Embedded System (ES)

Lecturer: Dr. Cheng-Kai Lu
Phone: (02)7749-3554
Office: TD302/BAIR Lab
Email: cklu@ntnu.edu.tw

國立臺灣師範大學
NATIONAL TAIWAN NORMAL UNIVERSITY

Outline

In this lecture, we will cover:

- Introduction to Embedded System
- Simplest Embedded C program
  - Variables
  - Arrays
  - C-strings
- Grouping (17 groups, Next week)
- Raspberry Pi set distribution (next week or the week after next week)

Introduction

Embedded Systems Overview

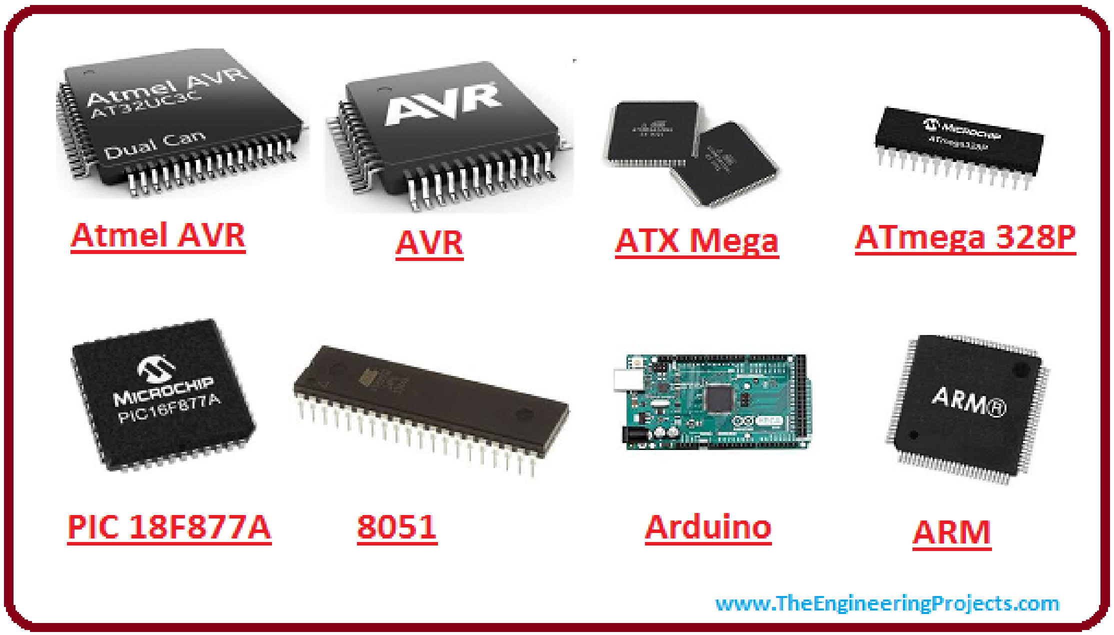
- An embedded system
  - employs a combination of hardware & software (a "computational engine") to perform a specific function;
  - is part of a larger system that may not be a "computer";
  - works in a reactive and time-constrained environment.
- Software is used for providing features and flexibility
- Hardware = {Processors, ASICs, Memory,...} is used for performance (& sometimes security)

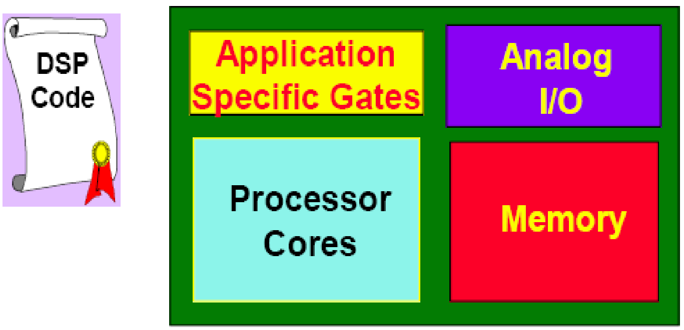
Due to their compact size, low cost and simple design aspects made embedded systems very popular and encroached into human lives and have become indispensable. They are found everywhere from kitchen ware to space craft. To emphasize this idea here are some illustrations.

ES + Microcontroller

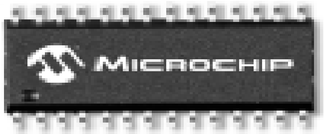
An embedded system is typically a design making use of the power of a microcontroller, like the Microchip PIC® MCU or dsPIC® Digital Signal Controller (DSCs).

These microcontrollers combine a microprocessor unit (like the CPU in a desktop PC) with some additional circuits called "peripherals", plus some additional circuits on the same chip to make a small control module requiring few other external devices. This single device can then be embedded into other electronic and mechanical devices for low-cost digital control.





Why Microcontroller ?



A microcontroller is a single silicon chip with memory and all Input/Output peripherals on it. Hence a microcontroller is also popularly known as a single chip computer. Normally, a single microcomputer has the following features :

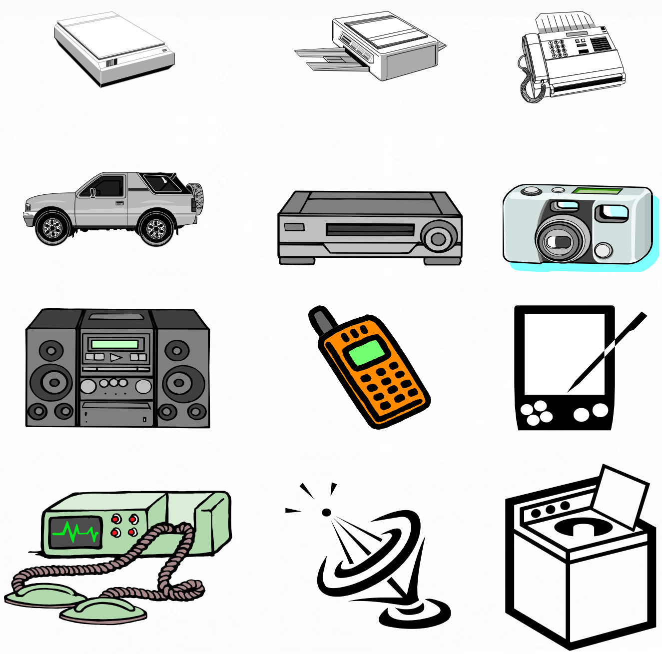
- Arithmetic and logic unit
- Memory for storing program
- EEPROM for nonvolatile data storage
- RAM for storing variables and special function registers
- Input/output ports
- Timers and counters
- Analog to digital converter
- Circuits for reset, power up, serial programming, debugging
- Instruction decoder and a timing and control unit
- Serial communication port

Microcontroller is the most sought-after device for designing an efficient ES

ES + IOT Applications

Anti-lock brakes
Auto-focus cameras
Automatic teller machines
Automatic toll systems
Automatic transmission
Avionic systems
Battery chargers
Camcorders
Cell phones
Cell-phone base stations
Cordless phones
Cruise control
Curbside check-in systems
Digital cameras
Disk drives
Electronic card readers
Electronic instruments
Electronic toys/games
Factory control
Fax machines
Fingerprint identifiers
Home security systems
Life-support systems
Medical testing systems

Modems
MPEG decoders
Network cards
Network switches/routers
On-board navigation
Pagers
Photocopiers
Point-of-sale systems
Portable video games
Printers
Satellite phones
Scanners
Smart ovens/dishwashers
Speech recognizers
Stereo systems
Teleconferencing systems
Televisions
Temperature controllers
Theft tracking systems
TV set-top boxes
VCR’s, DVD players
Video game consoles
Video phones
Washers and dryers



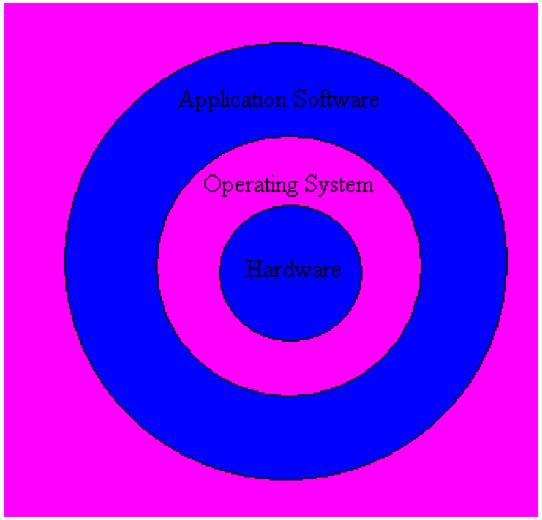
ES + IOT applications are everywhere …

And the list goes on and on…

What is inside an ES?

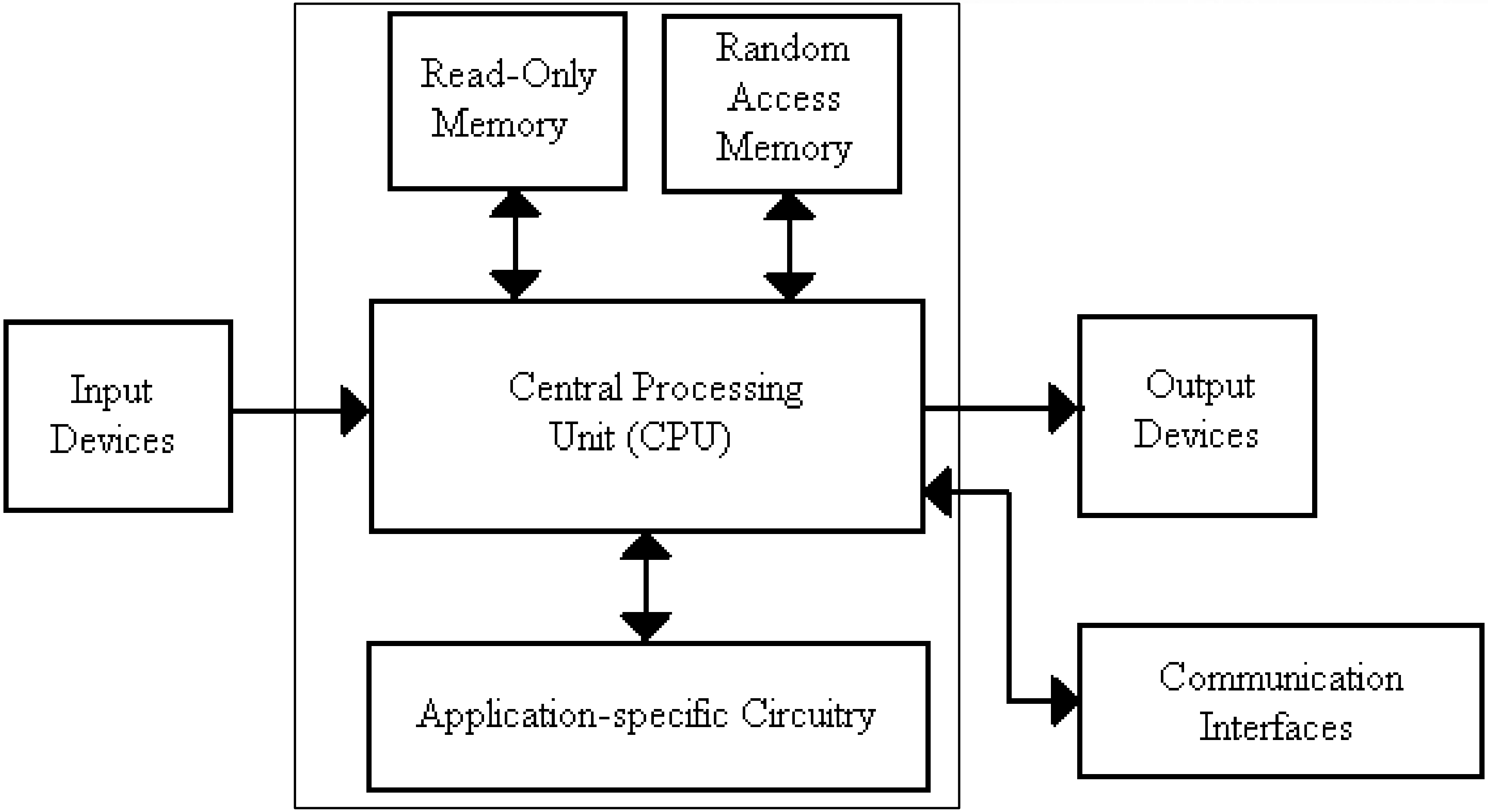
Every embedded system consists of **custom-built hardware** built *around* a Central Processing Unit (**CPU**). This hardware also contains memory chips onto which the software is loaded. The software residing on the memory chip is also called the '**firmware**'.

The operating system runs above the hardware, and the application software runs above the operating system. The same architecture is applicable to any computer including a desktop computer. However, there are significant differences. It is **not compulsory** to have an operating system **in every embedded system**.



Layered architecture of ES

Hardware architecture of ES



The co-design Concept

- *Hardware/software codesign* is a loose term that encompasses a large slice of embedded systems design, trade-off analysis, and optimization starting from the abstract function and architecture specification down to the detailed hardware and software implementation.
- Hardware/software codesign involves **analysis and trade-offs**: analyzing the hardware and software as they work together and discovering what adjustments or trade-offs you need to make to match your parameters. For example, anytime you debug a software driver on the hardware (or a model of the hardware), and you **tweak** the hardware or the software as a result, that's codesign.
- Hardware/software codesign is the enabler of the embedded systems revolution.

Common characteristics of ES

- Single-functioned
  - Executes a single program **repeatedly doing a specific task**.
- Tightly-constrained
  - have **very limited resources**, particularly the **memory**. Generally, they do not have secondary storage devices such as the CDROM or the floppy disk..
  - **constrained for power**, many ES operate through a battery, the power consumption must be very low.
- Reactive and real-time
  - Continually **react**s to changes in the system's **environment**
  - Must compute certain results in **real-time** without delay, else serious consequences.
- Embedded systems **need to be highly reliable**.
  - Once in a while, pressing ALT-CTRL-DEL is OK on your desktop, but you cannot afford to reset your embedded system in outer space!

Why Embedded Systems Must Avoid Failures

Since **embedded systems often operate in remote or inaccessible locations**, they need:

- **Fault tolerance** (ability to continue operating despite errors).
- **Self-recovery mechanisms** (such as watchdog timers).
- **Redundant systems** to prevent failures.

Classification of ES

Based on functionality and performance requirements, embedded systems are classified as :

- Stand-alone Embedded Systems
- Real-time Embedded Systems
- Networked Information Appliances
- Mobile Devices

Stand-alone ES

As the name implies, stand-alone systems work in stand-alone mode. They take inputs, process them and produce the desired output. The input can be electrical signals from transducers or commands from a human being such as the pressing of a button. The output can be electrical signals to drive another system, an LED display or LCD display for displaying of information to the users.

Embedded systems used in process control, automobiles, consumer electronic items etc. fall into this category.

Real-time ES

Embedded systems in which some **specific work** has **to be done in a specific time period** are called real-time systems. For example, consider a system that has to open a valve within 30 milliseconds when the humidity crosses a particular threshold. If the valve is not opened within 30 milliseconds, a catastrophe may occur. Such systems with **strict** deadlines are called ***hard real-time*** systems.

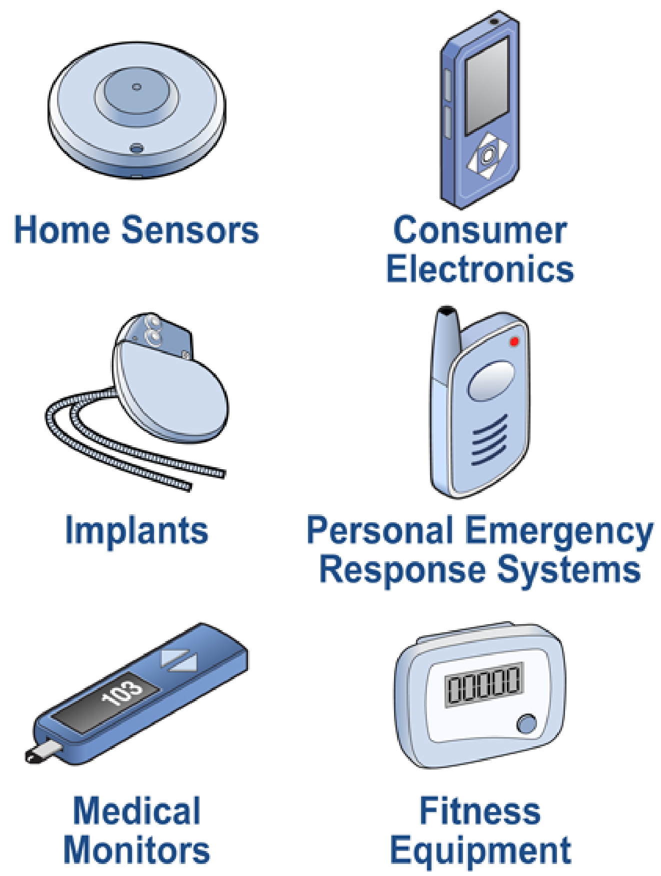
In some embedded systems, deadlines are imposed, but **not adhering to** them once in a while may **not** lead to **a catastrophe**. For example, consider a DVD player. Suppose, you give a command to the DVD player from a remote control, and there is a delay of a few milliseconds in executing that command. But this delay won't lead to a serious implication. Such systems are called ***soft real-time*** systems .

Networked Information Appliances

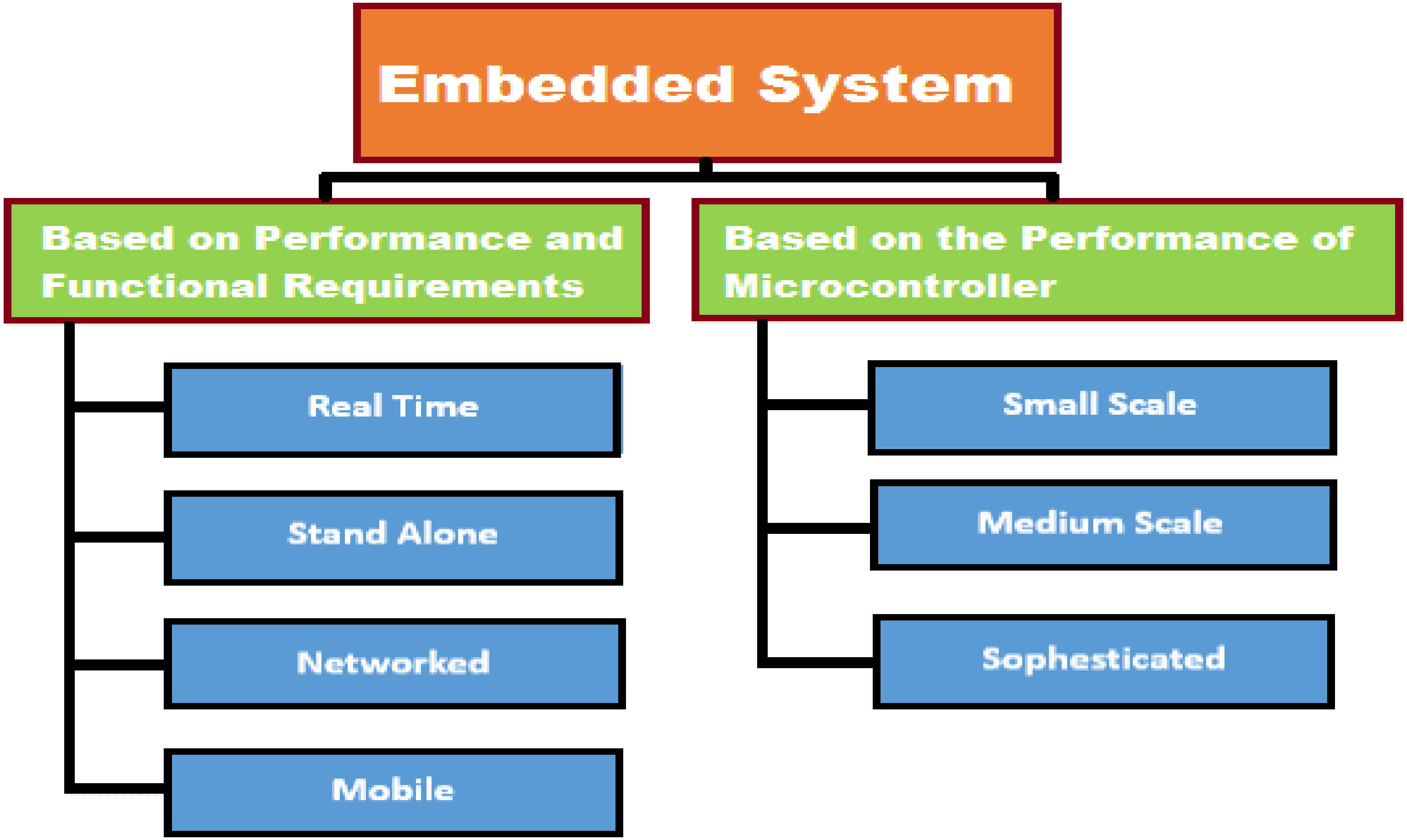
ES that are provided with network interfaces and accessed by networks such as Local Area Network or the Internet are called networked information appliances. Such ES are connected to a network, typically a network running TCP/IP (Transmission Control Protocol/Internet Protocol) protocol suite, such as the Internet or a company's Intranet. These systems run the protocol TCP/IP stack and get connected through PPP or Ethernet to a network and communicate with other nodes in the network. Any computer connected to the Internet cans access the system to obtain real-time data.

Mobile Devices

Mobile devices supporting special category of ESs where they need to be designed just like the 'conventional' ES.



Other Variants



ES Programming Language

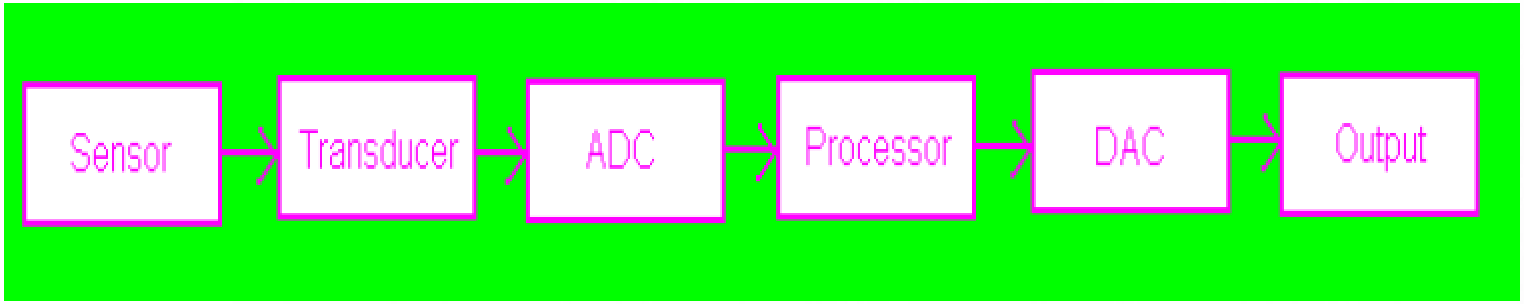
Assembly language was the pioneer for programming ES till recently. Nowadays there are many more languages to program these systems. Some of the languages are **C, C++,** Ada, Forth, **Python,** and Java together with its new enhancement J2ME.

C is very close to **assembly programming** and it allows very easy access to underlying hardware. A huge number of high-quality compilers and debugging tools are available for the C language.

Most of the software for ES is still done **in C** language. Recent survey indicates that approximately 45% of the embedded software is still being done in C language.

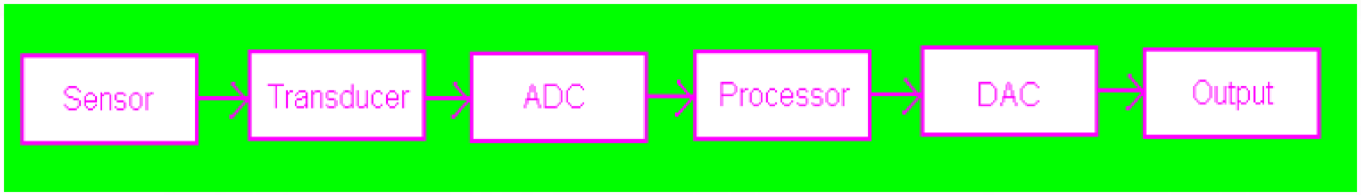
Simple Case Study

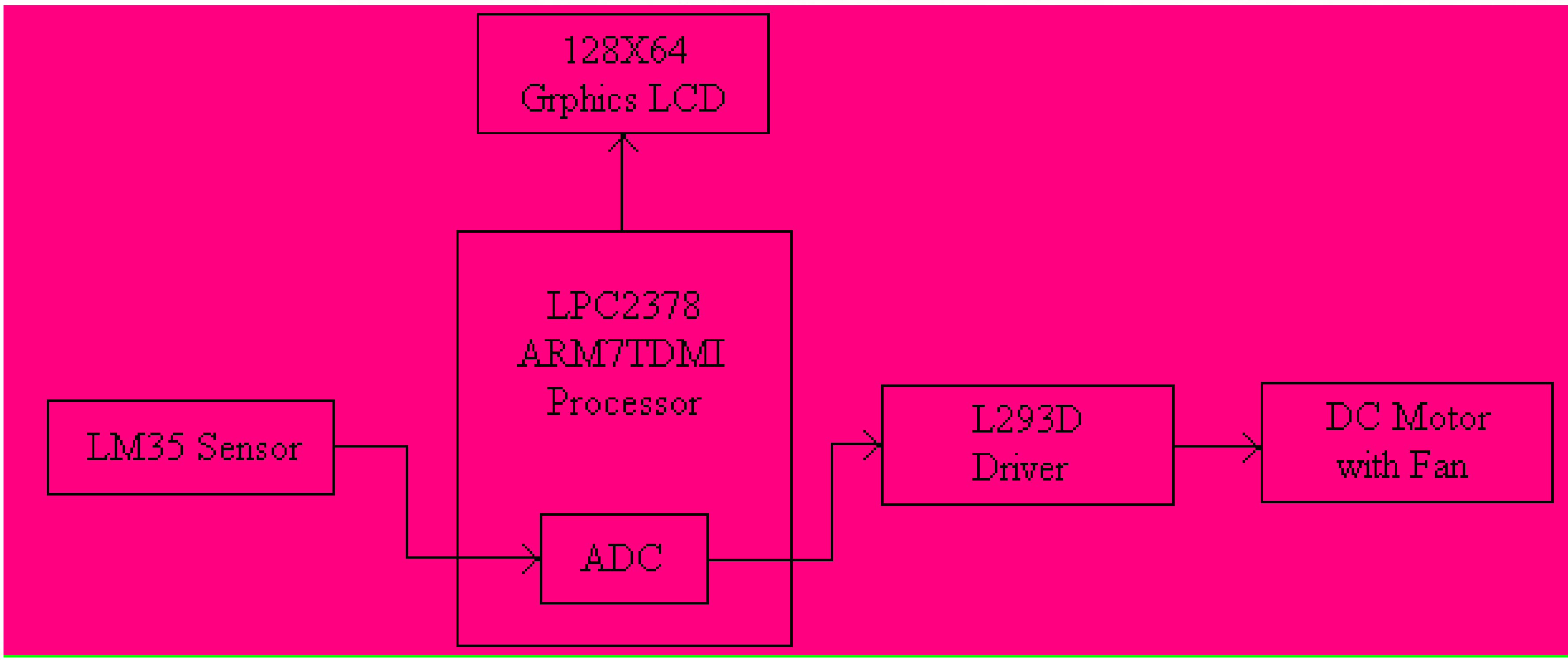
To understand the design of a simple embedded system let us consider the idea of a data acquisition system:



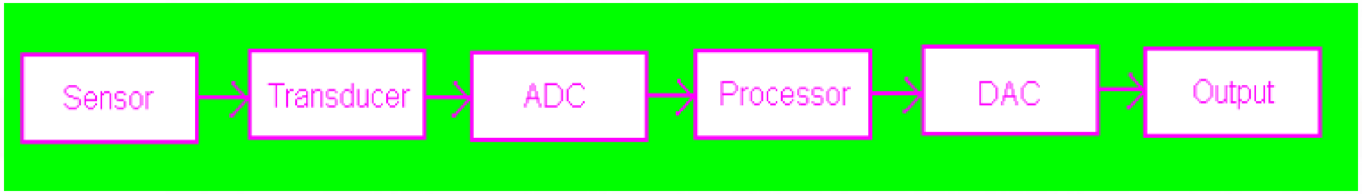
A conceptual idea of temperature measurement ES

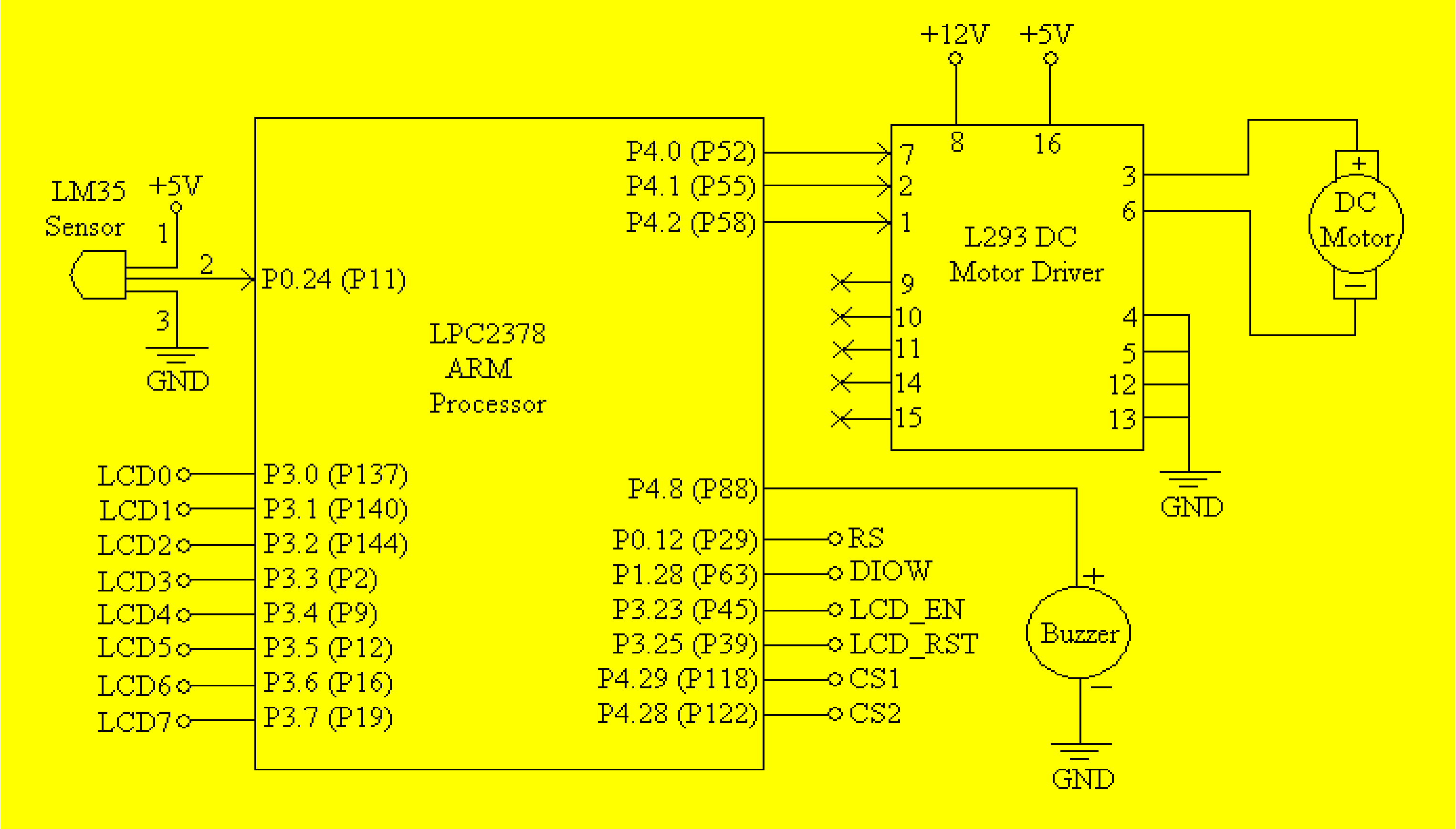
Block Diagram





Implementation





Summary

- Embedded Systems overview
- ES + Microcontroller
- What is ES
- Common characteristics of ES
- Classification of ES
- ES programming languages
- Co-design concept
- Case study

EMBEDDED SYSTEMS

Methods for Representing Data

- Bit
  - 1 (True)
  - 0 (False)
- Nibble (less commonly used)
  - 4 bits
- Byte
  - 8 bits
- N-byte Words
  - 2-byte / 16-bit Word, 4-byte / 32-bit Word

Methods for Representing Data

- Three of the most common forms of notation
  - Decimal (base 10) 0123456789
  - Hexadecimal (base 16) 0123456789ABCDEF
  - Binary (base 2) 01
- Converting between forms
  - When converting binary to hexadecimal, every group of 4 bits (nibble) represents a hexadecimal digit
  - Examples:

Binary	Hexadecimal
0010	2
0100	4
1010	A

Base Conversion (by hand)

- Base n to base 10

Problem: Convert base 2: 0b0100_1011, to base 10

Solution:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128's	64's	32's	16's	8's	4's	2's	1's
0	1	0	0	1	0	1	1

64 + 8 + 2 + 1 = 75

Base Conversion (by hand)

- Base 10 to base n

Problem: Convert 175 to base 16

Solution:

Create a table of the columns in a base 16 number and subtract from the original number:

16^1	16^0
16's	1's
A	

175 – 160 = 15

16^1	16^0
16's	1's
A	F

Base in C

- Syntax in C
  - Computers understand binary
  - The following lines of code are **all the same** (the complier does not care what base the programmer uses):

```
char x = 2 + 1;
char x = 0b10 + 1;
char x = 0x2 + 1;
char x = 0x02 + 0x01;
```

What are Embedded Systems?

- Your Definition?
- What are some properties of an Embedded System?

Quadcopter

Micro SD Card?

Blu-Ray / Remote

Programmable Thermostat

Roomba

C IN EMBEDDED SYSTEMS

BASICS

C

- **C** is a **procedural** language
  - No classes or objects
  - “**Function**” is the building block
- C structure: Uses a minimum set of language constructs
- “The C programming language” (Library has web version)
  - Quick Overall Intro: Chapter 1 (pgs 5 – 34)
  - Chapter 2
- Course Webpage: Resources sections
  - “The C Book”: http://publications.gbdirect.co.uk/c_book/

Simplest Embedded Program

```
void main()
{
  while (1)
  {

  }  // do forever…
}
```

- Embedded programs often run forever

Simplest Embedded Program

```
#include <stdio.h>

void main()
{
    printf(“hello, world\n”);
}
```

VARIABLES IN C

Variables

- **Variables** are the primary mechanism for **storing data** to be processed by your program
- Examples:
  - area, graph, distance, file1, file2, height, wheel_right
- Must not be a reserved keyword (next slide)
- Good practice: use descriptive variable names
  - Good names: height, input_file, area
  - Bad names: h, if, a
- Rule of thumb: Always code as though the person maintaining your code knows where you sleep

Reserved Words: Primitive Data Types

- **char**
- **short**
- **int**
- **long**
- **double**
- **float**

- enum
- struct
- union
- typedef

- break
- case
- continue
- default
- do
- else
- for
- goto
- if
- return
- switch
- while

- auto
- const
- extern
- register
- signed
- static
- unsigned
- volatile

- sizeof

- void

Variables

- Variables must be *declared* by specifying the variable's **name** and the **type** of information that it will hold

data type **variable name**

```
int total;
int count, temp, result;
```

Multiple variables can be created in one declaration

Variables

- A variable can be given an initial value in the declaration

```
int sum = 0;
int base = 32, max = 149;
```

- If no initial value is given, **do** **not** assume the default value is 0

```
int k, i;
for (i = 0; i < 10; i++)
{
    k = k + 1;
}
printf("%d", k);
```

Primitive Types and Sizes

Name	Number of Bytes sizeof()	Range
char	1	0 to 255 or -128 to 127 (Depends on Compiler settings)
signed char	1	-128 to 127
unsigned char	1	0 to 255
short	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
int	Varies by platform	Varies by platform
int (on TM4C123)	4	-2,147,483,648 to 2,147,483,647
unsigned int (on TM4C123)	4	0 to 4,294,967,295
(pointer)	Varies by platform	Varies by platform
(pointer on TM4C123)	4	Address Space

- Primitive types in C: char, short, int, long, float, double **default** modifier on primitive types is **signed** (not unsigned)
- Note: char does not have a standard default, it depends on Compiler settings

Primitive Types and Sizes

Name	Number of Bytes sizeof()	Range
long	4	-2,147,483,648 to 2,147,483,647
signed long	4	-2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295
long long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	±1.175e-38 to ±3.402e38
double	Varies by platform	
double (on TM4C123)	8	±2.3E-308 to ±1.7E+308

- Primitive types in C: char, short, int, long, float, double default modifier on primitive types is **signed** (not unsigned)
- Note: char does not have a standard default, it depends on **Compiler settings**

Why This Matters in Embedded Systems

- Choosing the correct data type size is crucial for memory efficiency and performance in embedded systems.
- Using unsigned types when negative values are unnecessary helps prevent overflow.
- Always check microcontroller documentation for exact type sizes.

Key Takeaways for Embedded Systems

Use Case	Best Data Type	Reason
Temperature sensor (-40 to 125°C)	`int8_t` or `signed char` (1 byte)	Saves memory
Digital pin state (0 or 1)	`bool` or `uint8_t` (1 byte)	No need for extra bytes
Millisecond timer (50+ days)	`uint32_t` (4 bytes)	Prevents overflow
Serial communication buffer (UART, SPI)	`uint8_t` array	Best for raw data
12-bit ADC data	`uint16_t` (2 bytes)	Matches ADC resolution

Tips for Embedded Systems Developers

- ALWAYS check the microcontroller datasheet for type sizes.
- MINIMIZE memory usage when working with RAM-limited systems.
- USE stdint.h (uint8_t, uint16_t, etc.) for portability across different compilers.

Variables: Size

```
char    sum_char  = 0;
short   sum_short = 0;
int     sum_int   = 0;
```

- sum_char value is a 8-bit value:
  - Binary: 0b0000 0000
  - Hex: 0x00
- sum_short value is a 16-bit value:
  - Binary: 0b0000 0000 0000 0000
  - Hex: 0x0000
- sum_int value is a 32-bit value:
  - 0b0000 0000 0000 0000 0000 0000 0000 0000
  - Hex: 0x0000 0000

Variables: Size

```
unsigned char  my_number   = 255;
unsigned char  my_number_too_big = 257;
```

- my_number in:
  - Binary: 0b1111 1111
  - Decimal: 255
- my_number_too_big in:
  - Binary: 0b1 0000 0001
  - Decimal:

Variables: Size

```
unsigned char  my_number   = 255;
unsigned char  my_number_too_big = 257;
```

- my_number in:
  - Binary: 0b1111 1111
  - Decimal: 255
- my_number_too_big in: // Need 9-bits, too big for a unsigned char.
  - Binary: 0b0000 0001 // the C compiler will truncate to 8-bits
  - Decimal: 1

ARRAYS IN C

Arrays in C

- **Sequence** of a **specific** variable type stored in **memory**
- **Zero-indexed** (starts at zero rather than one)
- Define an array as

  Type *VariableName* [ArraySize];

  Example: int my_array[100] **Size: i.e. Number of elements**

  data type **variable name**
- Last element is found at *N-1* location
- Curly brackets can be used to initialize the array

Arrays in C

- Examples:

```
// allocates and initializes 3 chars’s
char myarray1[3] = {2, 9, 4};

// allocates memory for 4  char’s
char myarray2[4];


// allocates memory for 2  short’s
short myarray3[2];
```

Arrays in C

- Examples:

```
char  myarray1[3] = {2, 9, 4};
char  myarray2[4];
short myarray3[2];
```

- When defining an array, the array name is the address in memory for the first element of the array
  - `myarray3 == 0xFF07`

Memory Address FF00 FF01 FF02 FF03 FF04 FF05 FF06 FF07 FF08 FF09 FF0A

Value 0x02 0x09 0x04 ? ? ? ? ? ? ??

Array myarray1 myarray2 myarray3

Index 0 1 2 0 1 2 3 0 1

Arrays in C

- Examples:

```
char  myarray1[3] = {2, 9, 4};
char  myarray2[4];
short myarray3[2];
```

```
myarray1[0] // First element of myarray1
myarray1[2] // Last element of myarray1
```

Memory Address: FF00 FF01 FF02 FF03 FF04 FF05 FF06 FF07 FF08 FF09 FF0A

Value: 0x02 0x09 0x04 ? ? ? ? ? ? ??

Array: myarray1 myarray2 myarray3

Index: 0 1 2 0 1 2 3 0 1

Arrays

- Be careful of boundaries in C
  - No guard to prevent you from accessing beyond array end
  - **Write beyond array**

    => Potential for disaster:

    1. Unexpected behavior (random crashes, corrupted data).

    2. Hard-to-debug bugs in embedded systems.

    3. System crashes (if modifying memory used by the OS or hardware registers)
- No built-in mechanism for copying arrays

Arrays in C

- Examples:

```
char  myarray1[3] = {2, 9, 4};
char  myarray2[4];
short myarray3[2];
```

```
myarray1[3] // Passed end of myarray1!!!
            // Overwrote myarray2!!
```

Memory Address: FF00 FF01 FF02 FF03 FF04 FF05 FF06 FF07 FF08 FF09 FF0A

Value: 0x02 0x09 0x04 ? ? ? ? ? ? ??

Array: myarray1 myarray2 myarray3

Index: 0 1 2 0 1 2 3 0 1

*No compiler error! But the program may crash or produce unexpected results.

Arrays in C

- Examples:

```
char  myarray1[3] = {2, 9, 4};
char  myarray2[4];
short myarray3[2];
```

```
myarray1[9] = 0x32;    ???
```

Memory Address	FF00	FF01	FF02	FF03	FF04	FF05	FF06	FF07	FF08	FF09	FF0A
Value	0x02	0x09	0x04	?	?	?	?	?	?	??	
Array		myarray1				myarray2				myarray3	
Index	0	1	2	0	1	2	3	0		1	

Arrays in C

- Examples:

```
char  myarray1[3] = {2, 9, 4};
char  myarray2[4];
short myarray3[2];
```

```
myarray1[9] = 0x32; (update the memory map)
```

Memory Address	FF00	FF01	FF02	FF03	FF04	FF05	FF06	FF07	FF08	FF09	FF0A
Value	0x02	0x09	0x04	?	?	?	?	?	?	0x32	?
Array	myarray1			myarray2				myarray3			
Index	0	1	2	0	1	2	3	0		1	

Safe Array Access with Bounds Checking

```
#include <stdio.h>

#define ARRAY_SIZE 3  // Define the correct array size

int main() {
    char myarray1[ARRAY_SIZE] = {2, 9, 4};
    int index = 3; // Simulating user input or a calculation

    if (index >= 0 && index < ARRAY_SIZE) {
        myarray1[index] = 10; // Safe access
    } else {
        printf("Error: Index out of bounds!\n");
    }
    return 0;
}
```

- Prevents out-of-bounds access.
- Recommended for embedded systems where reliability is critical.

Array Copy Example

```
int  TestArray1[20];  // An array of 20 integers
int  TestArray2[20];  // An array of 20 integers

TestArray1 = TestArray2; // This does not “copy” !!!

for (int i = 0; i < 20; i++)
{
   TestArray1[i] = TestArray2[i]; // This copies
}
```

STRINGS IN C

Character Strings in C

- There are **no Strings** in C like in Java (there are no classes)
- Strings are represented as char arrays
- **char** is a primitive data type
  - stores 8 bits of data, not necessarily a character
  - can be used to store small numbers

  e.g. a. A character (e.g., 'A', 'B’).

  b. A small integer (0 to 255 if unsigned).
- A string of characters can be represented as a *string literal* by putting double quotes around the text:
- Examples:

```
char str1[] = "This is a string literal.";
char str2[] = "123 Main Street";
char str3[] = "X";  // Valid string (stored as {'X', '\0'})
char ch = 'X';  // A single character, NOT a string!
```

Character Strings in C

- The **end** of a string (char array) is signified by a **null byte**
  - String literals (i.e. “some text“) have an automatic null byte included
  - Null bytes is a byte with a value of 0
- str1, str2, and str3 below each consume 4 bytes of memory and are equivalent in value:

```
char* str1 = “123”;  // pointer (will introduce it later)
char str2[] = “123”;
char str3[4] = {‘1’, ‘2’, ‘3’,  0};
```

Character Strings in C

- **Do not** use statements like: *if (str1 == str2)* to test equality
  - Again: str1, str2, and str3 are the address of the first char in each array.
  - Use a function like *strcmp* to test if char arrays are equivalent

```
char str1[] = “123”;
char str2[] = “123”;

if (strcmp(str1, str2) == 0)
{
   // str1 matches str2
}
```

Character Strings in C

- Each character is encoded in 8 bits using ASCII:
- The following statements are equivalent:

```
char str[] = “hi”;
char str[3] = { ‘h’, ‘i’, ‘\0’ };
char str[3] = { 104, 105, 0 };
Char str[3] = {0x68, 0x69, 0x0};
```

Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	SP
010 0001	041	33	21	!
010 0010	042	34	22	"
010 0011	043	35	23	#
010 0100	044	36	24	$
010 0101	045	37	25	%
010 0110	046	38	26	&
010 0111	047	39	27	'
010 1000	050	40	28	(
010 1001	051	41	29	)
010 1010	052	42	2A	*
010 1011	053	43	2B	+
010 1100	054	44	2C	,
010 1101	055	45	2D	-
010 1110	056	46	2E	.
010 1111	057	47	2F	/
011 0000	060	48	30	0
011 0001	061	49	31	1
011 0010	062	50	32	2
011 0011	063	51	33	3
011 0100	064	52	34	4
011 0101	065	53	35	5
011 0110	066	54	36	6
011 0111	067	55	37	7
011 1000	070	56	38	8
011 1001	071	57	39	9
011 1010	072	58	3A	:
011 1011	073	59	3B	;
011 1100	074	60	3C	<
011 1101	075	61	3D	=
011 1110	076	62	3E	>
011 1111	077	63	3F	?

Binary	Oct	Dec	Hex	Glyph
100 0000	100	64	40	@
100 0001	101	65	41	A
100 0010	102	66	42	B
100 0011	103	67	43	C
100 0100	104	68	44	D
100 0101	105	69	45	E
100 0110	106	70	46	F
100 0111	107	71	47	G
100 1000	110	72	48	H
100 1001	111	73	49	I
100 1010	112	74	4A	J
100 1011	113	75	4B	K
100 1100	114	76	4C	L
100 1101	115	77	4D	M
100 1110	116	78	4E	N
100 1111	117	79	4F	O
101 0000	120	80	50	P
101 0001	121	81	51	Q
101 0010	122	82	52	R
101 0011	123	83	53	S
101 0100	124	84	54	T
101 0101	125	85	55	U
101 0110	126	86	56	V
101 0111	127	87	57	W
101 1000	130	88	58	X
101 1001	131	89	59	Y
101 1010	132	90	5A	Z
101 1011	133	91	5B	[
101 1100	134	92	5C	\
101 1101	135	93	5D	]
101 1110	136	94	5E	^
101 1111	137	95	5F	_

Binary	Oct	Dec	Hex	Glyph
110 0000	140	96	60	`
110 0001	141	97	61	a
110 0010	142	98	62	b
110 0011	143	99	63	c
110 0100	144	100	64	d
110 0101	145	101	65	e
110 0110	146	102	66	f
110 0111	147	103	67	g
110 1000	150	104	68	h
110 1001	151	105	69	i
110 1010	152	106	6A	j
110 1011	153	107	6B	k
110 1100	154	108	6C	l
110 1101	155	109	6D	m
110 1110	156	110	6E	n
110 1111	157	111	6F	o
111 0000	160	112	70	p
111 0001	161	113	71	q
111 0010	162	114	72	r
111 0011	163	115	73	s
111 0100	164	116	74	t
111 0101	165	117	75	u
111 0110	166	118	76	v
111 0111	167	119	77	w
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z
111 1011	173	123	7B	{
111 1100	174	124	7C	|
111 1101	175	125	7D	}
111 1110	176	126	7E	~

Character Strings in C

- Examples:

```
char myword1[6] = “Hello”;  //  declare and initialize
char myword2[4]   = “288”;  //  declare and initialize
```

Memory Address	DF00	DF01	DF02	DF03	DF04	DF05	DF06	DF07	DF08	DF09
	myword1						myword2			
Value	‘H’	‘e’	‘l’	‘l’	‘o’	‘\0’	‘2’	‘8’	‘8’	‘\0’
Array Index	0	1	2	3	4	5	0	1	2	3

Character Strings in C

- Examples:

```
char myword1[6] = “Helloo”;  //  declare and initialize
char myword2[4]    = “288”;  //  declare and initialize

Note: myword1[6] does not give room for the NULL byte.
```

Memory Address	DF00	DF01	DF02	DF03	DF04	DF05	DF06	DF07	DF08	DF09
	myword1						myword2			
Value	‘H’	‘e’	‘l’	‘l’	‘o’	‘o’	‘2’	‘8’	‘8’	‘\0’
Array Index	0	1	2	3	4	5	0	1	2	3

Escape Sequences

- What if we wanted to print the quote character?
- The following line would **confuse the compiler** because it would interpret the second quote as the end of the string:

```
char str[] = "I said "Hello" to you.";
```

- An *escape sequence* is a series of characters that represents a special character
- An escape sequence begins with a **backslash** character (\)

```
char str[] = "I said \"Hello\" to you.";
```

Escape Sequences

Binary	Oct	Dec	Hex	Abbr	Carrot	Escape	Description
000 0000	0	0	0	NUL	^@	\0	Null character
000 0111	7	7	7	BEL	^G	\a	Bell
000 1000	10	8	8	BS	^H	\b	Backspace
000 1001	11	9	9	HT	^I	\t	Horizontal Tab
000 1010	12	10	0A	LF	^J	\n	Line feed
000 1011	13	11	0B	VT	^K	\v	Vertical Tab
000 1100	14	12	0C	FF	^L	\f	Form feed
000 1101	15	13	0D	CR	^M	\r	Carriage return
001 1011	33	27	1B	ESC	^[	\e	Escape
010 0111	47	39	27	'		\'	Single Quote
010 0010	42	34	22	"		\"	Double Quote
101 1100	134	92	5C	\		\\	Backslash

Formatting Strings

- *printf, sprintf, fprintf* = standard library functions for printing data into char arrays
- Must include stdio.h in order to use these function

```
#include <stdio.h>
```

- These functions have an argument called a formatter string that accepts % escaped variables
- Review the documentation on functionality of *sprintf*
  - Google "sprintf", first result is:
  - http://www.cplusplus.com/reference/clibrary/cstdio/sprintf/

Formatting Strings: Example % formats

- *This table can be found in many places on the Internet*

Character	Argument type; Printed As
`d,i`	`int`; decimal number
`o`	`int`; unsigned octal number (without a leading zero)
`x,X`	`int`; unsigned hexadecimal number (without a leading `0x` or `0X`), using `abcdef` or `ABCDEF` for 10, ...,15.
`u`	`int`; unsigned decimal number
`c`	`int`; single character
`s`	`char *`; print characters from the string until a `'\0'` or the number of characters given by the precision.
`f`	`double`; `[-]`*m.dddddd*, where the number of *d*'s is given by the precision (default 6).
`e,E`	`double`; `[-]`*m.dddddd*`e+/-`*xx* or `[-]`*m.dddddd*`E+/-`*xx*, where the number of *d*'s is given by the precision (default 6).
`g,G`	`double`; use `%e` or `%E` if the exponent is less than -4 or greater than or equal to the precision; otherwise use `%f`. Trailing zeros and a trailing decimal point are not printed.
`p`	`void *`; pointer (implementation-dependent representation).
`%`	no argument is converted; print a %

Formatting Strings: Example % formats

int age = 18;

int course = 288;

char message[] = "Hello World";

char short_msg[5] = {'H', 'I'};

printf("My age is **%d**", age);

// gives: My age is 18

printf("Say **%s** my age is %d", message, age);

//gives: Say Hello World my age is 18

printf("Hi is spelled **%c %c**, in class **%d**", short_msg[0], short_msg[1], course)

//gives: Hi is spelled H I, in class 288

STRING MANIPULATION

String Manipulation Functions

- int sprintf(char * str, const char * format, ...);
  - Formats a string into a character array
- int strlen(const char * str);
  - Finds the length of a string.
- int strncmp(const char * str1, const char * str2, size_t num);
  - Compares two strings for a given number of characters.

String Manipulation Functions: sprintf (if with terminal)

int sprintf (char * str, const char * format, ...);

Param1: location to store the string (e.g. character array)

Param2: formatted string to store in the array

Param3-n: formatting variables that appear in the formatted string.

Example:

```
int class_num = 288;
char my_array[20];
char another_array[10] = “Goodbye”
sprintf(my_array, “Hello CPRE %d \n”, class_num);
// my_array now contains: Hello CPRE 288
printf(“%s”, another_array);   // prints (standard I/O function) Goodbye
```

String Manipulation Functions: strlen

int strlen (const char * str);

Param1: location of a string (e.g. character array name)

Return value: returns the length of the string (not counting NULL byte).

Example:

```
char my_array[20] = "Hello CPRE288";
int     my_len = 0;
my_len = strlen(my_array);

// my_len now has a value of 13
```

***Useful in embedded systems for memory-efficient string handling**

String Manipulation Functions: strcmp

int strcmp (const char * str1, const char * str2,);

Param1: location of a string

Param2: location of a string

Return value: if equal then 0, if the first position that does not match is greater in str1 then +, else -.

Example:

char my_array1[20] = “apple”;

char my_array2[20] = “pair”;

int my_compare = 0;

my_compare = strcmp(my_array1, my_array2);

// ‘a’ has a lower value than ’p’, so my_compare will be negative

✓ **Compares strings alphabetically using ASCII values.**

✓ **Commonly used in sorting and searching applications in embedded systems.**

Class Activity_ Example

- Predict the value of *message* after each line:

char str1[] = “hello”;

char str2[] = “world”;

char message[100];

Summary of Key Takeaways

Function	Purpose	Example
`sprintf()`	Formats a string into a character array	`sprintf(buffer, "Value: %d", 42);`
`strlen()`	Returns the length of a string	`int len = strlen("hello"); // len = 5`
`strcmp()`	Compares two strings	`strcmp("apple", "banana"); // Returns negative`

Key Differences of Strings in Embedded C_(1)

In Embedded C, handling character strings (char[]) is slightly different from general-purpose C programming due to memory constraints, real-time processing, and hardware limitations.

1. Avoid dynamically allocated strings (malloc, free)
– Many embedded systems lack dynamic memory management.
2. Use fixed-size character arrays – Helps prevent memory overflow and unpredictable behavior.
3. Store constant strings in Flash memory (const char[]) – Saves valuable RAM.

char myString[] = "Hello, Embedded C!"; % Static String in RAM

- Stored in RAM (volatile memory).
- Takes up extra space since it is modifiable.
- Uses RAM unnecessarily if the string does not change.

const char myString[] = "Hello, Embedded C!";

✓ **Stored in Flash (ROM)** instead of RAM.
✓ **Prevents modification**, saving RAM.
✓ **More efficient for microcontrollers.**

Key Differences of Strings in Embedded C_(2)

a. Use const char *str: This prevents modification of the string, which is efficient for embedded systems.
b. Send one character at a time: Embedded systems often send data via UART, SPI, or I2C, not printf().
c. Some embedded applications require storing strings permanently (e.g., device settings, logs). -> Storing Strings in EEPROM (Non-Volatile Memory)
   ✓ **EEPROM retains data even after power loss.**
   ✓ **Useful for logging device status, serial numbers, and configurations.**

Key Differences of Strings in Embedded C_(3)(pp.75)

```
#include <stdio.h>
#include <string.h>
// Function to send a string over UART (Replace with your MCU's UART function)
void UART_sendString(const char *str) {
   while (*str) {  // Loop until null terminator '\0'
      while (!(UCSR0A & (1 << UDRE0)));  // Wait for UART buffer to be empty
      UDR0 = *str;  // Send character
      str++;  // Move to next character
   }
}
int main() {
   int class_num = 288;
   char my_array[20];
   char another_array[10] = "Goodbye";

   // Store formatted string into buffer
   sprintf(my_array, "Hello CPRE %d\n", class_num);

   // Send formatted string via UART (since printf is unavailable)
   UART_sendString(my_array);

   // Send another_array via UART
   UART_sendString(another_array);

   while (1);  // Keep running (common in embedded systems)
}
```

Example: Display "Hello" on an LCD (Optional)

```
#include <avr/io.h>
#include <util/delay.h>

void LCD_sendCommand(char cmd);
void LCD_sendChar(char c);
void LCD_sendString(const char *str);

int main() {
    LCD_sendString("Hello, World!");
    while (1);
}

void LCD_sendString(const char *str) {
    while (*str) {
        LCD_sendChar(*str++);
    }
}
```

✓ Sends a string to an LCD using **manual character-by-character transmission**.
✓ **Works efficiently in microcontrollers**.

STRUCTs in C

struct

- The **struct** type allows a programmer to define a **compound data/**custom data type.
- The size of a stuct is the size of its components added together.

```
struct RGB
{
  char red;
  char green;
  char blue;
};

struct RGB my_color;   // Declare a variable of struct type
my_color.blue = 255;   // Set the blue component to 255

struct RGB *my_color_ptr = &my_color;  // Pointer to the struct
(*my_color_ptr).blue = 255;   // Accessing via pointer dereferencing
my_color_ptr->blue = 255;     // equivalent to previous line
```

- http://en.wikipedia.org/wiki/Struct_(C_programming_language)

✓ The arrow (->) operator is a shortcut for accessing struct members through a pointer.
✓ This is commonly used in embedded systems for efficient memory handling.

Struct for arrays to store multiple records

```
struct student
{
   char name[30];
   int ID;
};
```

```
struct student student_records[100];  // Array of 100 students
```

```
// Set  student ID at index 10
student_records[10].ISUID = 5678;
```

struct

```
struct sensor
{
  float distance;
  unsigned char bumpLeft;
  unsigned char bumpRight;
};

struct sensor my_sensor;  // Declare a struct
struct sensor *my_ptr = &my_sensor;// Pointer to the struct

float my_distance;
// Access Distance
my_distance = my_sensor.distance;  // Access directly
my_distance = my_ptr->distance  // Access using a pointer
```

✓ Efficient way to store multiple sensor readings.
✓ Used in robotics, IoT, and automotive systems.

union (only one member can store data at a time)

- http://en.wikipedia.org/wiki/C_language_union

Union: Merge multiple components

```
union u_tag {
   int ival;       // size 4 bytes
   short sval; // size 2 bytes
   float fval;    // size 4 bytes
};
```

The size of a union variable is the size of its maximum component.

* The members of union can be accessed using Dot Operator (.)

union

union: Merge multiple components

```
union u_tag {
  int ival;      // size 4 bytes
  short sval; // size 2 bytes
  float fval;    // size 4 bytes
};
```

The size of a union variable is the size of its maximum component.

This example the size is 4, since the largest component is 4 bytes

✓ Saves memory compared to struct, making it useful in embedded systems.

Structure and Union

Use of union inside of a struct

```
struct {
  char *name;
  int flags;
  short s_type;
  union {
    short val;
    float fval;
    char  cval;
  } u;
} symtab;
```

How large is the struct symtab?

Use of union inside of a struct

```
struct {
  char *name;      4
  int flags;       4
  short s_type;    2
  union {
    short val;     2
    float fval;    4 //largest member of union u
    char  cval;    1
  } u; //largest member defines a union's size
} symtab;
    Just sum the size of each struct member.
    symtab size is: 4+4+2+4 = 14 bytes
```

Key features of structure and union

	structure	union
Define	Keyword ‘struct’ is used to define a structure	Keyword ‘union’ is used to define a union
Size	The size of a structure is the sum of the size of all data members and the packing size.	The size of the union is the size of its data member, which is the largest in size.
Memory management	Inefficient and requires packing memory	Efficient

Key features of structure and union

	structure	union
Data members	All data members store some value at every point in the program	Only the latest initialized data member stores the value.
Memory allocation	All the data members are provided appropriate memory at different memory addresses	All data members share a single memory address and occupy the memory of the largest data member
Initialisation of data members	All the data members could be initialised at once	Only one data member can be initialized at a time.
Updation	Updation of every data member is independent of the value stored in other data members	Updation of any data member leads to changes in values stored in other data members.
Data members value	We can access data members' exact values if they are initialized.	Only the latest initialized data member returns its exact value. All other data members return garbage values.
Accessing data members	'.' operator is used to access the data members	'.' operator is used to access the data members

Key Takeways

Feature	Union (`union`)	Structure (`struct`)
Memory Allocation	All members share the same memory	Each member gets separate memory
Size	Size of the largest member	Sum of all members' sizes
Usage	Memory-efficient when only one member is used at a time	Stores multiple values simultaneously

Exercises

A) The members of a structure/union can be accessed using ___.

1.Dot Operator (.)
2.And Operator (&)
3.Asterisk Operator (*)
4.Right Shift Operator (>)

B) Which of these is a user-defined data type in C?

1.int
2.union
3.Structure
4. union & struture
5.All of these

C) "A struct can contain data of different data types". True or False?

1.True
2.False

D) Which of the below statements are incorrect in case of union?

1. Union is a user-defined data structure
2. All data share same memory
3. Union stores methods too
4. Union keyword is used to initialize
5. The size of a union is predefined by the compiler

E) In which case union is better than structure?

1.Less memory is available
2.Faster compilation is required
3.When functions are included
4.None of these

F) What would be the output of the following C code?

```
#include <stdio.h>
int main()
{
    char str1[] = { 'H', 'e', 'l', 'l', 'o' };
    char str2[] = "Hello";
    printf("%ld,%ld", sizeof(str1), sizeof(str2));
    return 0;
}
```

G) What will be the output of the size of unionJob and structJob?

```
#include <stdio.h>
union unionJob
{
  //defining a union
  char name[32];
  float salary;
  int workerNo;
} uJob;

struct structJob
{
  char name[32];
  float salary;
  int workerNo;
} sJob;
int main()
{
  printf("size of union = %d bytes", sizeof(uJob));
  printf("\nsize of structure = %d bytes", sizeof(sJob));
  return 0;
}
```

Other important things

1. Grouping
2. RPi set and components (e.g. 3/4G USB, sensors)

Other important things

1. 1 x (Pi5+ Pi5 Power adaptor+ Pi5 debugging module+ 64 GB memory card for Pi 5)
2. Pi Camera Set x 1
3. 4GB USB x 1
4. Arduino UNO & communication chip (NRF24L01+)
5. 8x8 matrix LED & 5 pin wire
6. Sensors
   - Ultrasound (HC-SR04)
   - MQ2 sensor
   - Temperature & Humidity & 3Pin wires
   - Near-Infrared for motion detection
7. USB ethernet card and cables
8. Breadboard x 1
9. Cables/ Wires x2

END