf("%d", m);
}
```

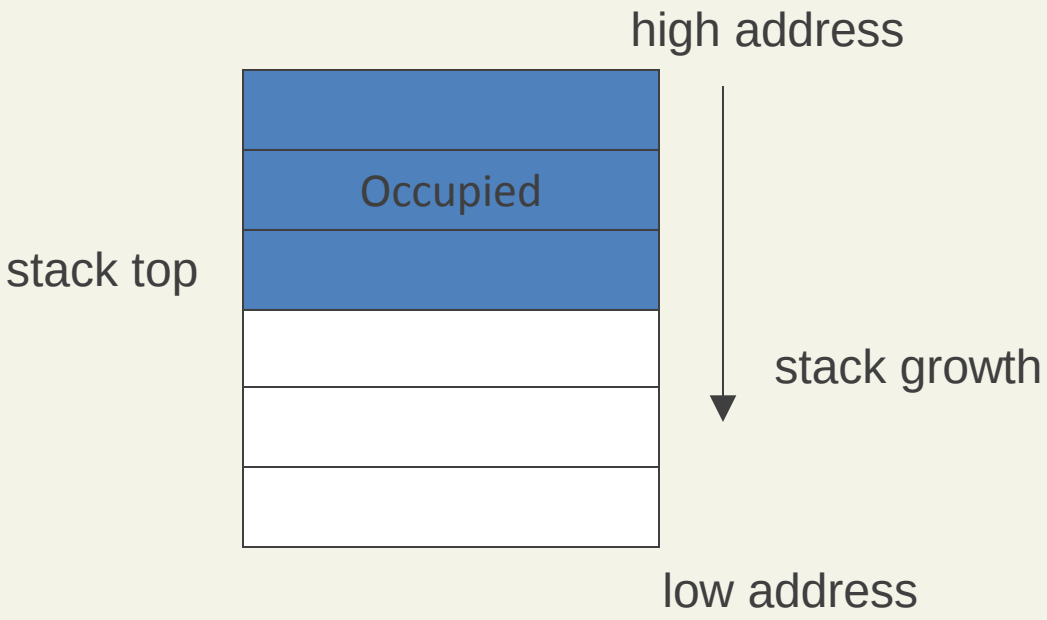
MEMORY LAYOUT

Understanding Data

- Stack
  - Stores data related to function variables, function calls, parameters, return variables, etc.
  - Data on the stack can go "out of scope", and is then automatically deallocated
  - Starts at the top of the program's data memory space, and addresses move down as more variables are allocated
- Heap
  - Stores dynamically allocated data
  - Dynamically allocated data usually calls the functions *alloc* or *malloc* (or uses *new* in C++) to allocate memory, and *free* to (or *delete* in C++) deallocate
  - There's no garbage collector!
  - Starts at bottom of program's data memory space, and addresses move up as more variables are allocated

Function and Stack

Conventional program stack grows downwards: New items are put at the top, and the top grows down



Function and Stack

Auto, local variables have their storage in stack

Why stack?

- The LIFO order matches perfectly with functions call/return order
  - LIFO: Last In, First Out
  - Function: Last called, first returned
- Efficient memory allocation and de-allocation
  - Allocation: Decrease SP (stack top)
  - De-allocation: Increase SP

Function and Stack

Function Frame: Local storage for a function

Example: 1. A is called; 2. A calls B; 3. B calls C; 4. C returns

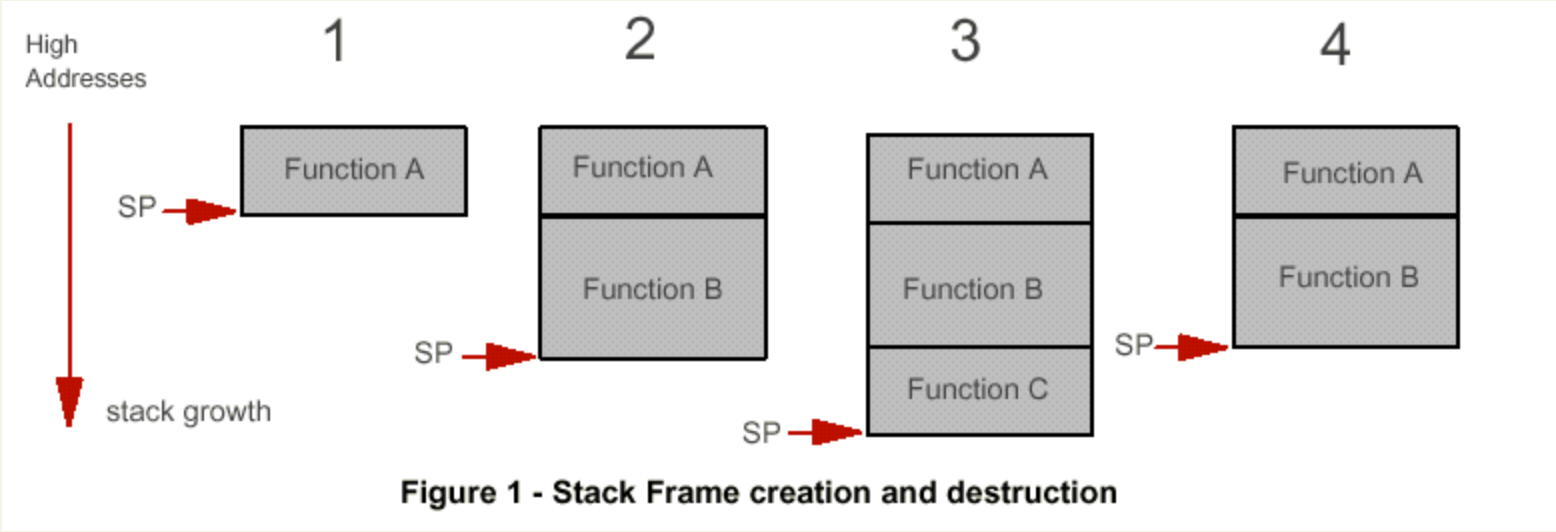


Figure 1 - Stack Frame creation and destruction

Function and Stack

What can put in a stack frame?

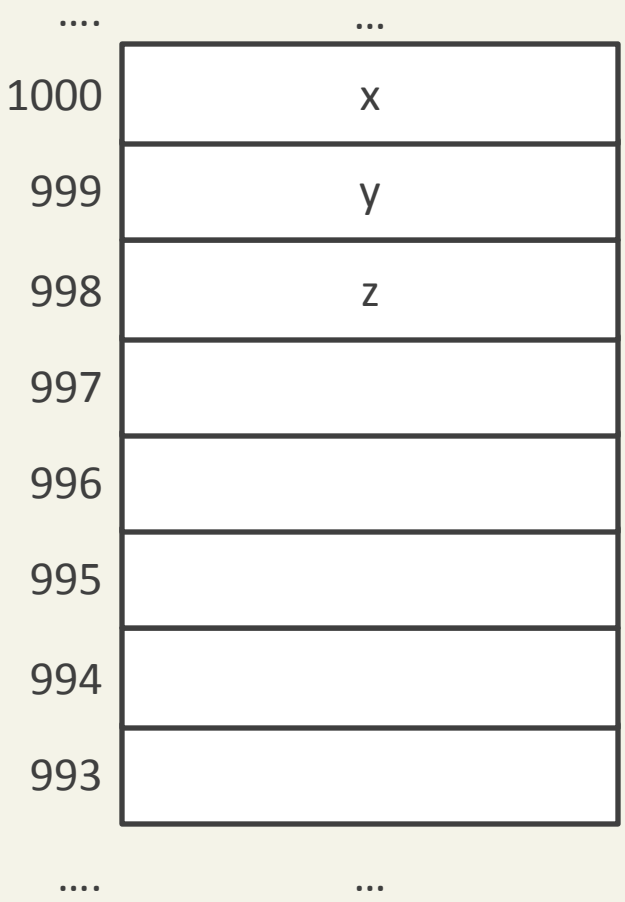
- Function return address
- Parameter values
- Return value
- Local variables
- Saved register values

Example: Stack

- The following example shows the execution of a simple program (left) and the memory map of the stack (right)

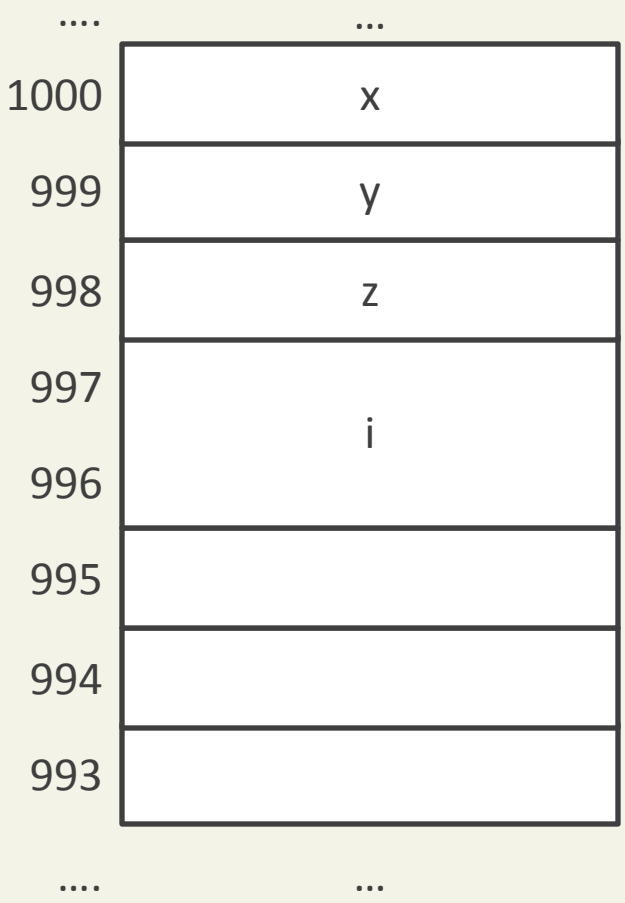
Example: Stack

```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



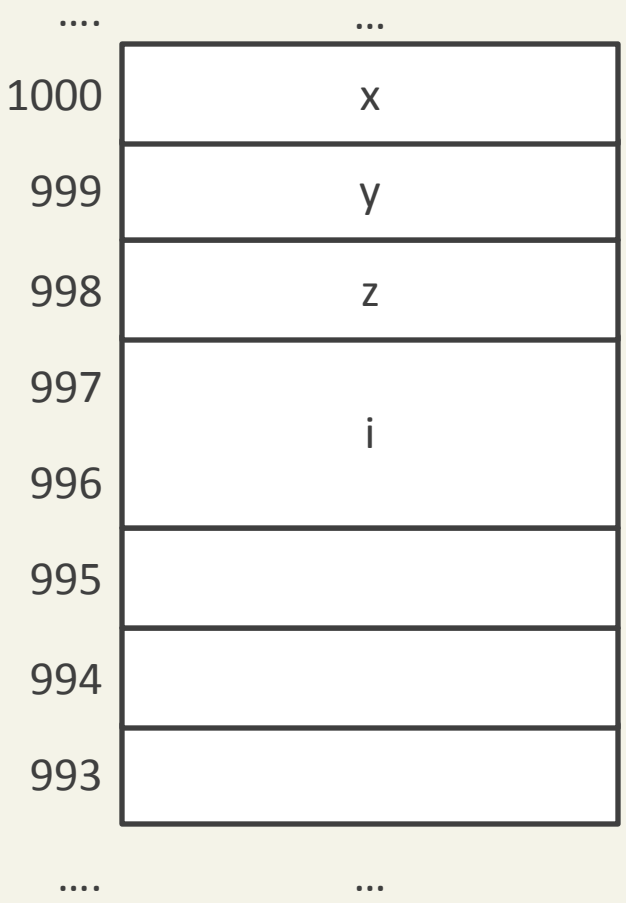
Example: Stack

```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



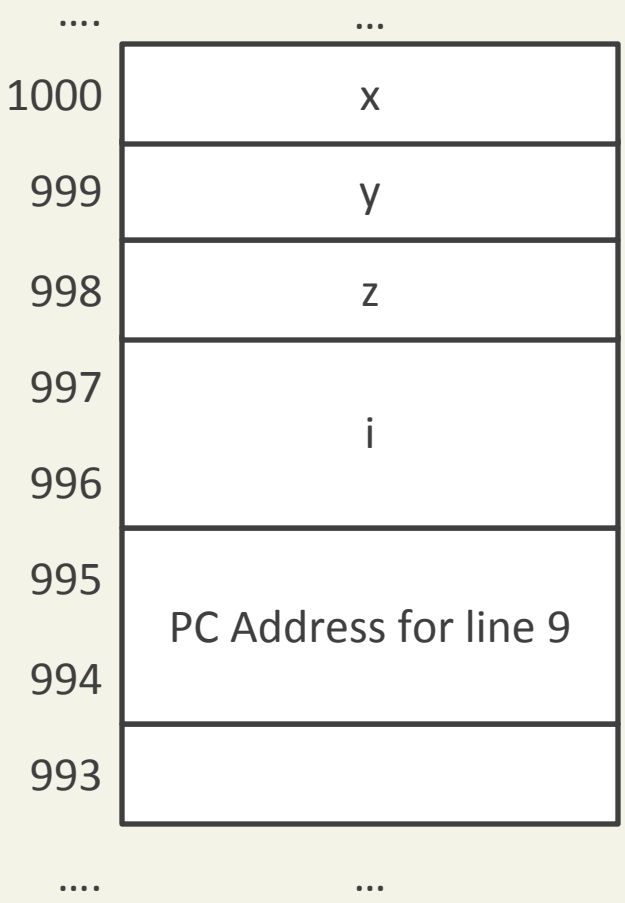
Example: Stack

```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



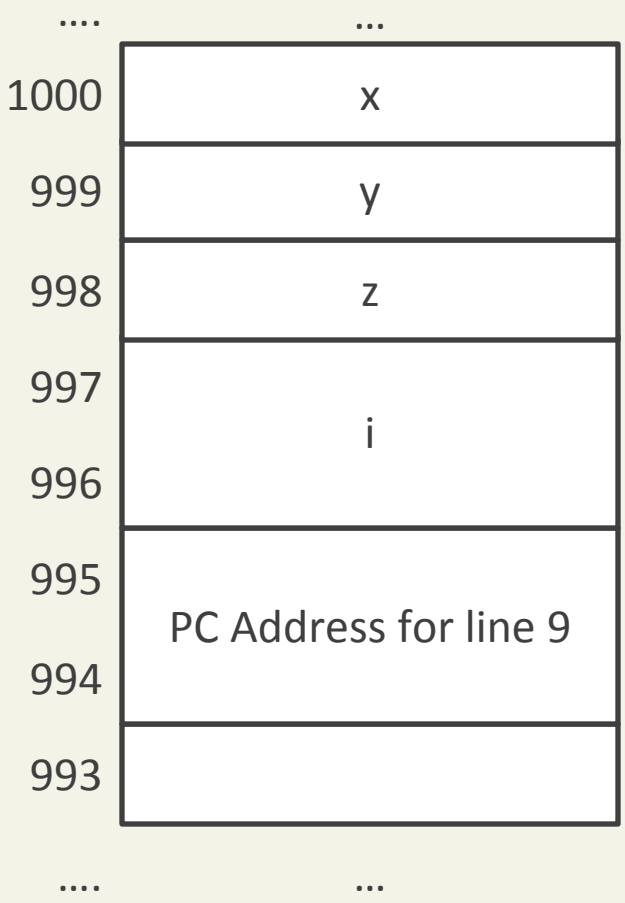
Example: Stack

```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



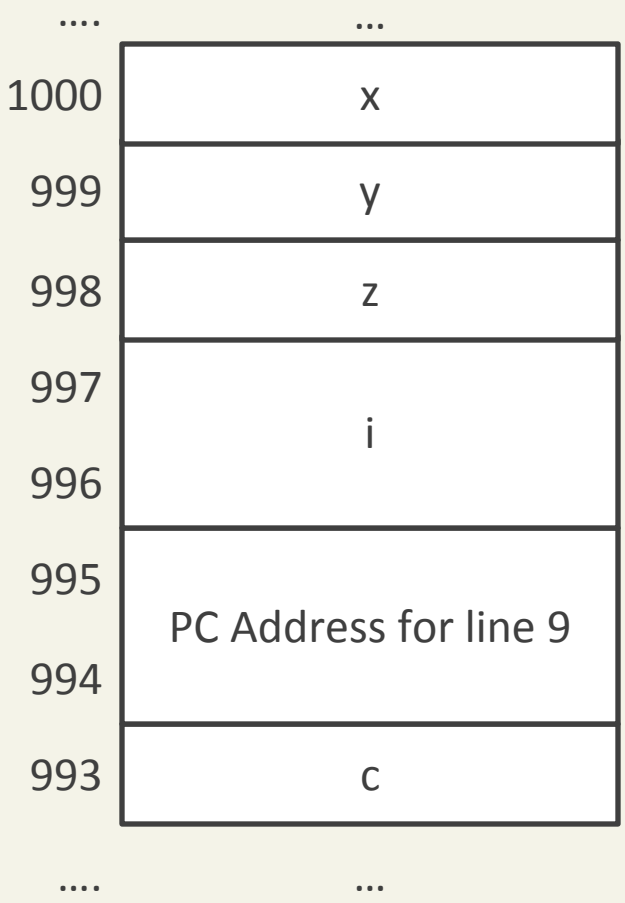
Example: Stack

```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



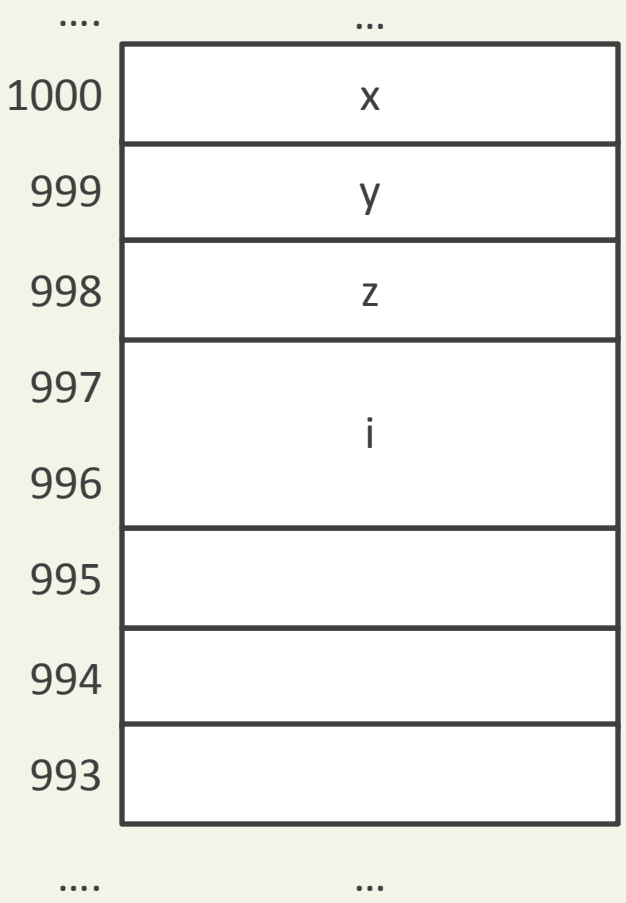
Example: Stack

```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```



Example: Stack

```
void doNothing() {
        char c;
}

int main() {
        char x, y, z;
        int i;
        for (i = 0; i < 10; i++) {
                doNothing();
        }
        return 0;
}
```

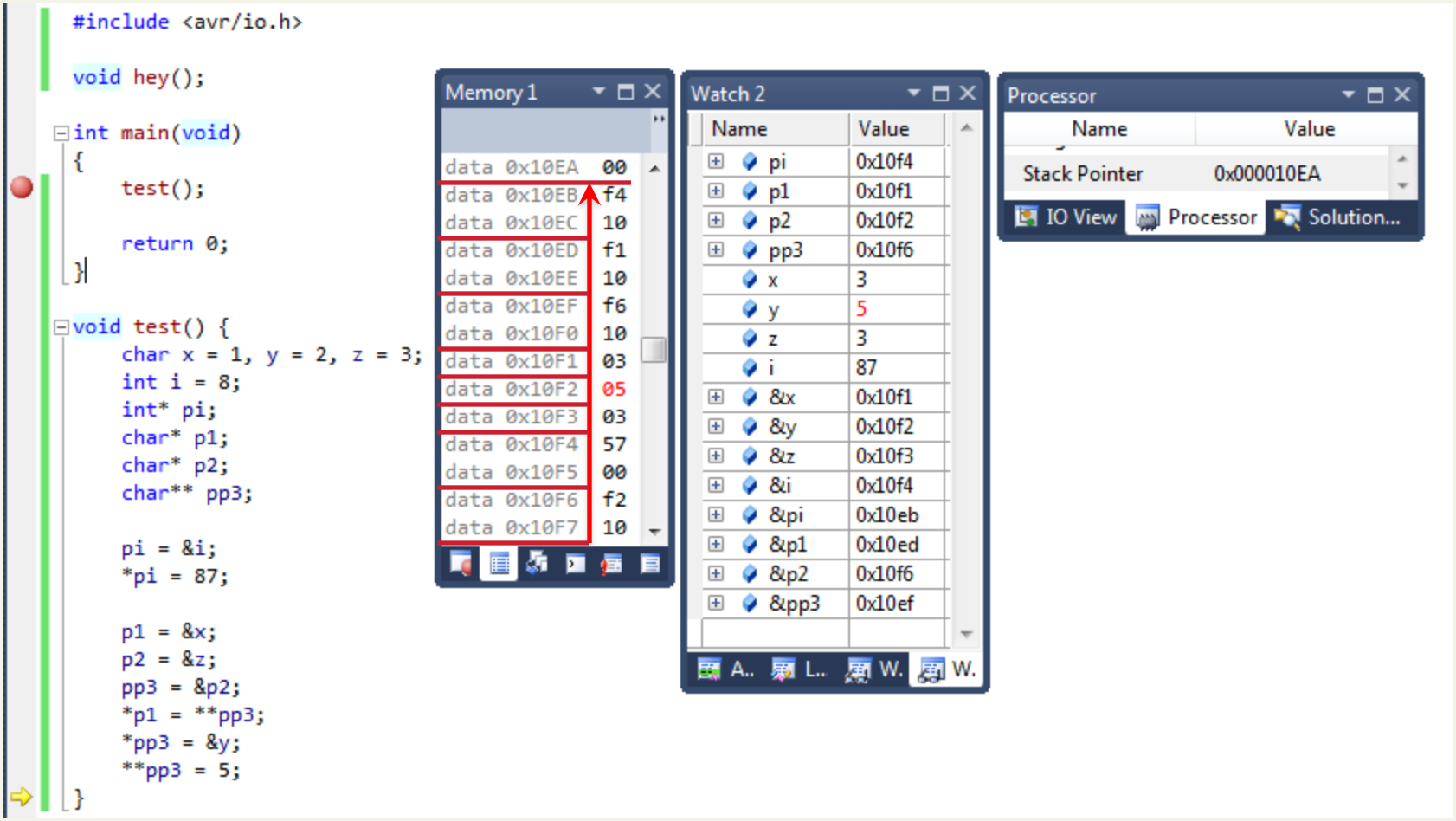


Stack Memory Layout: Example

```
char x = 1, y = 2, z = 3;
int i = 8;
int* pi;
char* p1;
char* p2;
char** pp3;

pi = &i;
*pi = 87;              // i = 87;

p1 = &x;
p2 = &z;
pp3 = &p2;
*p1 = **pp3;           // x = z;
*pp3 = &y;
**pp3 = 5;             // y = 5;
```

- Class work out on board. Final values for all memory locations.

Stack Memory Layout: Example



Note: Before calling test(), the stack pointer started at 0x10FB, added the program counter and the current stack pointer to the stack (at address 0x10F9 and 0x10FB)

Memory Address Space

It is the addressability of the memory

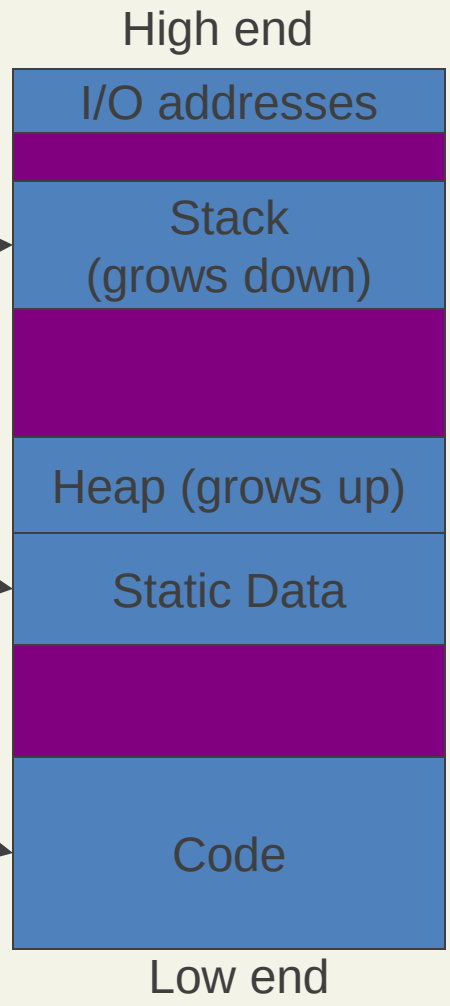
- Upper bound of memory that can be accessed by a program
- The larger the space, the more bits in memory addresses
- 32-bit address – accessibility to 4GB memory

What are

- Virtual memory address space
- Physical memory address space
- Physical memory size
- I/O addresses (ports)

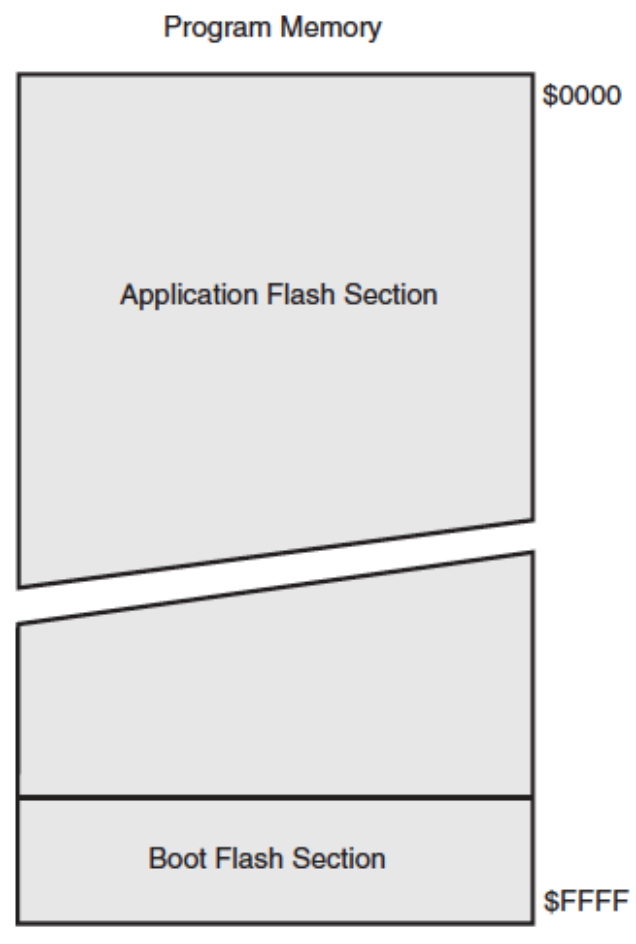
General Memory Layout

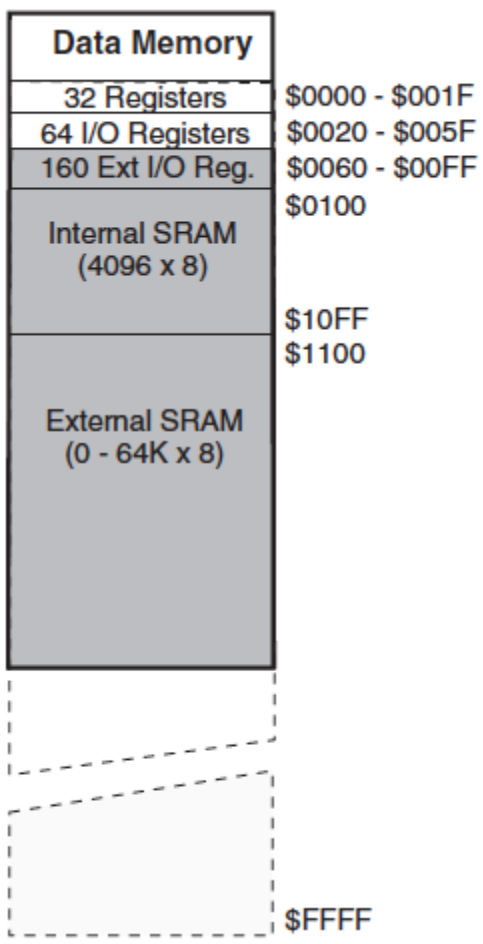
```
static char[] greeting
   =“Hello world!”;

main()
{
   int i;
   char bVal;

   LCD_init();
   LCD_PutString(greeting);
   …
}
```



ATmega128 Memory Layout

Harvard Architecture: Two separate memory address spaces for instruction and data





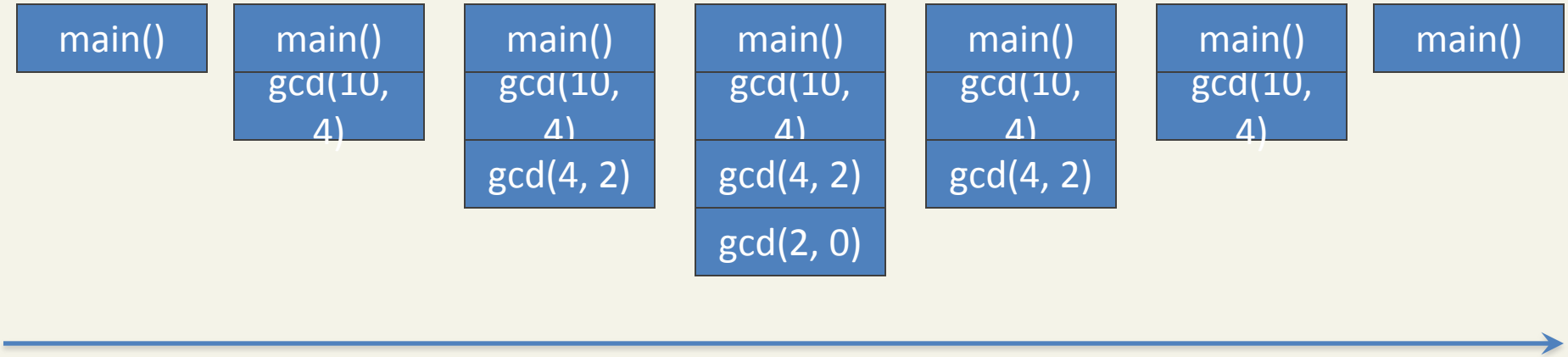
Recursive Function

A function that calls itself

```
/* calculate the greatest common
    divisor */
int gcd(int m, int n)
{
    if (n == 0)
        return m;
    else
        return gcd(n, m % n);
}
```

Function and Stack

The use of stack by a recursive function:



What happens if a function keeps calling itself and does not end the recursion?

TYPE CONVERSION (CASTING)

Type Conversion and Casting

Recall C has the following basic data types:

char, short, int, long, float, double

Assume:

```
char c;   short h;   int n;   long l;
float f; double d;
```

What's the meaning of

```
c = h;
n = h;
f = n;
(f > d)
```

Implicit Conversion

A longer integer value is cut short when assigned to a shorter integer variable or char variable

```
char c;
short h = 257;
long l;

c = h;          // The rightmost 8-bit of h is copied into c

n = l;          // The rightmost 16-bit of l is copied into n
```

Implicit Conversion

A shorter integer value is extended before being assigned to a longer integer variable

```
l = h;          // the 16-bit value of h is extended to 32-bit

h = c;          // the 8-bit value of c is extended to 16-bit
                // signed extension or not is dependent on
                //    the system
```

Implicit Conversion

A double type is converted to float type and vice versa using IEEE floating point standard

```
d = 10.0;       // 10.0 with double precision
f = d;          // 10.0 with single precision

f = 20.0;       // 20.0 with single precision
d = f;          // 20.0 with double presion
```

Implicit Conversion

A float/double is floored to the closest integer when assigned to an integer/char variable

```
f = 10.5;
n = f;          // n = 10

d = -20.5;
l = d;          // l = -20
```

Implicit Conversion

In an expression:

- A shorter value is converted to a longer value before the operation
- The expression has the type of the longer one

(c + h)	c is extended to 16-bit and then added with h
(n + l)	n is extended to 32-bit and then added with l
(f + d)	f is extended to double precision before being added with d

Explicit Conversion: From String to Others

```
#include <inttype.h>
#include <stdlib.h>

n = strtol(“10”);          // n = 10
f = strtof(“2.5”);         // f = 2.5 in single precision
d = strtod(“2.5”);         // d = 2.5 in double precision
```

strtol: string to long

strtof: string to float

strtod: string to double

Type Casting

Explicitly convert one data type to another data type

(type name) expression

```
int n1 = -1;
unsigned int n2 = 1;

if (n1 < (int) n2)                    // this is true

if ((unsigned int) n1 < n2)           // this is false
```

Explicit Casting

```
int i = 60;
float f = 2.5;

f = (float) (i + 3);
```

C LIBRAY FUNCTIONS

C Library Functions

In C many things are carried out by library functions

- Simple language, rich libraries

Commonly used libraries

- File I/O (include user input/output)
- String manipulations
- Mathematical functions
- Process management
- Networking

C Library Functions

Use standard file I/O

```
/* include the header file for I/O lib */
#include <stdio.h>

main()
{
  /* use the fprintf function */
  fprintf(stdout, “%s\n”, “Hello World\n”);
}
```

C Library Functions

Formatted output: printf, fprintf, sprintf and more; use conversion specifiers as follows

%s	string
%d	signed decimal
%u	unsigned decimal
%x	hex
%f	floating point (float or double)

How to output the following variables in format "a = …, b =…, c = …, str = …" in a single line?

```
int a;
float b;
int *c;
char str[10];
```

C Library Functions

String operations: copy, compare, parse strings and more

```
#include <string.h>
```

- strcpy: copy one string to another
- strcmp: compare two strings
- strlen: calculate the length of a string
- strstr: search a string for the occurrence of another string

C Library Functions

Error processing and reporting: use exit function

```
#include <stdio.h>
#include <stdlib.h>
...
void myfunc(int x)
{
  if (x < 0) {
    fprintf(stderr, “%s\n”,
          “x is out of range”);
    exit(-1);
  }
}
```

C Library Functions

Math library functions

```
#include <math.h>

…
  n = round (x); /* FP round function */
…
```

To build:

gcc –Wall –o myprogram **–lm** myprogram.c

C Library Functions

How to find more?

On Linux machines: Use man

```
man printf
man string
man string.h
man math.h
```

Most functions are available on Atmel platform

C Library Functions

More information on C Library functions:http://www.acm.uiuc.edu/webmonkeys/book/c_guide/

Other commonly used:

- stdlib.h: Some general functions and macros
- assert.h: Run-time self checking
- ctype.h: Testing and converting char values

AVR C Library Functions

TI Driver Library User's Guide:
http://www.ti.com/lit/ug/spmu298d/spmu298d.pdf
AVR Libc Home Page: http://www.nongnu.org/avr-libc/

Non AVR-specific:

- alloca.h: Allocate space in the stack
- assert.h: Diagnostics
- ctype.h: Character Operations
- errno.h: System Errors
- inttypes.h: Integer Type conversions
- math.h: Mathematics
- setjmp.h: Non-local goto
- stdint.h: Standard Integer Types
- stdio.h: Standard IO facilities
- stdlib.h: General utilities
- string.h: Strings

AVR C Library Functions

AVR Libc Home Page: http://www.nongnu.org/avr-libc/

AVR-specific

- avr/interrupt.h: Interrupts
- avr/io.h: AVR device-specific IO definitions
- avr/power.h: Power Reduction Management
- avr/sleep.h: Power Management and Sleep Modes
- util/setbaud.h: Helper macros for baud rate calculations
- Many others