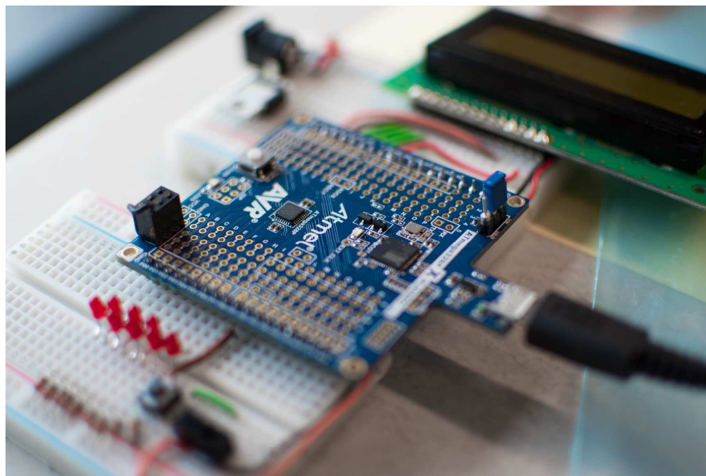
ECE3411 – Fall 2017
Lecture 0a.

Course Outline

Marten van Dijk
Department of Electrical & Computer Engineering
University of Connecticut
Email: vandijk@engr.uconn.edu

UCONN

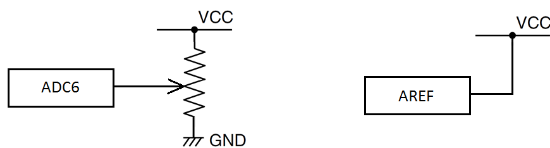
ATmega328P Development Board

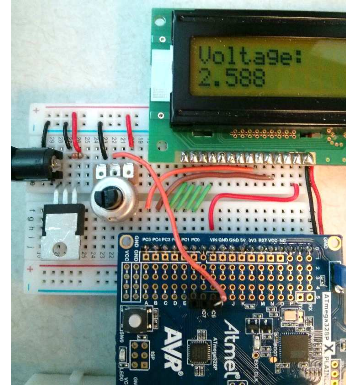


2

Interesting bits (1): Interfacing Analog Sensors

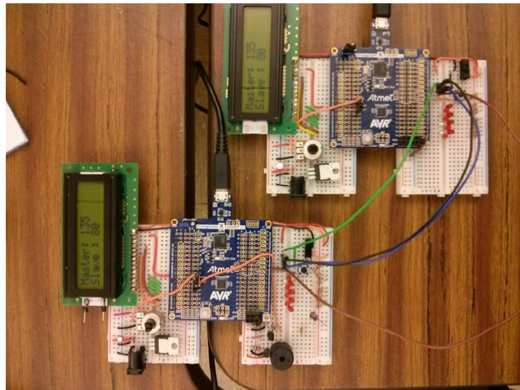
- Real world is Analog, whereas our computing systems are Digital
- Interfacing of Analog Sensors with the MCU is crucial component of Embedded Systems design
- In this course, you'll interface Temperature and Ambient Light sensors with the MCU to perform various control tasks.





Interesting bits (2): Communication Across Devices

- Communication across devices is a vital part of Embedded Systems
- You will explore two important communication protocols namely
  - SPI
  - I2C



Interesting bits (3): Playing with Timers & Interrupts

- A lot of Embedded Systems handle time-triggered and time-critical tasks!
- Timers of Microcontrollers serve several useful purposes related to embedded system tasks.
- We will be designing:
  - Timer based applications such as Stopwatch
  - Multi-tasking applications with time-triggered tasks
  - Pulse Width Modulation applications

Learning Objectives

[**O1**] Emulation of read examples (how to apply C primitives and how to layout your C code and well-comment your C-code such that it can be interpreted and understood by colleagues)

[**O2**] To be able to set up an Interrupt Service Routine (ISR) and how to communicate over microcontroller pins

[**O3**] To be able to write Finite State Machines (FSMs) that specify state transitions based on interrupt events and explain how program variables change

[**O4**] To be able to adopt a task based programming approach (without blocking delay functionality)

[**O5**] To be able to write non-blocking procedures (to communicate with e.g. UART and LCD).

[**O6**] To be able to debug programming errors and use the debugging tool to observe how the microcontroller steps through assembly instructions that represent procedures and ISRs.

[**O7**] To understand the importance of a Real Time Operating System (RTOS) and to be able to implement basic schedulers.

[**O8**] To be able to understand how the pins of the microcontroller are connected to peripheral devices.

[**O9**] To be able to read and understand the corresponding Atmega datasheet, in particular, how to enable interrupts and program their properties.

Material

- Optional reading: Elliot Williams, *Make: AVR Programming*, 2014
- Great website: Bruce Land's course at Cornell: http://people.ece.cornell.edu/land/courses/ece4760/
- Sign up for Piazza: piazza.com/uconn/fall2017/ece3411 (see email invitation)
- Piazza used for distribute general announcements, solutions to lab problems and problem sets, as well as additional reading material when needed
- We will also populate http://scl.uconn.edu/courses/ece3411/index.php with slide decks etc. but without solutions (only educators can request access to solution files)
- **Check Order Kit**
- **Make sure to have set up Eclipse/GCC & Atmel Studio; see document Tools_Setup.pdf which explains how to do this for your laptop**

Organization Lectures/Labs

- Split up in 7 blocks of two weeks each: B0 (assignments due before 1st drop date), B1, B2, B3 (assignments due before 2nd drop date), B4, B5, and B6
- BO: Revisits and refreshes prerequisites: Basic programming skills "if statements", "while loops", "procedures", "arrays", etc. (an understanding of 'complex' pointer based data structures is not needed for a successful completion of this course)
- B1, B2, B3: Guide students step-by-step towards solutions – You need to take initiative and ask questions
- B4, B5: We will let you work independently – You should still ask us questions when stuck!
- B6: Completely independent RedBot project – No help from instructors

Blocks

		O1	O2	O3	O4	O5	O6	O7	O8	O9
B0	Testing prerequisite coding skills before the first drop date.	Y								
B1	GPDO, GPDI, LEDs, UART, LCD	Y							Y	Y
B2	ISRs, Timers, non-blocking UART and LCD	Y	Y	Y	Y	Y				Y
B3	Debugging, External Interrupt, Timers	Y	Y		Y	Y	Y			Y
B4	PWM, ADC, Eeprom, Watchdog, Assembly		Y		Y	Y				Y
B5	RTOS, DAC, SPI, I2C, Servo Control		Y		Y	Y		Y		Y
B6	Overview advanced topics, RedBot project		Y		Y	Y	Y		Y	Y

Adult-Based Learning Theory

- Adults learn best when they have a flexible but challenging learning environment
- There will be a variety of assignments from which each student can choose how much to do in a positive/safe but challenging learning environment
- The open-ended assignments allow students to pursue tasks in a manner customized to individual needs and interests.
- No tests (quizzes, in class questions, lab tests)
- Grading is “safe” in that each (sub-)assignment (lab tasks or questions in problem sets) is graded **either pass or fail,** and the requirements for a pass are clearly specified
- Incomplete work justifies a “fail.”
- We reinforce the safety of the learning environment by giving a number of tokens representing opportunities to revise work and allowing 24-hour extensions.

Pass/Fail

- Dependent lab problems and independent lab questions are each pass/fail
- You need to satisfy the following specifications:
  - Working code:
    - Your code should compile.
    - Your code should solve the lab problem statement – i.e., the code should meet the lab problem's specification.
    - You need to demo your code on your own MCU during lab.
  - Coding style:
    - Your code should have at least one meaningful comment before each variable declaration, procedure or function definition, if-then-else statement, and loop/while/for statement. This will help you to make your code readable (to yourself as well as colleagues).
    - Unless otherwise stated, you are not allowed to use delay_ms and delay_um functionality.
    - You need to follow the coding template as taught in class (in later blocks you will do task based programming).
    - Since the first parts with hints in a lab problem lead to the final specification, you will only submit the code corresponding to this final specification in pdf form to the TA, who will verify the coding style and add questions, see below.
  - Understanding:
    - See syllabus for formulation: You need to be able to understand and express in own words the thought process that led you to your design choices – This will be tested in personalized (closed-book and/or oral) tests
    - If you cannot explain your thought process at a spot in your code, **your letter grade will be reduced by 1/3 letter grade.** So, if you cannot properly explain your thought process to say 3 *different* spots in your code, then your letter grade drops by a full letter!

11

Pass/Fail

- Problem set questions are each Pass/Fail
  - Each (sub-)question in the problem set must be answered using **at least 3 sentences. If you do not satisfy this requirement, then the points collected by your good answers will be halved.** So, we are checking whether you put in effort.
  - Each question part is graded pass/fail. If a question part is correct/passed, then it receives the full number of points allocated to that part. If failed, zero points are given.
  - See syllabus for formulation: You need to be able to understand and express in own words the thought process that led you to your answers – This will be tested in personalized (closed-book and/or oral) tests
    - If you cannot explain your thought process that led you to an answer for a question part, **your letter grade will be reduced by 1/3 letter grade.** So, if you cannot properly explain your thought process to say 3 *different* question parts in your problem set, then your letter grade drops by a full letter!
  - See syllabus for formulation: You need to be able to understand and express in own words how the posted solution for question parts are different from yours and why – This will be tested in personalized (closed-book and/or oral) tests
    - If you cannot explain how a posted solution for a question part is different from yours and why, **we reduce your problem set score by 20% of the points allocated to the question part referred at.** So, if you cannot properly explain the posted solution to say 3 *different* question parts in your problem set which were allocated 4, 8, and 8 points out of a 100, then 4 = 20%*(4+8+8) points will be subtracted from the points accumulated by your correctly answered problem parts.

12

Block Structure

Mo	We	Mo	We	Mo	We
Leca	Lecb	Lecc	Review	(Next Block starts)	
Laba out		Demo Laba “E” Test Laba Sol Laba out			
	Labb out		Demo Labb “E” Test Labb Sol Labb out		
		Labc out		Demo Labc “E” Test Labc Sol Labc out	
			LAB out	Demo LAB “E” Test LAB Sol LAB out	
Problem sets P and A out				“E” Test P and A If no test is postponed, then Sol P and A out	“S” Test P and A

!! Exact due dates/times of lab assignments and problem sets are in the syllabus !!

13

Grading

Letter grade or Pass/Fail requirements	Dependent Labs (B0 has 2; B1-5 each 3)	Independent Lab Questions (B0-5 each 2)	PSets	Adv. PSets	24hr Ext. Tok.	RedBot Project (No tokens)
	Help from instructors Collaboration students		Help from instructors NO collaboration students			
A/Pass	17 (B0-B5) 3 Rev. Tokens	12 (B0-B5) 1 Rev. Token	P1-P5 85%	A1-A6 80%	6 plus 1 for essay	Pass
B/Pass	17 (B0-B5) 4 Rev. Tokens	12 (B0-B5) 2 Rev. Token	P1-P5 80%	A1-A5 70%	6 plus 1 for essay	Not Req.
C/Pass	17 (B0-B5) 5 Rev. Tokens	2 (B0)	P1-P5 70%	Not Req.	4 plus 1 for essay	Not Req.
D/Fail	11 (B0-B3) 5 Rev. Tokens	2 (B0)	P0-P3 60%	Not Req.	4 plus 1 for essay	Not Req.

Revision token: See syllabus for formulation, you will see our solutions and you will make your code work accordingly and explain in at least 5 sentences in own words how your previous code needed to be improved

14

Collaboration

- **You can ask for help during labs and office hours from the instructors for the red colored columns in the table, i.e., dependent lab assignments and problem sets.**
- **You are allowed to collaborate with colleague students only on dependent labs – but not any of the independent labs and problem sets.**
- **You may not discuss independent labs and problem sets in any way, shape, or form with anyone other than your instructor. You are not allowed to collaborate or receive help from instructors for the independent labs and RedBot project.**
- **DURING THIS COURSE YOU ARE NOT ALLOWED TO RETRIEVE OR LOOKUP SOLUTIONS FROM OTHER SOURCES**

15

How to be successful in class?

- **I expect you to be actively engaged in your learning.** Ask questions during labs, office hours, review sessions. In order to learn how to program an MCU with peripherals you will need to practice.
- **Prepare diligently outside of class and come to class ready to work.** Don't procrastinate and ask for help when stuck. Be engaged and active in your learning – make effective use of lab time! Be engaged and active in your learning after class
- **Adopt a "growth mindset" for your intellectual development**
- **Practice self-regulated learning**
- **You should make sure to anticipate unexpected distractions and finish your work early.** See syllabus for formulation: You can use a *maximum of one* 24-hour extension token per problem set, dependent lab, or independent lab.
  - A revision token is only possible for labs and this will buy you extra days with access to our solutions
  - Problem sets must be completed over the course of about 2 weeks – plenty of time
- **All work (questions in problem sets, lab problems, project) is graded on a pass/fail basis, so careful attention to the specifications for acceptable work is a must**

16

Calendar: B0

1	Mo 28-Aug	**Lec0a:** Course Outline + Grading policy	**Lab0a:** Examples basic C-Programming	Read syllabus **Lab0a due Su 3-Sept** *P0 and Essay out*
	We 30-Aug	**Lec0b:** Introduction to C-Programming	**Lab0b:** Examples basic C-Programming Continued	**Lab0b due Tu 5-Sept**
2	4-Sept	Labor Day – no classes	Labor Day – no classes	
	We 6-Sept	**REVIEW, Q&A** C-Programming, Grading policy	**Independent LAB0:** Basic C-Programming using the Eclipse compiler	**LAB0, P0, and Essay due Su 10-Sept** *(No Office Hours)*

17

Calendar: B1

3	11-Sept	**Lec1a:** Microcontroller introduction + General Purpose Digital Output (GPDO)	**Lab1a:** AVR Board Setup (soldering) + LEDs (GPDO)	Ch. 1 Ch. 2 till page 21 Ch. 3 Ch. 4 **Lab1a due Su 17-Sept** *P1 and A1 out* **DROP DATE**
	13-Sept	**Lec1b:** Universal Asynchronous Receiver & Transmitter (UART)	**Lab1b:** UART (recognizing strings)	Ch. 5 till page 97 **Lab1b due Tu 19-Sept**
4	18-Sept	**Lec1c:** General Purpose Digital Input (GPDI) + LCD Display	**Lab1c:** LCD (GPDI)	Ch. 6 **Lab1c due Su 24-Sept**
	20-Sept	**REVIEW, Q&A** GPDO, GPDI, LEDs, UART, LCD	**Independent LAB1** GPDO, GPDI, LEDs, UART, LCD	**LAB1, P1, A1 due Su 24 Sept** *Office Hours*

18

Calendar: B2

5	25-Sept	**Lec2a:** Interrupt Service Routine (ISR)	**Lab2a:** Non-blocking UART + debounced switches (uses ISR)	**Lab2a due Su 1-Oct** *P2 and A2 out*
	27-Sept	**Lec2b:** ISRs + Timer 0 + Task based programming	**Lab2b:** Non-blocking LCD	Ch. 9 **Lab2b due Tu 3-Oct**
6	2-Oct	**Lec2c:** Timers 0, 1, and 2	**Lab2c:** Non-blocking LCD continued	**Lab2b due Su 8-Oct**
	4-Oct	**REVIEW, Q&A** Main focus: ISRs, Timers, non-blocking UART and LCD	**Independent LAB2** Main focus: ISRs, Timers, non-blocking UART and LCD	**LAB2, P2, A2 due Su 8-Oct** *Office Hours*

19

Calendar: B3

7	9-Oct	**Lec3a:** Debugging	**Lab3a:** Debugging	**Lab3a due Su 15-Oct** *P3 and A3 out*
	11-Oct	**Lec3b:** External Interrupt + Pin Interrupt	**Lab3b:** Human reaction time + Capture Interrupt Timer 1	Ch. 8 **Lab3b due Tu 17-Oct**
8	16-Oct	**Lec3c:** External Interrupt + Task based programming	**Lab3c:** Stopwatch	**Lab3c due Su 22-Oct**
	18-Oct	**REVIEW, Q&A** Main focus: Debugging, External Interrupt, Timers	**Independent LAB3** Main focus: Debugging, External Interrupt, Timers	**LAB3, P3, A3 due Su 22-Oct** *Office Hours*

20

Calendar: B4

9	23-Oct	**Lec4a:** Pulse Width Modulation (PWM)	**Lab4a:** PWM	Ch. 10 **Lab4a due Su 29-Oct** *P4 and A4 out*
	25-Oct	**Lec4b:** Analog-to-Digital Conversion (ADC)	**Lab4b:** ADC	Ch. 7 Ch. 12 **Lab4b due Tu 31-Oct**
10	30-Oct	**Lec4c:** Eeprom + Watchdog	**Lab4c:** Eeprom + Watchdog + Assembly	Ch. 18: p. 387-396 Ch. 19 Ch. 20 **Lab4c due Su 5-Nov** **DROP DATE / CONVERSION TO Pass/Fail** **(D+, D, D-, F students will have completed the course)**
	1-Nov	**REVIEW, Q&A** Main focus: PWM, ADC, Eeprom, Watchdog, Assembly	**Independent LAB4** Main focus: PWM, ADC, Eeprom, Watchdog, Assembly	**LAB4, P4, A4 due Su 5-Nov** *Office Hours*

21

Calendar: B5

11	6-Nov	**Lec5a:** Task based programming revisited + Real Time Operating System (RTOS) global understanding	**Lab5a:** RTOS Scheduling	**Lab5a due Su 12-Nov** *P5 and A5 out*
	8-Nov	**Lec5b:** RTOS Cont'd + SPI	**Lab5b:** SPI + DAC	Ch. 16 **Lab5b due Tu 14-Nov**
12	13-Nov	**Lec5c:** I2C + RedBot (PID control) + Servo Control	**Lab5c:** I2C	Ch. 11 Ch. 15 Ch. 17 **Lab5c due Su 19-Nov**
	15-Nov	**REVIEW, Q&A** Main focus: RTOS, DAC, SPI, I2C, Servo Control	**Independent LAB5** Main focus: RTOS, DAC, SPI, I2C, Servo Control	**LAB5, P5, A5 due Su 19-Nov** *Office Hours*

22

Calendar: B6

	20-Nov	Thanksgiving Recess – no classes	Thanksgiving Recess – no classes	
	22-Nov	Thanksgiving Recess – no classes	Thanksgiving Recess – no classes	
13	27-Nov	**Lec6a:** Advanced Topics	**Independent LAB6/Project:** RedBot	**LAB6/RedBot:** **Code due Tu 5-Dec** **Demo We 6-Dec** *A6 out*
	29-Nov	**Lec6b:** Topics advanced MCU Applications Laboratory (Spring 2018)	**Independent LAB6/Project:** RedBot	
14	4-Dec	**REVIEW, Q&A** Anything	**Independent LAB6/Project:** RedBot	**A6 due 5-Dec**
	6-Dec	**TBD**	**RedBot Demo**	
15	11-15 Dec	**Finals Week – No final**	**Finals Week – No Final**	

23

Success !!

24

ECE3411 – Fall 2017
Lab 0a.

Introduction to C-Programming

Marten van Dijk, Hamza Omar
Department of Electrical & Computer Engineering
University of Connecticut
Email: {marten.van_dijk, hamza.omar}@uconn.edu

UCONN

Adopted from Lab1a slides "Introduction to C-Programming"
by Marten van Dijk and Syed Kamran Haider, Fall 2015.

Prerequisites

- Eclipse development environment (with C Development Tools) installed
- Basic understanding of C Programming

2

Task 1: Approximate the value of π

- The value of π can be calculated by the following series expansion

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}+\cdots=\frac{\pi}{4} \quad \Rightarrow \quad \sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1}=\frac{\pi}{4}$$

- Task 1(a): Write a C program that takes a positive (≥ 0) integer n as input and prints the value of π computed up to the nth term of the above series.
  - E.g. if $n = 3$ then the program computes $\pi = 4 \times \left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}\right)$
- Task 1(b): Modify the program from Task1(a) such that it terminates only when the absolute value of the nth term becomes less than 10^{-6}
  - Implement your own function to compute the absolute value of a double
- Task 1(c): Modify the program from Task1(b) such that it terminates only when the relative error in the π values from two consecutive iterations becomes less than 10^{-8}, i.e., when the absolute value of (pi-last_pi)/last_pi is less than 10^{-8}
  - The final output should be the π value from the most recent iteration, i.e. one with the higher value of n.

3

Task 2: Finding Prime Numbers

- A **prime number** (or a **prime**) is a natural number greater than 1 that has no positive divisors other than 1 and itself.
  - E.g. 2, 3, 5, 7, …
- Task 2: Write a C program which takes an integer as input from the user and prints all the prime numbers (separated by a comma) that are less than the entered number.
  - E.g. if the user inputs "10" then the program should print "2, 3, 5, 7".

4

ECE3411 – Fall 2017

Lecture 0b.

Introduction to C-Programming

Marten van Dijk

Department of Electrical & Computer Engineering

University of Connecticut

Email: vandijk@engr.uconn.edu

UCONN

Slides adopted from Marten van Dijk & Syed Kamran Haider ECE 3411 - Fall 2016

Programming Languages

Evolutionaire analyse van computertalen levert een stamboom die meer wegheeft van een visnet

De evolutie van **codetaal**

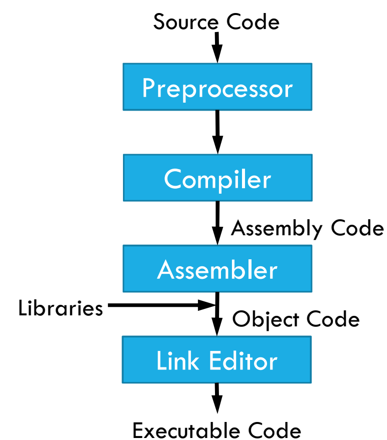
2

Introduction to C-Programming

- The C programming language was designed by Dennis Ritchie at Bell Laboratories in the early 1970s.
- C is mother language of all programming language used for systems programming.
- It is procedure-oriented and also a mid level programming language.

The C Compilation Model

- **The Preprocessor** accepts source code as input and is responsible for
  - Removing comments
  - Interpreting special preprocessor directives denoted by #.
  - Examples: **#include <stdio.h> , #define begin { , #define end }**
- **The C compiler** translates source to assembly code.
- **The assembler** creates object code.
- **The Link Editor** combines any library functions referenced in the source code with the **main()** function to create an executable file.



4

A simple C program : Printing ‘Hello World’

```
#include <stdio.h>
int main ()
{
    printf("Hello World");
    return 0;
}
```

stdio.h

```
#ifndef    _STDIO_H_
#define    _STDIO_H_
.....
#include <sys/cdefs.h>
#include <machine/ansi.h>
.....
int          printf(const char *, ...);
int          scanf(const char *, ...);
...
```

- **#include <stdio.h>**
  - Preprocessor directive which loads contents of a certain file
  - `<stdio.h>` allows standard input/output operations
- **int main ()**
  - **main** is the driver function of a c program where execution starts.
  - **int** means that **main** returns an integer value
- Bodies of all functions must be contained in curly braces
  - ‘ { ’ start of function
  - ‘ } ’ end of function
- **printf(“Hello World”);**
  - Prints the string of characters within quotes
  - Entire line is called a statement
  - All statements must end with a semicolon
- **return 0;**
  - A way to exit a function
  - Here it means that the program terminated normally

5

Another ‘Hello World’ Program

```
#include <stdio.h>
#define   begin     {
#define   end       }
int main ()
begin
    printf("Hello World");
    return 0;
end
```

- You can define your own macros
- **begin** represents the opening brace ‘{’
- **end** represents the closing brace ‘}’
- The body of **main ()** can be enclosed in **begin** and **end**
- However, the recommended way of enclosing the function body is to use the braces ‘{ }’
- You can define other macros as well, e.g.
  - #define MAX_ARRAY_SIZE 100

6

Tokens in C

- Keywords
  - These are reserved words of the C language.
  - For example int, float, if, else, for, while etc.
- Identifiers
  - An Identifier is a sequence of letters and digits, but must start with a letter.
  - Identifiers are used to name variables, functions etc.
  - Identifiers are case sensitive.
  - Valid: Root, _getchar, __sin, x1, x2, x3, x_1, If
  - Invalid: 324, short, price$, My Name
- Constants
  - 13, 'a', 1.3e-5 etc.
- String Literals
  - A sequence of characters enclosed in double quotes as "…".
  - For example "13" is a string literal and not number 13.
  - 'a' and "a" are different.
- Operators
  - Arithmetic operators: +, -, *, / ,%
  - Logical operators: ||, &&, !
- White Spaces
  - Spaces, new lines, tabs, comments (A sequence of characters enclosed in /* and */) etc.
  - These are used to separate the adjacent identifiers, keywords and constants.

Basic data types

char	Stored as 8 bits. Unsigned 0 to 255. Signed -128 to 127.
short int	Stored as 16 bits. Unsigned 0 to 65535. Signed -32768 to 32767.
int	Same as either short int or long int
long int	Stored as 32 bits. Unsigned 0 to 4294967295. Signed -2147483648 to 2147483647
float	Approximate precision of 6 decimal digits (single precision).
double	Approximate precision of 14 decimal digits (double precision).

Constants

- Numerical Constants
  - Constants like 12, 253 are stored as int type (No decimal point).
  - Numbers with a decimal point (21.53) are stored as float or double.
- **`Character and string constants`**
  - **`'c'`** , a single character in single quotes are stored as char.
  - Some special character are represented as two characters in single quotes.
    - **`'\n'`** = newline,
    - **`'\t'`**= tab,
    - **`'\\'`** = backlash,
    - **`'\"'`** = double quotes.
  - A sequence of characters enclosed in double quotes is called a string constant or string literal.
    - For example : **`"Hello"`**

Variables

- Variable names correspond to locations in the computer's memory
- Every variable has a name, a type, a size and a value
- Naming a Variable
  - Must be a valid identifier
  - Must not be a keyword
  - Names are case sensitive
- Declaring a Variable
  - Each variable used must be declared. Example : `data-type var1, var2,…;`
  - Declaration announces the data type of a variable and allocates appropriate memory location.
  - Initializing value to a variable in the declaration itself: `data-type var = expression;`
  - Examples: `int sum = 0; char newLine = '\n'; float epsilon = 1.0e-6;`

10

Global and Local variables

- Global Variables
  - These variables are declared outside all functions.
  - Life time of a global variable is the entire execution period of the program.
  - Can be accessed by any function defined below the variable's declaration, in a file.
- Local Variables
  - These variables are declared inside some functions.
  - Life time of a local variable is the entire execution period of the function in which it is defined.
  - Cannot be accessed by any other function.
  - In general variables declared inside a block are accessible only in that block.

Example of global and local variable

```
/* Compute Area of  a circle */
#include <stdio.h>
float pi = 3.14159;  /* Global variable */

int main() {
 float rad;   /* Local variable*/

 printf( "Enter the radius " );
 /* scanf obtains a value from user */
 /* Value is stored in rad */
 /* %f indicates that value should be float */
 scanf("%f" , &rad);

 if ( rad > 0.0 ) {
   float area = pi * rad * rad;
   printf( "Area = %f\n" , area );
 }
 else {
   printf( "Negative radius\n");
 }
  return 0;
}
```

12

Arithmetic Operators

- A = B → Assignment: A gets the value of B
- A + B → Add A and B together
- A – B → Subtract B from A
- A * B → A multiplied by B
- A / B → A divided by B
- A % B → Modulo: Integer remainder of A/B

Example:

```
int A = 11;
int B = 4;
int X = A / B;      // X gets the value 2. Since X is an integer, the fractional part is ignored.
int Y = A % B;      // Y gets the value 3 since A=BX+Y
```


Comparison Operators

- A == B → A is equal to B?
- A != B → A is NOT equal to B?
- A > B → A is greater than B?
- A < B → A is less than B?
- A >= B → A is greater than/equal to B?
- A =< B → A is less than/equal to B?

Logical Operators

Logical Operators map the inputs to either **TRUE** (Logical 1) or **FALSE** (logical 0)

These operators result in a single bit output

- !A → **NOT** A
- A && B → A **AND** B
- A || B → A **OR** B

Example:

```
if (A || (B && C) || !D)
{
  //do something;
}
```

if statement is only satisfied if
- A is logical high **OR,**
- B **AND** C are logical high **OR,**
- D is logical low.

15

Bitwise Operators

Bitwise operators map input bit vectors to the same sized output bit vector

- ~A → Bitwise complement of A
- A & B → Bitwise AND of A and B
- A | B → Bitwise OR of A and B
- A ^ B → Bitwise XOR of A and B
- A << B → Bitwise left shift A by B positions
- A >> B → Bitwise right shift of A by B positions

16

Bitwise Operators Examples

Let A = 0b11 and B = 0b01 then

- A represents the bit vector 11
- B represents the bit vector 01

~A	= 0b00		
A & B	= 0b11 & 0b01	= 0b01	
A | B	= 0b11 | 0b01	= 0b11	
A ^ B	= 0b11 ^ 0b01	= 0b10	
A << B	= 0b11 << 0b01	= 0b11 << 1	= 0b10
A >> B	= 0b11 >> 0b01	= 0b11 >> 1	= 0b01

We use bitwise operators frequently to manipulate the register values.

Prefix & Postfix Increment/Decrement

- ++A → The value of A is incremented before assigning it to variable A
- --A → The value of A is decremented before assigning it to variable A
- A++ → The value is incremented after assigning it to the variable A
- A-- → The value is decremented after assigning it to the variable A

Pre/Post Increment Examples

```
int x = 0;
while(++x < 5)
{
  printf("%d ", x);
}
```

- This prints 1, 2, 3, 4
- x is incremented BEFORE the comparison. Since 1 is less than 5, a '1' is printed. This is repeated until x = 4.
- Then the condition for the while loop fails, since x will be assigned a value of 5 before the values are compared.

```
int x = 0;
while(x++ < 5)
{
  printf("%d ", x);
}
```

- This prints 1, 2, 3, 4, 5.
- x is incremented AFTER the comparison, therefore, it meets the criteria of the while loop until x = 5.

Compound Assignments

- A += B → A = A + B
- A -= B → A = A – B
- A *= B → A = A * B
- A /= B → A = A/B
- A %= B → A = A%B
- A &= B → A = A&B
- A |= B → A= A|B
- A <<= B → A = A<<B
- A >>= B → A = A>>B

Control Structures: if/else statement

```
if(expression)
        <statement>
```

```
if(expression)
{
        /* Block of statements */
}
```

```
if(expression){
        /* Block of statements */
} else {
        /* other statements */
} else if (expression) {
        /* other statements */
} else if (..){
        /* ... */
}
```

- if statement can be used to execute some code if the condition in the expression is met.
- It can be used to execute a single code statement or a block of statements.
- if/else statement defines the alternate code to execute if the if-condition is not met.
- Note: if/else statements can be strung together with more if/else statements to add conditions to the 'else' parts.

Control Structures: switch statement

```
switch (<expression>)
{
   case <label1> :
      <statements 1>
       break;
   case <label2> :
      <statements 2>
      break;
   default :
      <statements 3>
}
```

- Used as a substitute for lengthy if statements that look for several conditions of some variable.

Control Structures: Loops

```
while ( <expression> )
{
    <statements>
}
```

```
for ( <expression1>; <expression2>; <expression3> )
{
    <statements>
}
```

```
do
{
    <statements>
}
while ( <expression> );
```

- while loop: While the condition in the expression statement is true, execute the statements in the loop.
- for loop: Similar to the while loop. expression1 initializes a variable, expression2 is a conditional expression, expression3 is a modifier, like an increment (x++).
- do-while loop is similar to while loop. It ensures that the block of statements is executed at least once.

23

for Loop Example

Temperature units conversion from Fahrenheit to Celsius:

```
#include <stdio.h>
int main() {
        int f;
        for (f=0; f <= 300; f += 20) {
                printf("%3d %6.1f \n", f, (5.0 / 9.0) * (f – 32.0));
        }
        return 0;
}
```

- %3d
  - % means "Print a variable here"
  - 3 means "Use at least 3 spaces to display, padding as needed"
  - d means "The variable will be an integer"
- %6.1f means "Print a float using 6 digits and round up to 1 decimal digit".

Interesting Fact:

- To approximate Celsius from Fahrenheit in your head:
  - Subtract 32 from F
  - Take half of the result and increase it by 10%

24

Conditional Expressions

- Conditional expressions

  `expr1? expr2 : expr3;`

- If `expr1` is true then execute `expr2` else execute `expr3`

Example:

```
for (int i=0; i<n; i++){
        printf("%d %c", a[i], (i%10==9 || i==(n-1))? '\n' : ' ');
}
```

Break and Continue statements

- **break** is used to terminate a loop immediately.
- **continue** is used to skip the subsequent statements inside the loop.

Examples:

```
while(test expression){
    <statements>
    if(test expression)
        break;
    <statements>
}
```

```
while(test expression){
    <statements>
    if(test expression)
        continue;
    <statements>
}
```

Type conversion

- The operands of a binary operator must have the same type and the result is also of the same type.
- Integer division: `c = (9 / 5)*(f - 32)`
- The operands of the division are both int and hence the result also would be int.
- For correct results, one may write `c = (9.0 / 5.0)*(f - 32)`
- In case the two operands of a binary operator are different, but compatible, then they are converted to the same type by the compiler. The mechanism (set of rules) is called Automatic Type Casting.

```
c = (9.0 / 5)*(f - 32)
```

- It is possible to force a conversion of an operand. This is called Explicit Type casting.

```
c = ((float) 9 / 5)*(f - 32)
```

Functions

- Functions are blocks of code that perform a number of pre-defined commands to accomplish something productive.
  - Library Functions
  - User Defined Functions
- Function prototypes are usually declared in the header files.
- General format for a function prototype

```
return-type function_name ( arg_type arg1, ..., arg_type argN );
```

- General format for a function body

```
return-type function_name ( arg_type arg1, ..., arg_type argN )
{
        /* Code for function body  */
}
```

Functions Example

```
#include <stdio.h>
int mult ( int x, int y );          // Function Prototype

int main()
{
   int x, y, z;
   printf( "Please input two numbers to be multiplied: " );
   scanf( "%d", &x );            // Call to a library function
   scanf( "%d", &y );            // Call to a library function
   z = mult( x, y );             // Call to a user-defined function
   printf( "The product of your two numbers is %d\n", z );
}

/* Function Body */
int mult (int x, int y)
{
   return x * y;
}
```

29

ECE3411 – Fall 2017

Lab 0b.

Introduction to C-Programming

Marten van Dijk, Hamza Omar
Department of Electrical & Computer Engineering
University of Connecticut
Email: {marten.van_dijk, Hamza.omar}@uconn.edu

UCONN

Adopted from Lab1b slides "Introduction to C-Programming" by Marten van Dijk and Syed Kamran Haider, Fall 2015.

Prerequisites

- Eclipse development environment (with C Development Tools) installed
- Basic understanding of C Programming

2

Task 1: Analyzing the Email Address

We want to determine if the email address entered by a user is of a valid format or not.

- Write a C program that takes an email address as input (character by character) and verifies that:
  1. It contains one and only one "at" sign '@'
  2. It contains at least one period '.' which succeeds the '@' sign
- Please read the input character by character, and implement a state machine to analyze the input.
- For example
  - alice@gmail.com is considered valid.
  - alice@mydomain.co.uk is considered valid
  - alice@gmail@.com is considered invalid
  - alice@gmailcom is considered invalid

Task 2: Analyzing the Email Address

We want to determine if the email address entered by a user is of a valid format or not.

- Write a C program that takes an email address as input (character by character) and verifies that it is of the form -----@----.uconn.----
  - I.e. It contains one and only one at sign '@'
  - It contains ".uconn." character sequence which succeeds the '@' sign
  - The special characters '@' and '.' cannot be consecutive, i.e. '@.', '@@', '..', and '.@' are invalid
  - Please read the input character by character, and implement a state machine to analyze the input.
- For example
  - alice@engr.uconn.edu is considered valid.
  - alice@mydomain.uconn.co.uk is also considered valid
  - alice@engr..uconn.edu is considered invalid
  - alice@engruconnedu is considered invalid

Department of Electrical and Computing Engineering

UNIVERSITY OF CONNECTICUT

ECE 3411 Microprocessor Application Lab: Fall 2017

Problem Set P0

There are 6 questions in this problem set. Answer each question according to the instructions given in at least 3 sentences in own words.

If you find a question ambiguous, be sure to write down any assumptions you make.
Be neat and legible. If we can't understand your answer, we can't give you credit!

Any form of communication with other students is considered cheating and will merit an F as final grade in the course.

SUBMIT YOUR ANSWERS IN PDF FORMAT

Do not write in the boxes below

1 (x/20)	2 (x/20)	3 (x/20)	4 (x/20)	5 (x/20)	Total (xx/100)

Name:

Student ID:

1. **[20 points]:** Answer the following questions (each one is allocated 5 points):

a. Name the loop which executes it's loop body atleast once?

b. In the code given below, is 'x' a global variable (i.e. it can be accessed anywhere in the program) ?

```
int main(void)
{
 int x;
 ...
 ..
}
```

c. Does the following code print an "Okay" ?

```
int main(void)
{
 int x = 1;
 if(--x == 0)
 printf("Okay");
}
```

d. Consider the following code snippet and give the output of test(12) and test(10)

```
double test(int x)
{
 return (x%4==0)?(x/8):((double)x/8);
}
```

Initials:

2. [20 points]: How many times will the statement called `loopBody` be executed in the following construct?

```
int a = 5;
int b= 10;
while (a > 1)
{
   for (int i = 0; i < b/a; i++)
      loopBody;
   a-=2;
}
```

Initials:

3. **[20 points]:** What is the output of the following code segment? Explain your answer.

```
int x = 28, d = 2;
while(x != 0)
{
      if(x % d != 0)
          d = d + 1;
      else
      {
          x = x / d;
          printf("%d\n", d);
          if(x == 1)
              break;
       }
}
```

Initials:

4. [20 points]: Explain the output of the following code snippet.Assume the user gives 14 as the input. In the snippet below,the bitwise operations on integers are performed on their 16 bit representation.

```
int i,j,count = 0;
scanf("%d", &i);
for(j = 0; j < 16; j++)
{
  if((i & (1 << j)) != 0)
    {
       count++;
    }
}
printf("%d\n", count);
```

Initials:

5. [20 points]: Write the C code for the following function which returns a random bit (‘1’ or ‘0’) with 75% probability for ‘1’ and 25% probability for ‘0’. Explain your answer.
Hint: You may use `rand()` to generate a random integer.This function returns an integer value between 0 and `RAND_MAX`, where `RAND_MAX` $= 32763 = 2^{15} - 1$

```
int get_rand_bit()
{
   int bit;









   return bit;
}
```

Initials:

End of Problem Set

Initials: