

EMBEDDED SOFTWARE AND SYSTEMS

ESS

Andrew Markham

v0.1.73

Professional Programme
University of Oxford

1. OVERVIEW

INTRODUCTION

- Embedded systems are everywhere
- They are invisible until they fail!
- This course introduces good design principles

AIMS AND OBJECTIVES

- Understand characteristics and properties of Embedded Systems
- Apply software engineering principles
- Design a system within constraints
- Run and debug code on hardware

PLAN FOR THE WEEK

- Fundamentals of Embedded Systems
- Software Engineering for Embedded Systems
- Reinforced with class exercises and hands-on labs

PLAN FOR THE WEEK

- Mornings: Lectures and Exercises
- Afternoons: Labs

Have you ever:

- programmed in C?
- programmed in assembly?
- programmed a microcontroller?
- used a logic analyzer?
- used a soldering iron?

RELATED COURSES

- SCS: Safety Critical Systems
- TOI: Things of the Internet

2. INTRODUCTION

SCOPE

What is an embedded system?

DEFINITION

An embedded system is a computer system with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints.

(Wikipedia)

DEFINITION

Embedded software is software integrated with physical processes. The technical problem is managing time and concurrency in computational systems.
(Edward Lee, Berkeley)

DEFINITION

Information processing systems embedded into a larger product.
(Mardewel)

OUR DEFINITION

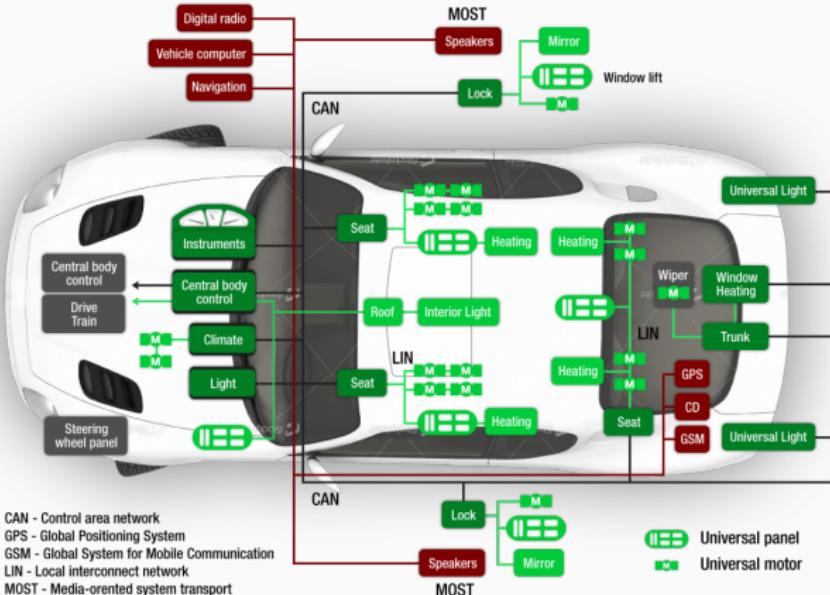
Broadly, an embedded system is:

- application or task specific
- dimensioned for the problem
- subject to constraints
- interacts with external physical system

EMBEDDED EVERYWHERE

“Embedded chips form the backbone of the electronics driven world in which we live ... they are part of almost everything that runs on electricity”
(Mary Ryan, EEDesign, 1995)

AUTOMOTIVE



<http://www.redsalt.com/service/automotive-systems>

CONSUMER



<http://www.appliance design.com/>

TELECOMMUNICATION



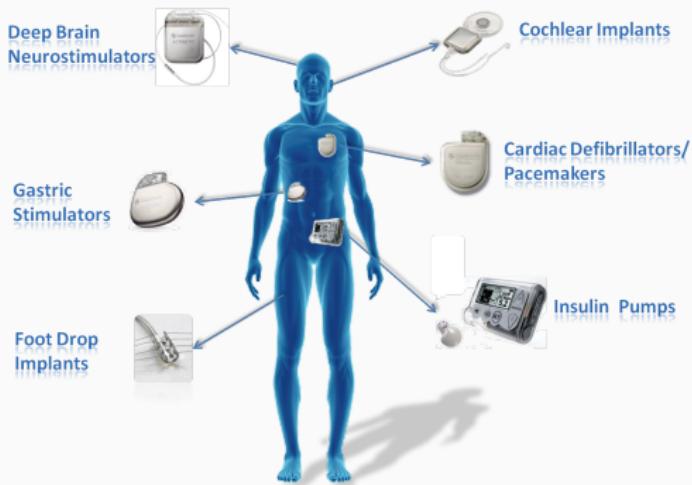
<http://www.apc-hero.co.uk/>

AVIONICS



<http://www.wonderfulengineering.com/>

WIRELESS IMPLANTABLE MEDICAL DEVICES



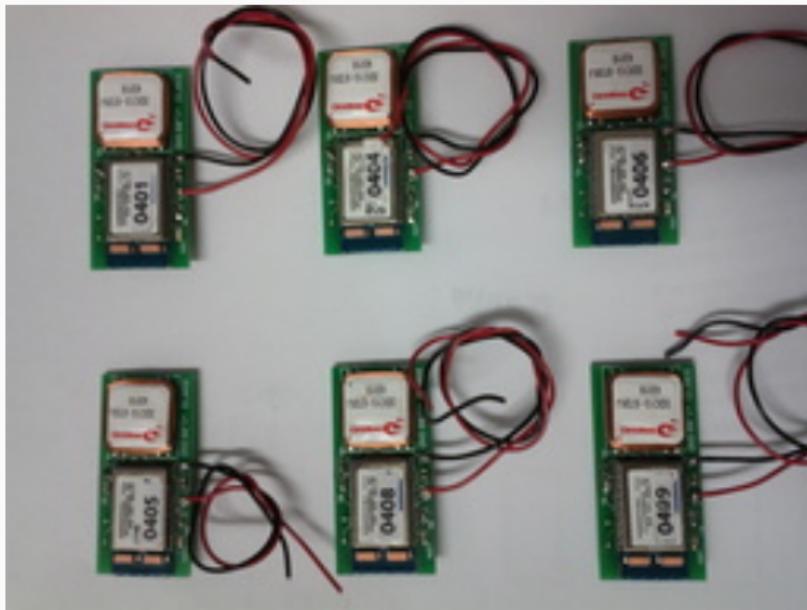
<http://www.groups.csail.mit.edu/>

ROBOTICS



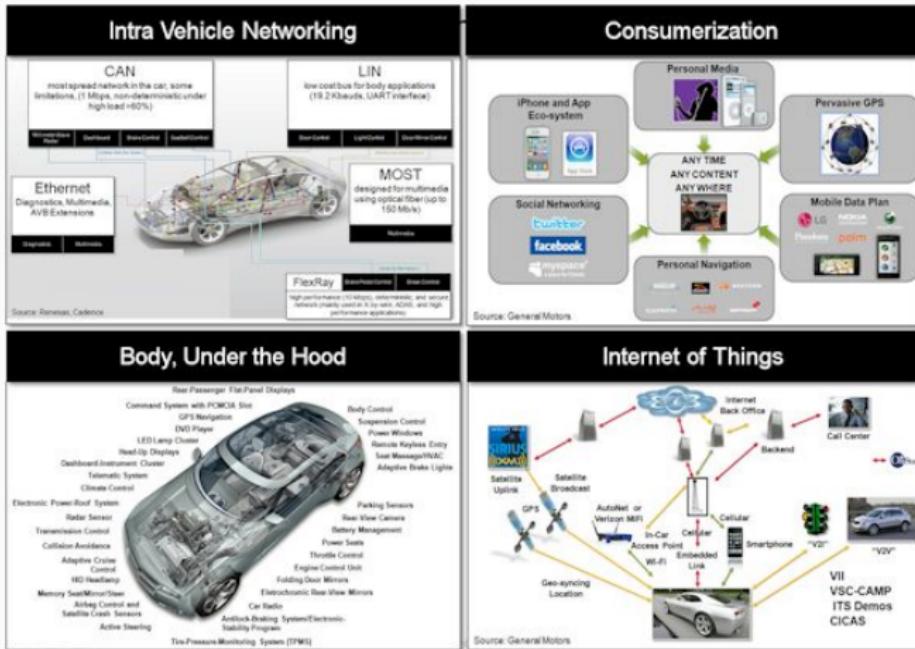
kuka.com

WIRELESS SENSORS (IOT)



amarkham.com

CYBER PHYSICAL SYSTEMS (CPS)



<http://community.cadence.com/>

CHARACTERISTICS

Embedded systems must be:

- Autonomous: Continuous operation without intervention
- Reliable: Provide a continuous level of service
- Safe: A failure must not cause harm
- Reactive: Respond with a known time
- Secure: Communication should be authentic and private

These are challenging goals to achieve, as the operating conditions can vary radically

We have to design for worst case scenarios

CONSTRAINTS

Embedded systems are often subject to severe and contradictory constraints

- Energy: many are battery powered
- Code size: e.g. 64 kbyte Flash
- Memory size: e.g. 4 kbyte RAM
- Performance: need to meet real-time constraints
- Weight/Size: key for portable devices
- **Cost**: especially for high-volume, this is often the dominant factor

Embedded systems by definition are integrated into a physical environment and must adhere to real-time constraints

- Real-time systems must react to external stimuli and control input
- Any output too late (or too early), even if correct, is wrong and could be harmful

Hard real-time

A **hard real-time** constraint is one where not meeting the constraint could result in catastrophe (death/injury/explosion)

Soft real-time

A soft real-time constraint is one where not meeting the constraint leads to degradation of service

COMPLEXITY

- Pacemaker: 100kLOC
- Apollo 11: 140 kLOC
- Spaceshuttle: 400kLOC
- F22 Raptor: 2 MLOC
- A320 Avionics: 10 MLOC
- High end car: 100 MLOC

COST OF FIRMWARE

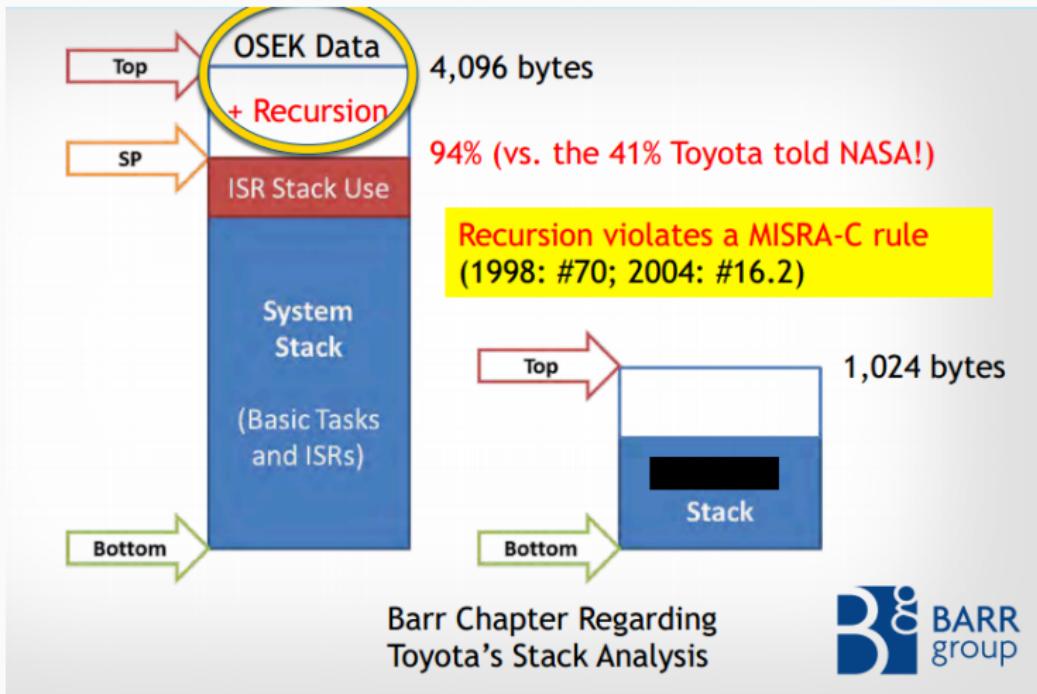
- Typically put at between \$10 and \$100 per LOC
- For complex systems like planes, firmware costs between 30% and 50% of system cost
- Cost of maintenance and failure can be significantly higher!

IMPORTANCE OF SOFTWARE ENGINEERING PRINCIPLES

- Toyota faced a major lawsuit in USA for ‘unintended acceleration’
- The accelerator control stuck open causing fatalities and a number of crashes
- Throttle is controlled by the ECU
- Toyota did not follow basic software engineering principles

- 11,000 global variables!
- Cyclomatic complexity of over 100 for some functions (unmaintainable)
- Critical: stack overflow killed the monitoring task that would reset the OS
- Critical: used recursive calls in an embedded system

STACK OVERFLOW



<http://www.safetyresearch.net/>

COST

- 89 fatalities and 57 injuries
- \$1.2B in damages
- Recall across fleet
- Reputation damage

SYSTEMS OF SYSTEMS



<https://aurora.tech/>

HUMAN OR MACHINE ERROR?



Figure 2. View of the self-driving system data playback at about 1.3 seconds before impact, when the system determined an emergency braking maneuver would be needed to mitigate a collision. Yellow bands are shown in meters ahead. Orange lines show the center of mapped travel lanes. The purple shaded area shows the path the vehicle traveled, with the green line showing the center of that path.

SUMMARY

- Embedded Systems integrate into the physical world to provide sensing, communication and actuation
- Most silicon ends up in embedded devices¹
- The IoT is going to lead to an exponential increase in device proliferation
- Challenges of embedded systems are not going away
- Strong need for good design principles

¹ 25 Billion units shipped in 2020

FURTHER READING

- ‘Making Embedded Systems: Design Patterns for Great Software’ - White
- www.embedded.com

3. ARCHITECTURE AND THE ALU

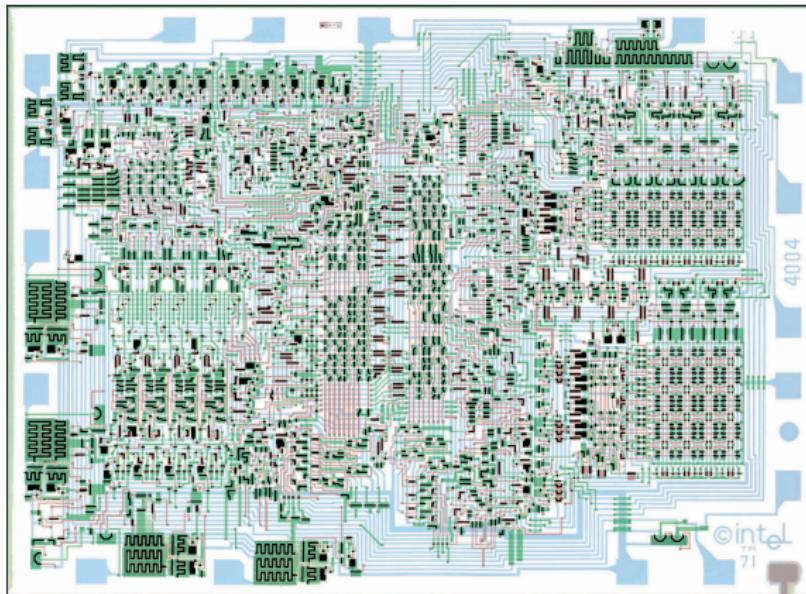
OVERVIEW

- How did microcontrollers evolve?
- How does a microcontroller work?
- How do we make it execute instructions?

HISTORY

- Intel 4004 was the world's first microprocessor, taped out in 1971
- 4 bit CPU, BCD oriented
- 46 instructions
- 16 registers
- 2300 transistors, 10 micron process
- Separate memory and control chips (4001,4002,4003)

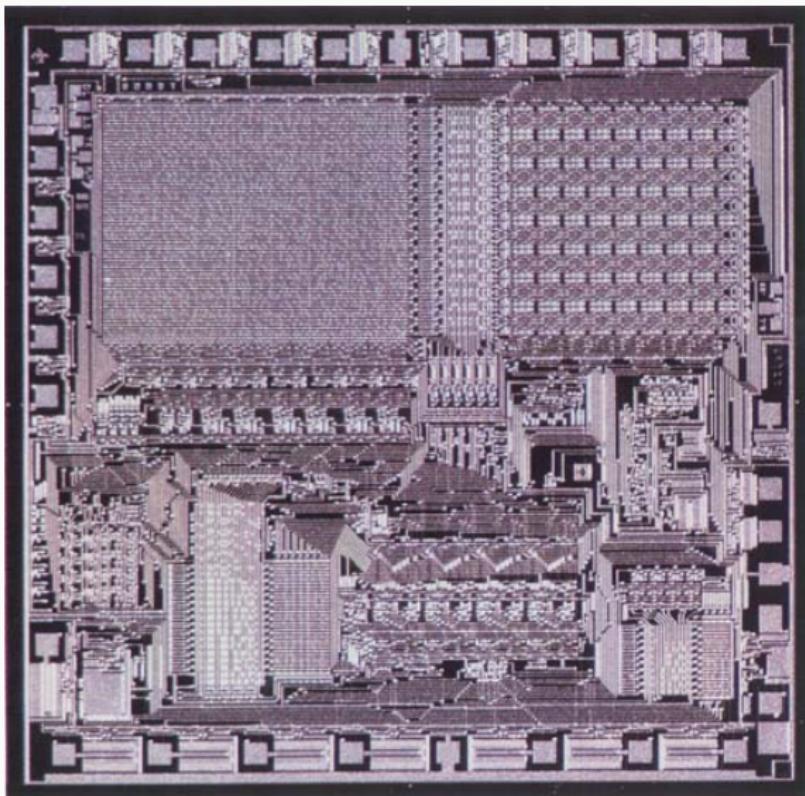
INTEL 4004



TMS 1000

- TI developed TMS1000, first microcontroller
- Integrated memory and CPU on single chip
- 4 bit CPU
- 3kbits ROM, 128 bits RAM
- Widely used in calculators

TMS 1000



INTEL 8051

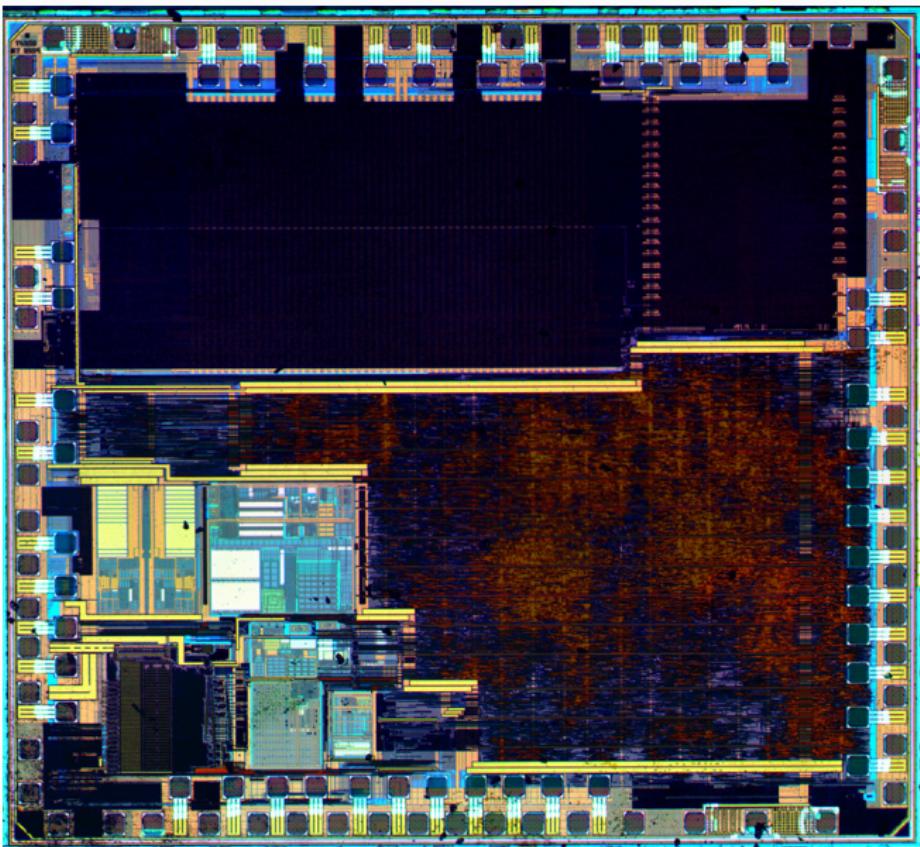
- One of the most popular microcontrollers of all time
- Introduced in 1980
- 8 bit CPU
- 128 bytes RAM, 4kbyte ROM
- Timers and USART

INTEL 8051

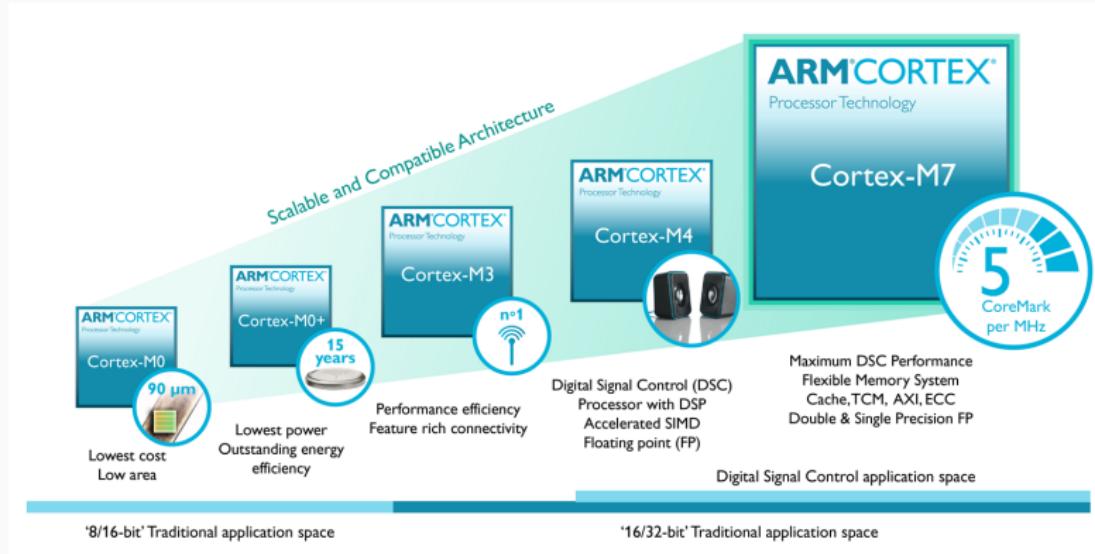
- Still in use and being developed today
- Available as silicon IP core
- Embedded in System-On-Chips such as wireless transceivers

- Major impact in the microcontroller world
- Fabless - supply IP blocks
- 32 bit Cortex-M processors account for a large proportion of embedded market
- More and more manufacturers integrating ARM cores
- Feature easy migration between different cores
- R series for SCS
- M0+: 12k gates
- M4: 90k gates (FPU)

CORTEX M3



ARM: CORTEX-M



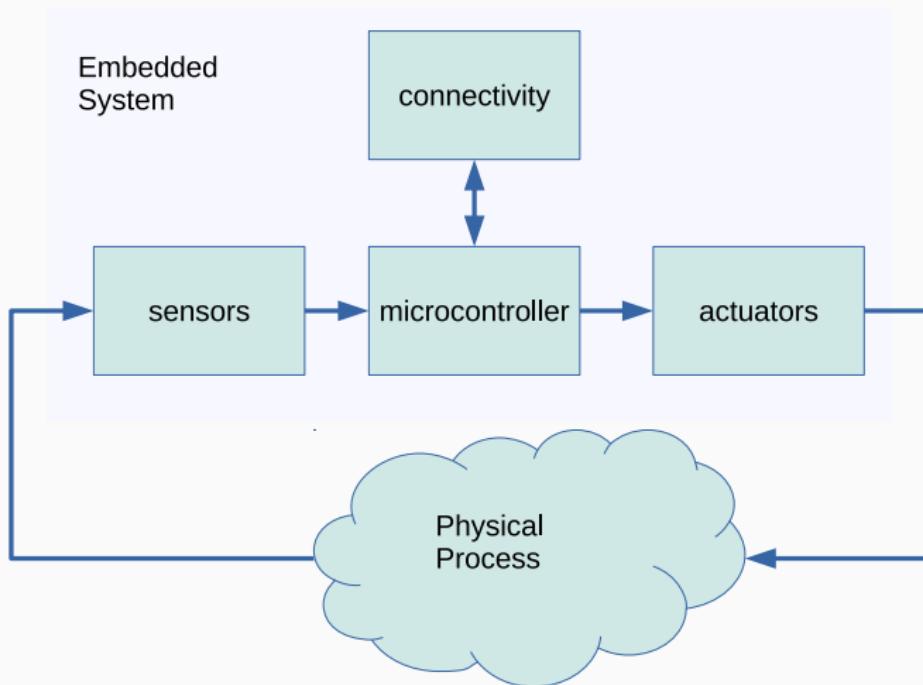
DISTRIBUTION OF MICROCONTROLLERS

- Value and volume¹



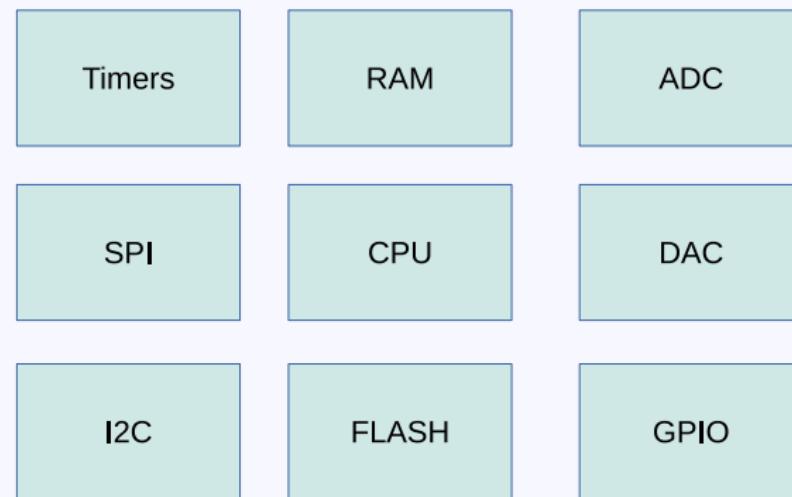
¹<http://www.grandviewresearch.com/>

BLOCK DIAGRAM OF AN EMBEDDED SYSTEM



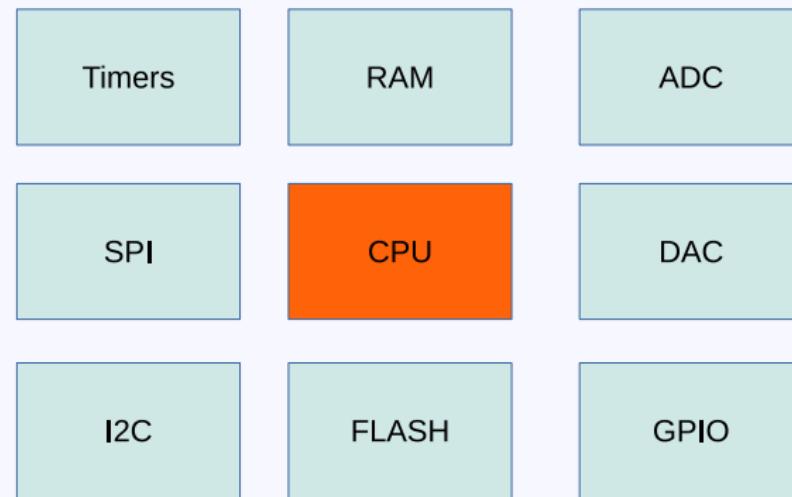
BLOCK DIAGRAM OF A MICROCONTROLLER

Microcontroller

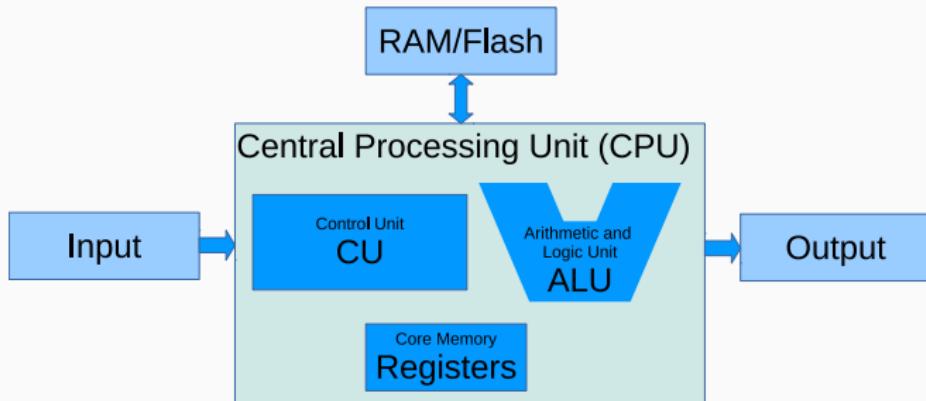


HEART OF A MICROCONTROLLER

Microcontroller



CPU BLOCK DIAGRAM



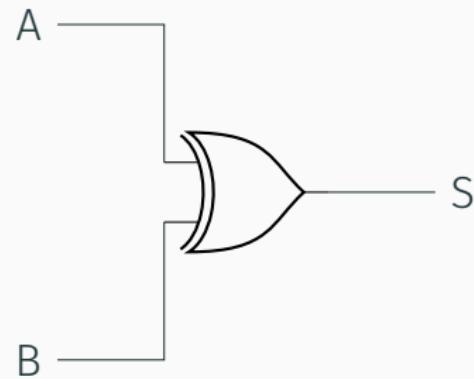
ALU: ARITHMETIC AND LOGIC UNIT

- Heart of the CPU
- Performs basic operations like adding two numbers
- Has special **flags** that are used to control logic flow

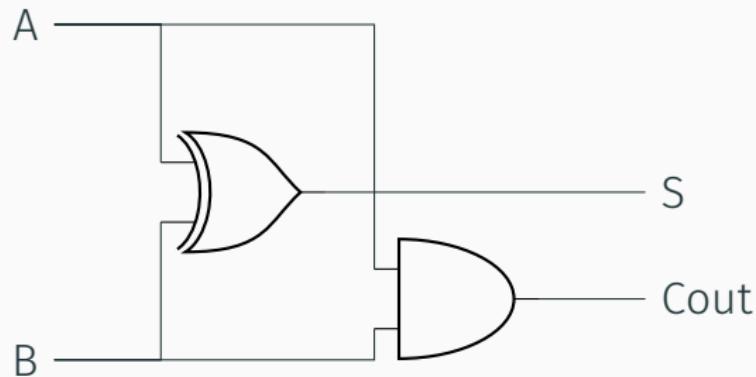
BUILDING BLOCKS OF COMPUTATION: GATES

- Digital gates (AND, OR, NAND) are the building blocks of logic and computation
- These are combined to build up more advanced functions, such as adders and multipliers
- The key is to minimize the number of gates required - less die space, less power consumption

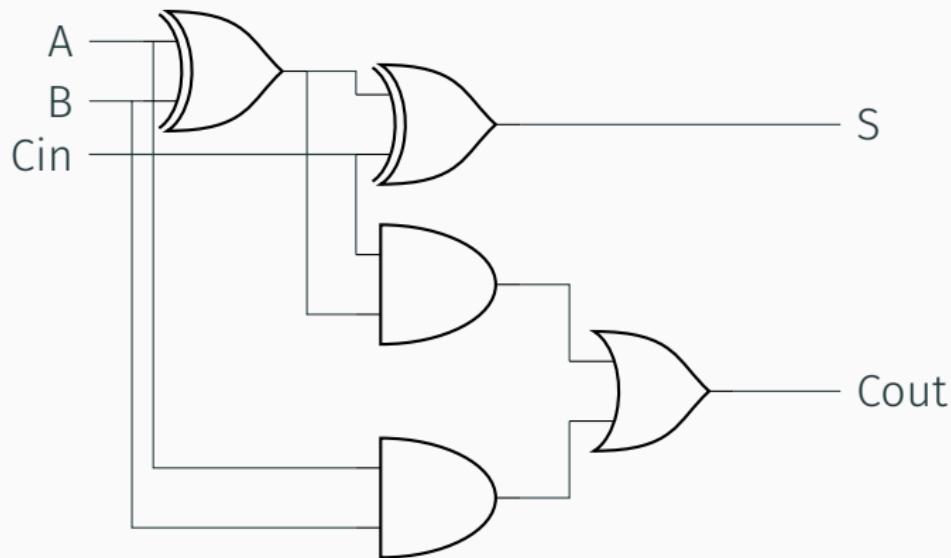
A 1-BIT ADDER (NO CARRY)



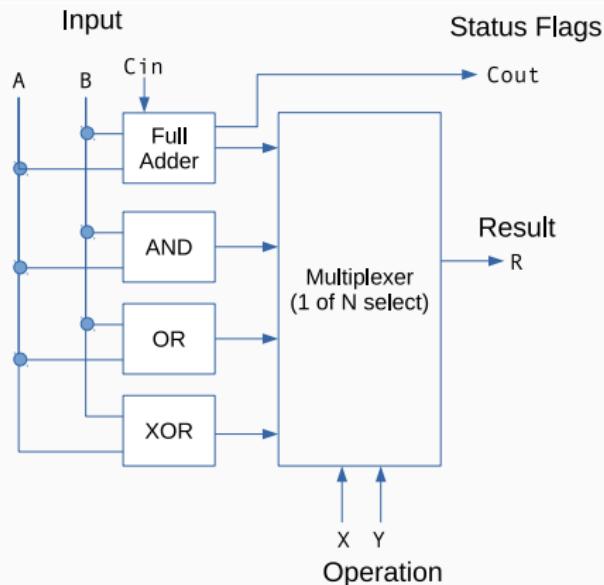
A 1-BIT ADDER WITH CARRY-OUT (HALF-ADDER)



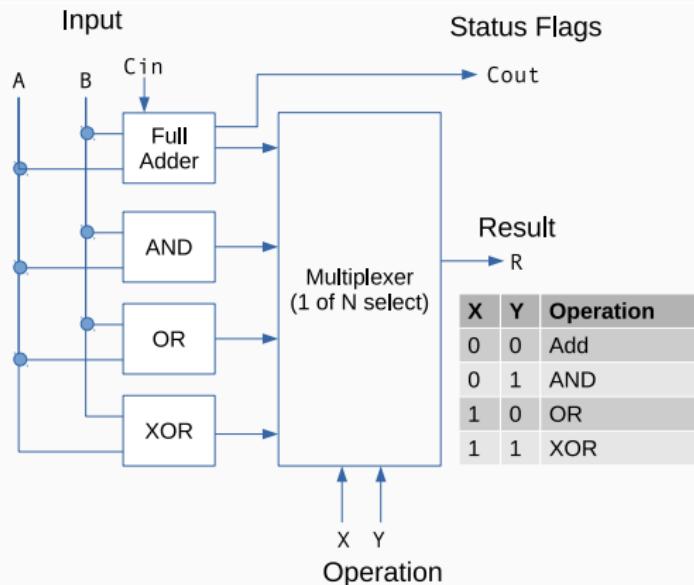
A 1-BIT FULL ADDER



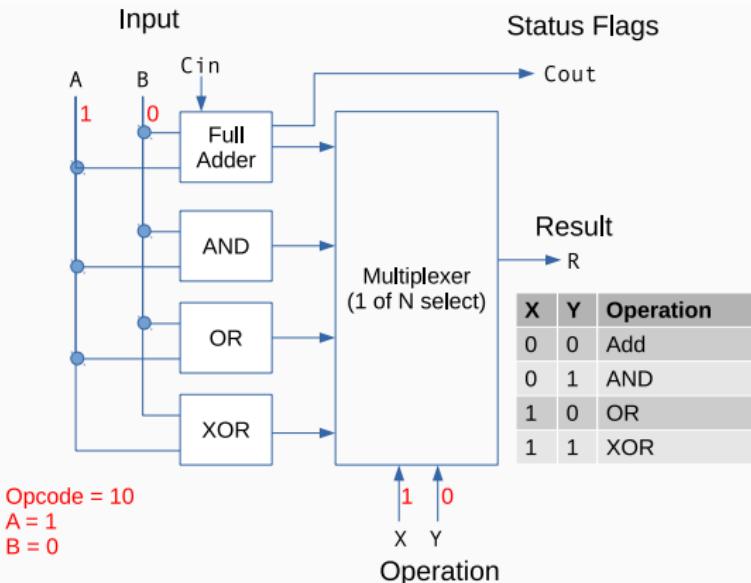
1-BIT ALU SLICE



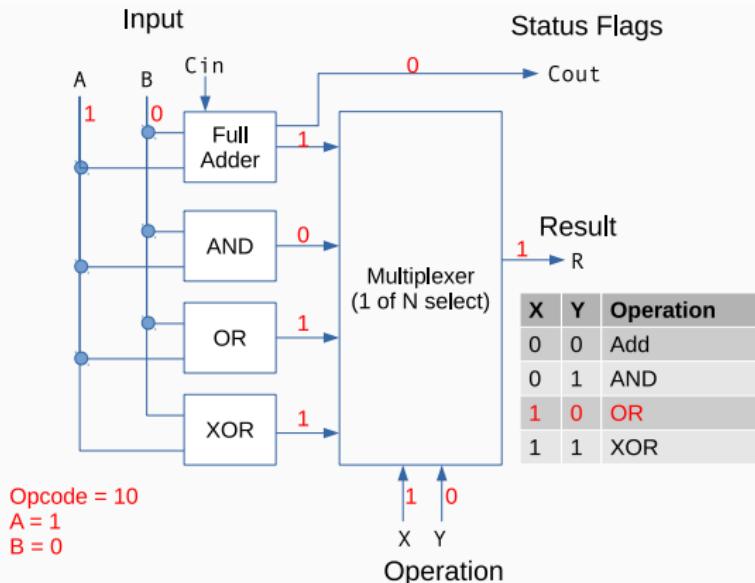
1-BIT ALU SLICE



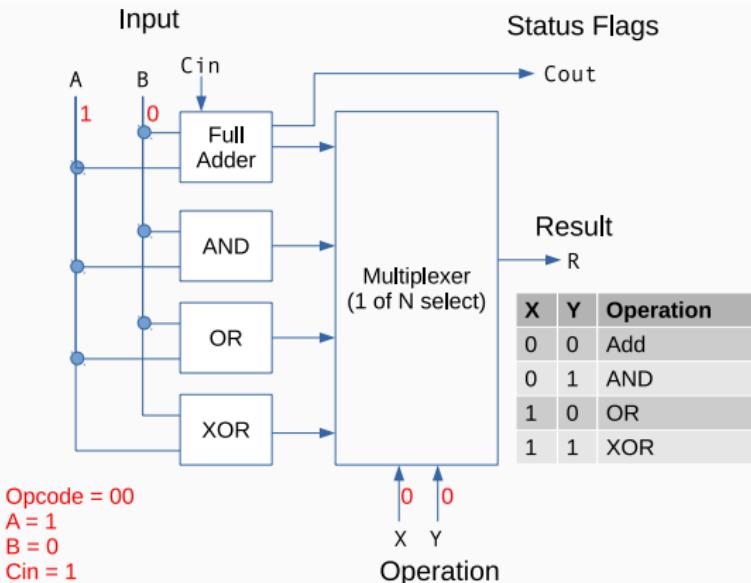
1-BIT ALU SLICE



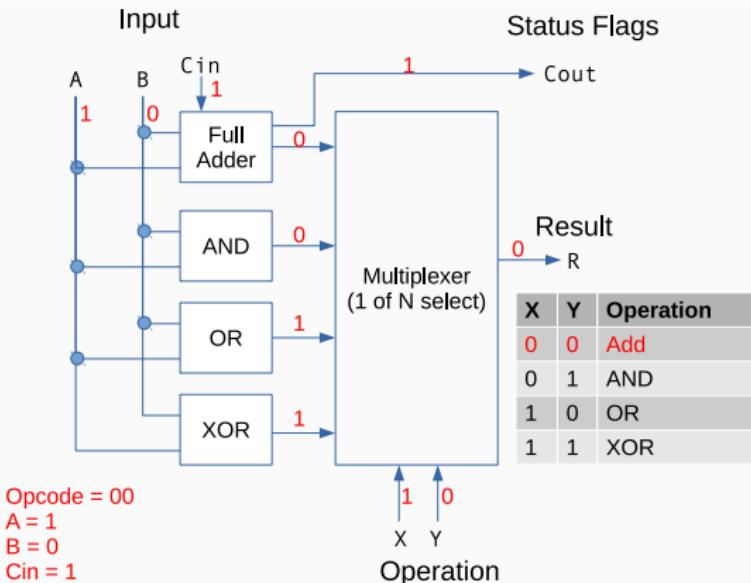
1-BIT ALU SLICE



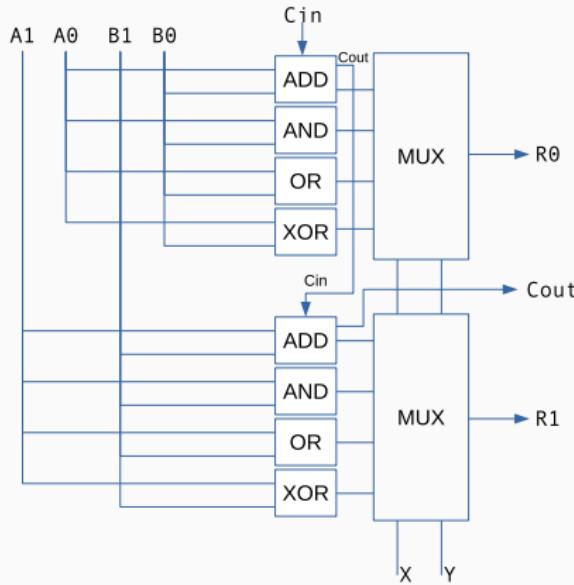
1-BIT ALU SLICE



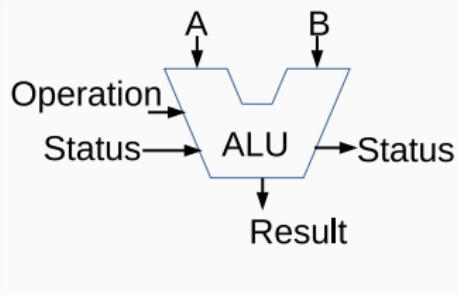
1-BIT ALU SLICE



2-BIT ALU



GENERAL ALU: HEART OF THE CPU



ACCUMULATOR MODEL

- In an accumulator style CPU there is a single register (also called the working register or accumulator)
- This is a 1-operand machine
- This holds the result of the last computation
- This is like a calculator which **accumulates** the total to date
- The 8051 and PIC16 families are accumulator based
- e.g. ADDLW 1: $A \leftarrow A + 1$

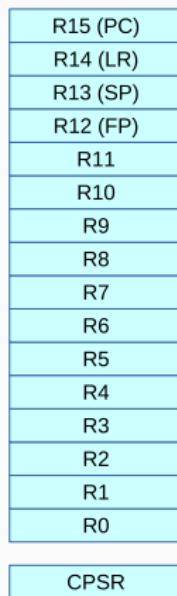
REGISTER MODEL

- In a register model, there are multiple local registers (e.g. 8 or 16) that hold results of computation
- An instruction typically has two source registers and one destination register
- Hence these are 2 or 3 operand machines
- ARM and MSP families are register based
- e.g. ADD R1, R2, R3: $R3 \leftarrow R1 + R2$

COMPARISON BETWEEN ALU MODELS

- Accumulator style ALU's are simple, however they are inefficient for high level programming languages
- This is because data in the accumulator has to be transferred back and forth to storage to implement simple structures like for-loops
- The register model is better suited to languages like C and typically leads to more efficient code, at the cost of a more complex ALU

REGISTERS (ARM M-CORTEX MODEL)



Registers

SPECIAL REGISTERS

- PC: Program Counter - address in memory to fetch next instruction from
- LR: Link Register - used for subroutines to store return address
- SP: Stack Pointer - offset into RAM for temporary allocation of memory for functions
- FP: Frame Pointer - used for subroutines to store prior stack pointer
- CPSR: Current Program State Register - contains flags

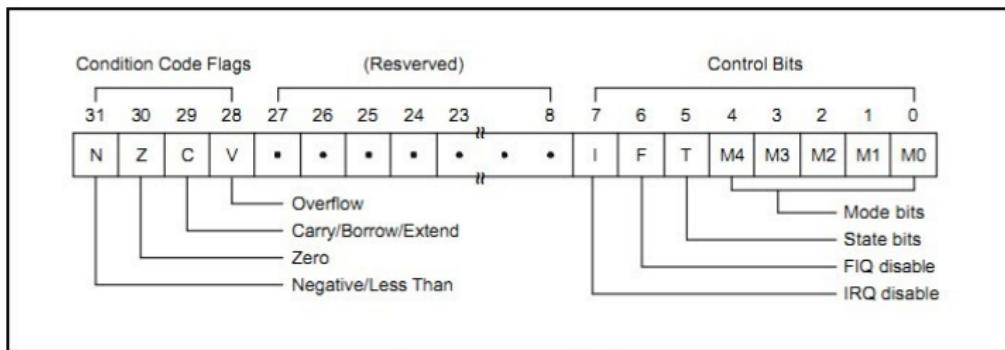
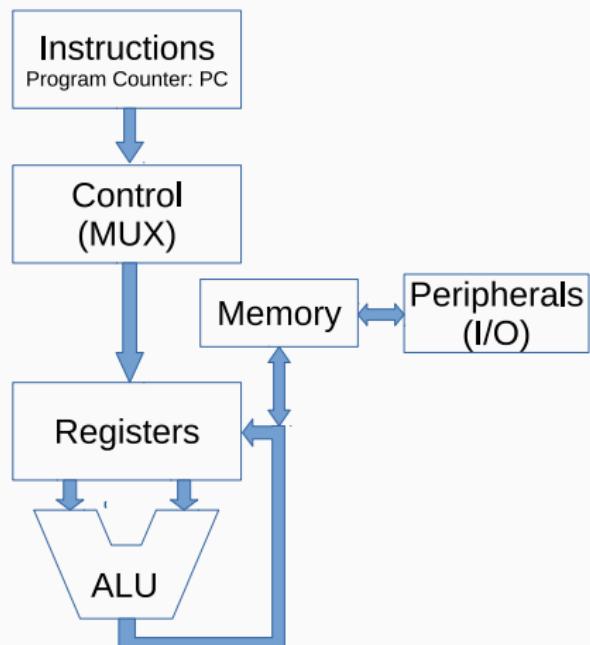
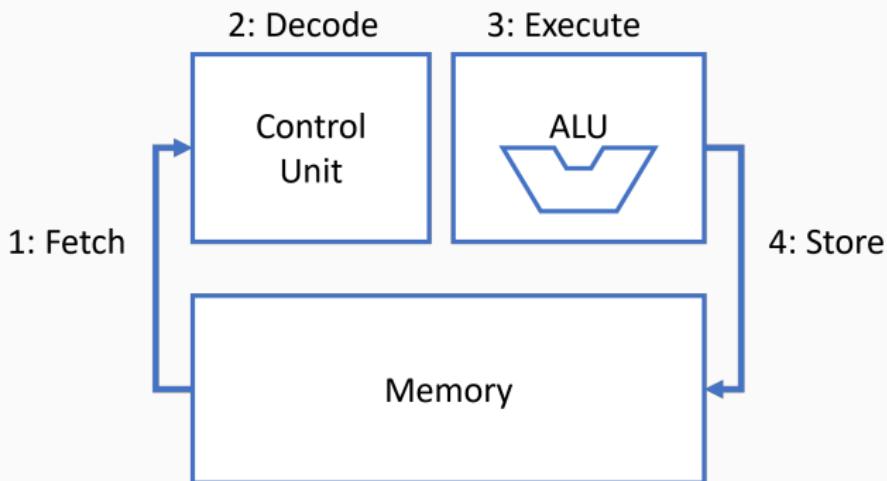


Figure 2-6. Program Status Register Format

SIMPLIFIED CPU: INTERFACING WITH MEMORY



SIMPLIFIED CPU: CYCLES OF OPERATION



INSTRUCTION SETS

- The operations that are supported by a particular CPU make up the **instruction set**
- A larger instruction set (e.g. support for hardware multiply):
 - Gives better performance
 - Costs more in terms of complexity, die space and power consumption

RISC: REDUCED INSTRUCTION SET CONTROLLER

- RISC processors have a very small instruction set
- Most instructions are single cycle
- Most microcontroller cores are RISC (ARM, AVR, PIC)

RISC EXAMPLE: MICROCHIP PIC16F SERIES

- 35 instructions in total
- Two arithmetic operations (add and sub)
- 12 bit wide op-code (6 bits for instruction, 8 bits for memory address)
- Accumulator based architecture (single register)

Section 29. Instruction Set

Table 29-1: Midrange Instruction Set

Mnemonic, Operands	Description	Cycles	14-Bit Instruction Word				Status Affected	Notes
			MSb		Lsb			
BYTE-ORIENTED FILE REGISTER OPERATIONS								
ADDWF	f, d	Add W and f	1	00	0111	ffff	ffff	C,DC,Z
ANDWF	f, d	AND W with f	1	00	0101	ffff	ffff	Z
CLRF	f	Clear f	1	00	0001	ffff	ffff	Z
CLRW	-	Clear W	1	00	0001	0xxx	xxxx	Z
COMF	f, d	Complement f	1	00	1001	ffff	ffff	Z
DECf	f, d	Decrement f	1	00	0011	ffff	ffff	Z
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00	1011	ffff	ffff	1,2,3
INCF	f, d	Increment f	1	00	1010	ffff	ffff	Z
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00	1111	ffff	ffff	1,2,3
IORWF	f, d	Inclusive OR W with f	1	00	0100	ffff	ffff	Z
MOVF	f, d	Move f	1	00	1000	ffff	ffff	Z
MOVWF	f	Move W to f	1	00	0000	1fff	ffff	
NOP	-	No Operation	1	00	0000	0xx0	0000	
RLF	f, d	Rotate Left f through Carry	1	00	1101	ffff	ffff	C
RRF	f, d	Rotate Right f through Carry	1	00	1100	ffff	ffff	C
SUBWF	f, d	Subtract W from f	1	00	0010	ffff	ffff	C,DC,Z
SWAPF	f, d	Swap nibbles in f	1	00	1110	ffff	ffff	1,2
XORWF	f, d	Exclusive OR W with f	1	00	0110	ffff	ffff	Z
BIT-ORIENTED FILE REGISTER OPERATIONS								
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff	
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff	
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff	3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff	3
LITERAL AND CONTROL OPERATIONS								
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z
CALL	k	Call subroutine	2	10	0kkk	kkkk	kkkk	
CLRWD	-	Clear Watchdog Timer	1	00	0000	0110	0100	T0,PD
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk	
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z
MOVlw	k	Move literal to W	1	11	00xx	kkkk	kkkk	
RETFIE	-	Return from interrupt	2	00	0000	0000	1001	
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk	
RETURN	-	Return from Subroutine	2	00	0000	0000	1000	
SLEEP	-	Go into standby mode	1	00	0000	0110	0011	T0,PD
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z

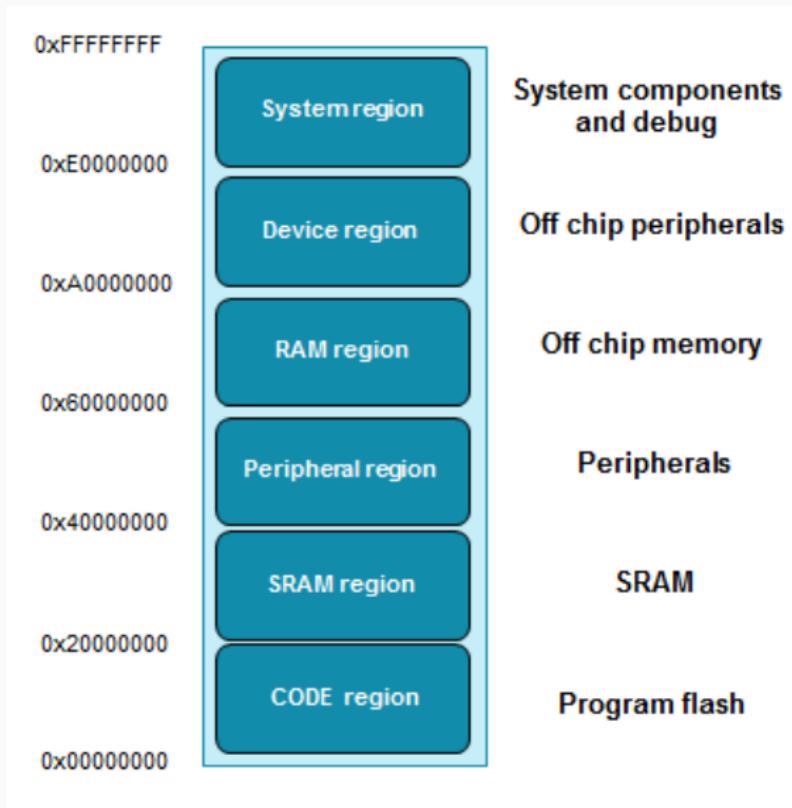
Note 1: When an I/O register is modified as a function of itself (e.g., MOVF PORTB, 1), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMRO register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 Module.

CISC: COMPLETE INSTRUCTION SET CONTROLLER

- CISC processors have a much larger instruction set
- Instructions take a variable number of cycles (2 to 10)
- Internally, CISC instructions are often translated to RISC-style micro-instructions
- e.g. Intel Quark runs a pentium instruction set
- Note: Definition of CISC vs RISC is pretty fuzzy

MEMORY MAPS



MACHINE CODE

MACHINE CODE

84C0: BE 1B 7D 51 BE 9D 51 BE F7	8778: BD AD 82 BD BD 72 BD AD E1	8A30: 02 E6 1F AE 75 BD DE 93 02
84C8: C9 0A 90 12 E9 0A 9D 51 70	8780: B1 BD BD 73 BD A9 2B 85 0B	8A38: BD D0 BB 60 A0 00 A9 00 1E
84D0: BE CA 30 0A FE 51 BE BD 92	8788: FC A9 8C 85 FD A9 00 BD AB	8A40: 4B B1 FC A9 68 1D 00 61 49
84D8: 51 BE C9 0A 00 EE 20 E3 75	8790: 6E BD A9 02 BD 6D BD A9 76	8A48: 4D 70 BD 51 FE 91 FE BD E0
84E0: 87 AD 4D BE 09 00 BD 4D 5E	8798: 00 BD 70 BD AD 7E BD 29 9D	8A50: 00 69 C8 CC 6F ED D8 E8 E1
84E8: BE BD 4D BE 20 4A 05 A9 0E	87A0: 7F F0 47 C9 02 90 17 D0 B2	8A58: 4D 70 BD 51 FE 91 FE 56 93
84F0: 16 85 FC A9 BC B5 FD A9 C0	87AB: 10 AC 73 BD CB C0 07 D0 5A	8A60: AD 71 BD 29 3F AB B9 89 B6
84F8: 00 BD 6D 00 A9 15 BD 6E 17	87B0: 05 A0 00 EE 72 BD BC 73 AF	8A68: 8A 0D 74 BD 85 FF AD 71 66
8500: BD AD 76 BD BD 72 BD AD E3	87B8: BD A9 00 BD 78 BD 2C 7E 72	8A70: BD 29 00 F0 02 A9 88 1B 76
8508: 77 BD BD 73 BD A9 9C BD F4	87C0: BD 10 10 A9 52 85 FC A9 83	8A78: 2C 71 BD 70 04 10 04 69 8A
8510: 71 BD A9 00 BD 70 BD 2D D5	87C8: BD 6E FD CE 71 BD A9 0F 50	8A80: 2B 69 2B 6D 72 BD 85 FE B3
8518: F7 89 2B 53 00 2B 00 7D	87D0: BD 6E BD 2C 83 BD 10 03 2C	8A88: 60 00 04 08 0C 10 14 18 AF
8520: A2 00 A0 00 CA D0 FD BB AE	87D8: 4C 7F 89 20 BF 89 AD 74 15	8A90: IC 00 04 08 0C 10 14 18 95
8528: D0 FA A2 00 BD 4E BE 9D 98	87E0: BD 49 60 00 74 BD C5 E6 JA	8A98: IC 01 05 09 0D 11 15 19 1D
8530: 4D BE EB EC 87 BD D0 F4 7A	87EB: F0 89 60 A9 00 BD 8F 8D 5C	8A9E: 1D 01 05 09 0D 11 15 19 25
8538: CE 87 BD 20 62 89 20 EE AB	87F0: AE 8F BD BD 51 BE 0A BD 2F	8AA0: 1D 02 06 0A 0E 12 16 1A 2D
8540: 89 20 62 89 28 CA 86 4C 82	87FB: 7B BD BA 10 6D 78 BD 69 3C	8A80: 1E 02 06 0A 0E 12 16 1A B5
8548: 92 84 A9 92 80 89 BD A2 6C	8800: 7F 85 FC A9 8C 05 FD 90 73	8A88: 1E 03 07 0B 0F 13 17 1B 3D
8550: 00 BE BD 00 AB 88 BD EC 88	8808: 02 E6 FD A9 00 BD 6D BD CC	8AC0: 1F 03 07 0B 0F 13 17 1C 55
8558: 87 BD F0 2F BD 4D BE BD 69	8810: BD 71 BD 80 BD 70 BD 73 17	8AC8: 1F B0 90 70 60 50 40 3A 7
8560: 7E BD AD 76 BD BD 72 BD 40	8818: BD A9 00 BD 6E BD E0 05 64	8A90: 10 01 FF F7 D8 C6 84 A2 98
8568: AD 77 BD BD 73 BD AD 09 69	8820: D0 A4 09 0E D0 04 BA 1B 74	8AD0: 98 00 60 A9 FF BD 19 BB F0
8570: 8B BD 71 BD A9 88 BD 03 9A	8828: 69 07 BD 72 BD 2B 8F 89 7E	8AE0: 4C FB BA A9 78 BD 19 BB 8E
8578: BD 20 05 87 AD 89 BD 3A 62	8830: AD 74 BD 49 60 BD 74 BD 2B	8AF0: 4C FB BA A9 71 BD 19 BB 8E
8580: E9 0E BD 89 BD EE BB BD 15	8838: C5 E6 F0 84 AE 8F BD E8 07	8AF8: 4C FB BA A9 72 BD 19 BB 9E
8588: 4C 54 85 60 AD 87 BD D8 FC	8840: E6 00 F0 86 BE 8F BD 4C DB	8B00: AD 00 A9 E0 BD 2C BB A9 88
8590: 01 60 A9 00 BD 88 BD A9 49	8848: F0 87 60 A9 02 BD 92 BD 53	8B08: FF BD 36 BB AD 36 BB 8D EC
8598: 06 BD BA BD AD 8B BD CD 88	8850: A9 14 BD 91 BD A9 00 BD A6	8B10: 55 BB 4E 53 BB 98 0C B9 24
85A0: 76 BD 90 0A D0 26 AD 8A 02	8858: 70 8D AD 90 00 0A 0A 0A 76	8B18: 00 70 C8 BD 54 BB A9 80 E1
85A8: BD CD 77 BD 00 1E AD 9A 99	8860: AD 60 92 BD C9 07 90 07 BF	8B20: BD 53 BB 4E 54 88 90 03 1E
85B0: 6D 69 06 C9 07 90 05 E9 AB	8868: EE 91 BD E0 07 07 F5 BD 1A	8B28: AD 30 C0 A2 FF EB D0 FD A7
85B8: 07 EE BB BD 00 BD AD AC	8870: 92 BD A9 07 BD 6E BD A9 BE	8B30: 90 03 AD 30 C0 A2 FF EB 82
85C0: BB BD 18 69 06 00 BB BD 99	8878: 02 BD 60 BD AD 91 BD 00	8B38: D0 FD EE 55 88 BB 00 D3 18 C9
85C8: C9 24 D0 D0 AD 00 BD 3B D3	8880: 72 BD AD 92 BD BD 73 BD 24	8B40: AD 36 BB E9 01 BD 36 BB 02
85D0: E9 03 00 05 69 07 CE BB 88	8888: A9 00 BD 71 BD A9 56 85 7C	8B48: AD 2C BB 69 01 BD 2C BB 6B
85D8: BD BD BA BD AD BB BD 2B 27	8890: FC A9 BB 85 FD 2B 0F BD 09 CD	8B50: 98 00 BA 60 20 2E 20 00 BE 1E
85E0: E9 03 BD BB BD 3B ED 76 AB	8898: AD 74 BD 49 60 BD 74 BD 93	8B58: 00 00 FF 81 00 AF 00 00 2C
85E8: BD BD 78 BD 00 BA 0A 3B CA	88A0: C5 E6 F0 CE 6A 54 C0 7C	8B60: E7 BD 00 FB BB 80 BD 80 50
85F0: ED 7B BD 18 6D BD ED E2	88AB: AD 51 C0 28 58 FC 60 20 96	8B68: 00 00 00 A0 05 00 BB D1 71
85F8: 77 BD BD 8C BD 10 05 49 9D	88B0: A5 B8 A9 00 88 85 25 20 22 BF	8B70: B1 9C E4 B1 BE D1 B3 B6 6D
8600: FF 3C 69 00 1B 69 51 BD DE	88B8: FC A9 00 85 24 A2 00 BD J5	8B78: C4 B3 B6 91 B7 87 B6 84 7E
8608: 00 8B 00 90 8D 00 BD A9 E9	88C0: EC 88 2B ED FD EB 00 1B BD	8B80: B6 91 BE CC AA 8C D0 AA B5
8610: 00 BD 8B BD AD BE BD BA	88C8: D0 F5 AD 10 C0 AD 00 C0 F3	8B88: 98 C0 AA 80 C0 A2 B1 C0 CD
8618: 00 BD AE BB EC BD 00 BD 6B	88D0: C9 D0 F0 12 C9 F0 BE 0E 3C	8B90: B2 85 C0 B2 85 D0 A0 B1 B1 BC
8620: D0 03 4C AA BD 4D BE 00 BD	88D8: C9 CB F0 04 C9 EB ED AD	8B98: D0 A0 B1 D0 A0 B1 F0 E3 4E
8628: 00 7E BD AD 76 BD BD 72 9F	88E0: A9 00 BD 6B BD 60 A9 FF 70	8BA0: B7 FC FF 97 F0 F0 87 F0 JE
8630: BD AD 77 BD BD 73 BD AD JA	88E8: 00 6B 60 C0 CE DE 05 54	8RA0: A7 C4 A0 A0 F0 9F R0 7A
8638: 89 BD C9 52 98 35 AD 00 0E	88F0: D0 A0 A0 D0 A9 C1 C4 2F	

INTRODUCTION TO ASSEMBLER

- Assembler is a simple language that means we don't have to write in hexadecimal op-codes
- Instructions are given by mnemonics (e.g. ADD) that make them easier to remember
- Nowadays, assembly is generally only used for speed-critical routines
- However, it is useful to know the basics so we can understand how compilers work

SIMPLE EXAMPLE: ADDITION

- Goal: Add 2 numbers $Z \leftarrow P + Q$

SIMPLE EXAMPLE: ADDITION

```
start:          @ Label
    mov    r0, #5      @ Load register r0 with the value 5 (P)
    mov    r1, #4      @ Load register r1 with the value 4 (Q)
    add    r2, r1, r0   @ Add r0 and r1 and store in r2 (Z)

stop:    b stop        @ Branch to label "stop"
```

SIMPLE EXAMPLE: CONDITIONAL BRANCHING

- Implement the following C code:

```
a = 0;  
for (i=0; i <10; i++)  
{  
    a = a+1;  
}
```

SIMPLE EXAMPLE: CONDITIONAL BRANCHING

```
start: mov r3, #0          @ a = 0
      mov r0, #0          @ set loop counter to 0
loop:  add r3, r3, #1      @ add 1 to a
      add r0, r0, #1      @ increment loop counter by 1
      cmp r0, #10         @ compare loop counter to 10, alter CPSR flag Z
      bne loop            @ branch if not equal (i.e. if Z is not set) to loop
stop:  b stop              @ Branch to label 'stop'
```

SUMMARY

- Microcontrollers combine a CPU with memory and peripherals in a single device
- The heart of the CPU is the ALU which performs basic operations
- The CPU fetches and decodes operations
- The CPU can alter registers (memory)
- Assembler is a very low-level language that maps directly to opcodes

FURTHER READING

- ‘Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications’ - Oshana and Kraeling
- ‘Embedded Systems Architecture, Second Edition: A Comprehensive Guide for Engineers and Programmers’ - Noergaard

4. CRASH COURSE IN C

OVERVIEW

- What high-level languages are used in the embedded space?
- Why C?
- How do we use it in an embedded system?

HIGH LEVEL LANGUAGES

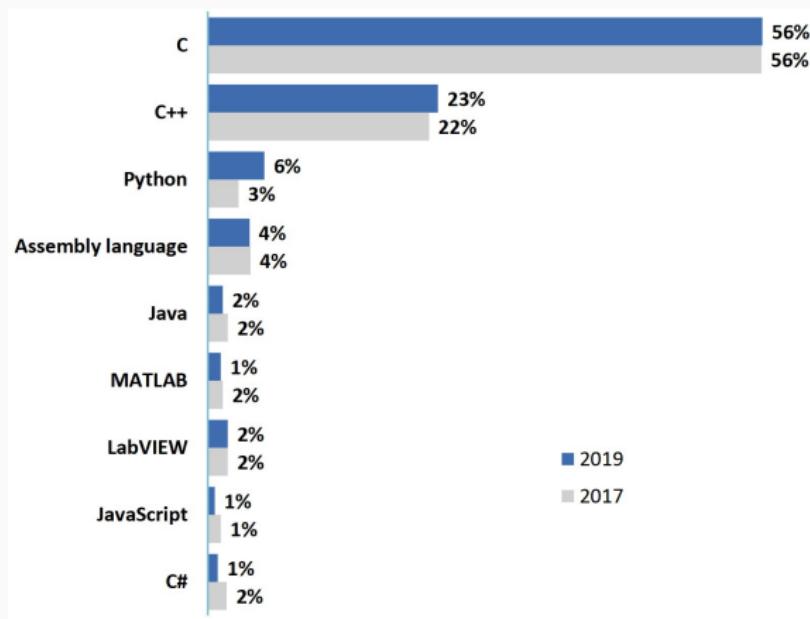
- Although assembler is great fun, it is not a very productive way of writing code
- High level languages provide layers of abstraction

“Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming.”¹

- These layers of abstraction are generally good, but they can be dangerous if not understood

¹Fred Brooks, No Silver Bullet

HIGH LEVEL LANGUAGES



2

²<https://www.embedded.com/>

- C is the dominant language in embedded systems
- It is not designed for embedded systems, so has no concept of real-time, multi-tasking
- Very low overhead - in some ways almost a superset of assembler

- OOP modelled evolution of C language
- Some C++ features (exceptions) are very heavy
- Embedded programmers are wary of dynamic memory allocation, so not all features of C++ are used
- Used in frameworks like Arduino³ and Mbed-OS⁴

³<https://www.arduino.cc/>

⁴<https://os.mbed.com/>

- Java is typically only suitable for high end processors
- Large overhead (garbage collection)
- Real-time Java aims to introduce concepts of time into Java

- Ada is used in safety critical systems such as avionics
- **Spark⁵** is a stripped down derivative for mid-range embedded devices
- Overhead is still relatively high - not suited to low end devices

⁵<https://www.adacore.com/about-spark>

- Rust is a language which is growing in popularity
- Ports exist for mid-range MCUs
- Sophisticated types
- Race free data-concurrency out-the-box
- Frameworks such as `embassy.rs`⁶ provide device independent wrappers

⁶<https://embassy.dev/>

```
fn main() {
    let p = stm32f30x::Peripherals::take().unwrap();

    let mut rcc = p.RCC.constrain();
    let gpioe = p.GPIOE.split(&mut rcc.ahb);

    let mut leds = Leds::new(gpioe);

    for led in leds.iter_mut() {
        led.on();
    }
}
```

⁷<https://pramode.in/>

MICROPYTHON

- MicroPython is a lean subset of python⁸
- Relatively low resources (16kB RAM, 256kB Flash)
- Easy to quickly script standard functions
- Outputs machine code (rather than bytecode)
- Still need to use C to extend functionality
- Can be used on RPi Pico⁹

⁸<http://micropython.org/>

⁹<https://www.raspberrypi.com>

MICROPYTHON

```
from machine import Pin
from time import sleep

led = Pin('LED', Pin.OUT)
print('Blinking LED Example')

while True:
    led.value(not led.value())
    sleep(0.5)
```

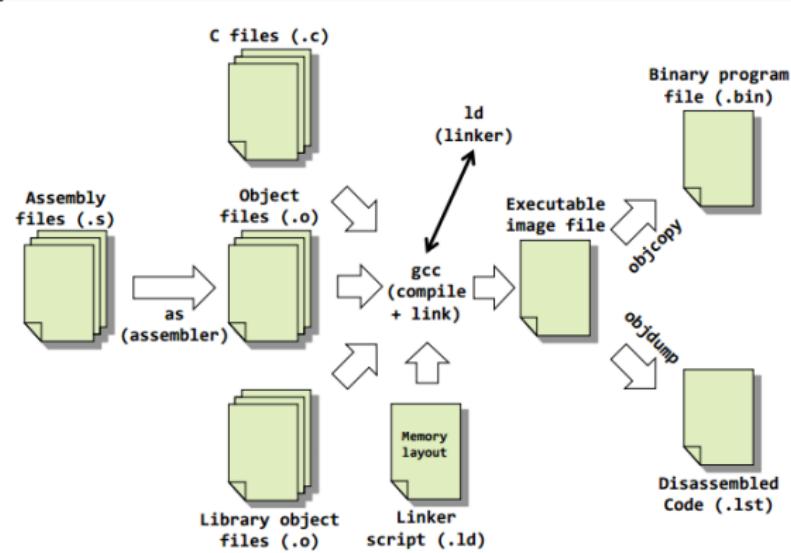
WHICH LANGUAGE TO USE?

- Depends on the application and its consequences of failure
- Ada/Spark has a number of nice constructs that make it appealing, but has a small, niche user-base
- Rust is gaining traction and being used increasingly for windows drivers
- C runs on pretty much any microcontroller and is well understood
- We use C in this course, but the principles are relatively general

- Began life as a language for PDP-11 (1971), an evolution of Combined Programming Language (CPL) invented by Christopher Strachey, the first Professor of Computer Science at Oxford
- Kernighan and Ritchie - The C Programming Language (1978)
- ANSI X3.159 (C89)
- ISO/IEC 9899 C90, C95, C99, C11
- Lots of variants and vendor specific extensions

C COMPILER

- C is a compiled (rather than interpreted) language
- The compiler reads the source code and creates code for the target machine
- C can be **cross-compiled** i.e. compiled for different targets



C: STANDARDS

- Standards are a set of rules
- In some applications, firmware has to be compliant with a particular standard
- Standards should be used in any firmware development, even if internal
- MISRA-C is a widely used standard, targetted at automotive industry
- Static analysis tools are used to automatically check compliance

DATA TYPES: INTEGERS

- Integers come in different widths and are either signed or unsigned
- The widths are target dependent, and thus `<stdint.h>` should be used¹⁰
- Depending on the target (e.g. 8 bit), some widths might not be available or require additional libraries
- Think carefully about whether you really need a 32 bit int on an 8 bit micro - it will cost you 12 cycles to do basic arithmetic

¹⁰MISRA Advisory Rule 13

DATA TYPES: INTEGERS

Type	Signed/Unsigned	Bits	Bytes	Minimum Value	Maximum Value
int8_t	Signed	8	1	-128	127
uint8_t	Unsigned	8	1	0	255
int16_t	Signed	16	2	-32,768	32,767
uint16_t	Unsigned	16	2	0	65,535
int32_t	Signed	32	4	-2,147,483,648	2,147,483,647
uint32_t	Unsigned	32	4	0	4,294,967,295

DATA TYPES: FLOATS AND DOUBLES

- C supports floats and doubles
- Depending on the target, these may be taken care of by a FPU (e.g. Cortex M4) or software library (e.g. Cortex M3)
- The difference in execution time between software or hardware is vast, typically 10-100 for addition and as much as 1000 for division
- Think **very** carefully about whether you really need a float on a device without hardware support

TYPE QUALIFIERS

- **const**: the value is read-only and cannot be modified¹¹
- **volatile**: something else that the compiler is not necessarily aware of can modify this value e.g. an input pin, ISR
- **static**: depends on the context
 - Top level of a file: not accessible from outside the file (file-scope)
 - Within a function: object retains its value between calls

¹¹in some targets, these are stored as literals

FUNCTIONS: DECLARATION

- Functions should be declared before use, typically in the header (.h) file
 - ‘Private’ functions can be declared in the source (.c) file
- This tells the compiler the call signature/prototype
- It also tells the programmer what the API is
- This also helps users to determine the way in which a function should be called

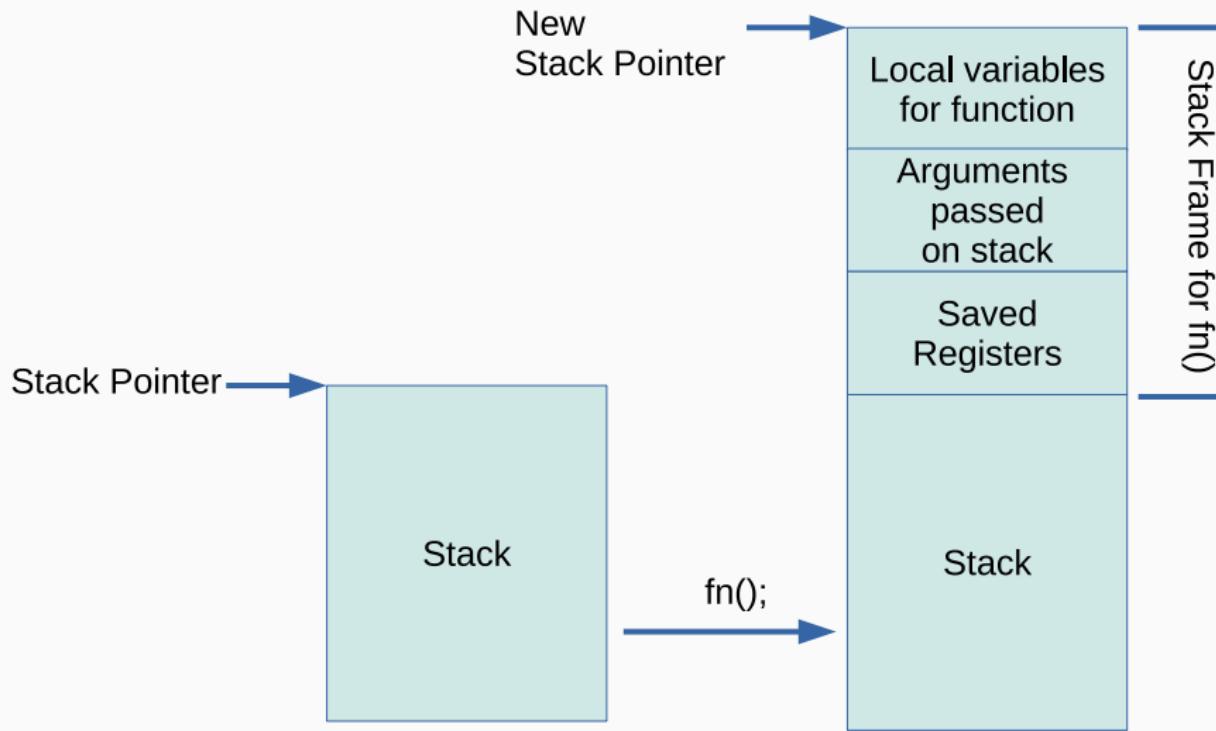
```
// consumes a and b, returns nothing
void device_set(uint8_t a, uint8_t b);
// consumes nothing, returns a uint16_t
uint16_t device_read(void);
```

FUNCTIONS: DEFINITION

- The definition of the function is the actual implementation
- This is typically done in the .c file, corresponding to the same prototype

```
uint16_t device_read(void)
{
    uint16_t ret;
    ret = read_serial();
    return ret;
}
```

CALLING A FUNCTION



FUNCTIONS: STACK FRAME

- Every function which is called consumes stack space
- The stack-frame stores the functions automatic variables, any arguments and previous context
- The stack occupies the same RAM as global variables
- Nested calls eat more and more stack space
- Stack overflow: stack collides with data
- Recursion is banned in embedded systems¹²

¹²MISRA Required Rule 70: "Functions shall not call themselves, either directly or indirectly."

VARIABLE SCOPE

Scoping issues can lead to tricky debugging¹³

```
uint8_t a = 0;
void fn(uint8_t a)
{
    uint8_t a = 3;
    printf("%u\n", a);
}
int main()
{
    a=1;
    fn(a);
}
```

¹³MISRA-C Rule 5.2: Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.

STATIC VARIABLES

Static variables maintain state

```
void fn(void)
{
    static uint8_t a = 1;
    printf("%u\n", a++);
}

int main()
{
    fn();
    fn();
}
```

INITIALIZATION

What happens with uninitialized variables?¹⁴

```
uint8_t x;
void fn(void)
{
    static uint8_t y;
    uint8_t z;
    printf("%u %u %u\n", x, y, z);
}
int main()
{
    fn();
}
```

¹⁴MISRA C 2004 rule 9.1: All automatic variables must be initialized before use.

ABSTRACT DATA TYPES: STRUCTS

- Structs provide a mechanism to create arbitrary/user defined types, composed of primitive types
- These help to encapsulate related parameters

```
struct sensor_state
{
    uint8_t status;
    uint8_t last_reading;
    uint16_t error_code;
};
```

TYPEDEFS

- Typedefs are a short-hand that allow us to relabel a primitive or derived type
- They make code more portable and changes only need to be made in one place
- However they shadow the real type, which can lead to issues

```
// Make a derived type
typedef uint16_t clock_time_t;
// Create a new variable with derived type clock_time_t (real type uint16_t)
clock_time_t system_time;
// They also can be used with structs, so we can leave out the struct keyword
typedef struct sensor_state sensor_state_t
```

POINTERS

- Pointers are objects that store a reference to a particular location in memory
- They are the subject of great mystery and bad practice
- If used correctly, they are powerful
- If abused, forget about robust code

POINTERS

- A pointer has a class the same as the object it is pointing to
 - A `uint8_t` pointer points to an unsigned 8 bit integer
 - A `struct sensor_state` pointer points to a struct
 - A pointer is defined using *
- ```
uint8_t * aPtr; // pointer to 8 bit uint, variable name is aPtr
```

# POINTERS

- To point to a location in memory, use &
- To indirectly access that location, we **dereference** the pointer, again using \*

```
uint8_t b = 3;
uint8_t c;
aPtr = &b; // aPtr now refers to b
*aPtr = 4; // b now has the value 4
c = *aPtr; // c now has the value 4 (dereferencing)
*aPtr = 5; // b now has the value 5
```

# CAREFUL WHAT YOU POINT TO...

What happens here?<sup>15</sup>

```
uint8_t * fn(void)
{
 uint8_t a = 9;
 uint8_t * b;
 b = &a;
 return b;
}
int main()
{
 uint8_t * k;
 k = fn();
 printf("%u\n", *k); // dereference
}
```

---

<sup>15</sup>MISRA 17.6: The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist

# POINTERS TO FUNCTIONS

- Pointers can also be made to functions<sup>16</sup>
- This is a shorthand that can loosen the coupling between modules, especially for functions like callbacks
- Function pointers are extremely useful to create OO-like patterns

---

<sup>16</sup>MISRA-C Required Rule 104: Non-constant pointers to functions shall not be used.

# POINTERS TO FUNCTIONS

```
// Our function
uint16_t add_numbers(uint16_t a, uint16_t b)
{
 return a + b;
}
// Define our function pointer - note it has the same call and return args
uint16_t (*fPtr)(uint16_t,uint16_t);
// Now tell it what to point to
fPtr = &add_numbers;
// And then dereference it - note brackets
uint16_t sum1 = (*fPtr)(2,3);
// Or just use it directly
uint16_t sum2 = fPtr(2,3);
```

# POINTERS TO FUNCTIONS

```
// We can make a typedef to make it cleaner
typedef uint16_t (*binaryFn)(uint16_t,uint16_t);
// And this makes it easy to pass as an argument
uint16_t calculate(binaryFn op, uint16_t a, uint16_t b)
{
 return op(a,b);
}
// use it:
binaryFn myOp = &add_numbers;
uint16_t sum3 = calculate(myOp,2,3);
```

## SUMMARY

- High level languages shelter the programmer from the details beneath
- High level languages still need to be used with caution
- Embedded programmers need to understand what is going on beneath the abstraction to avoid danger
- Standards like MISRA help, but they really are patching up the cracks rather than building a solid foundation

## FURTHER READING

- ‘The C Programming Language’ - K&R
- ‘Programming Embedded Systems: With C and GNU Development Tools, 2nd Edition’ - Barr and Massa
- ‘Embedded Rust Book’ - online<sup>17</sup>

---

<sup>17</sup><https://docs.rust-embedded.org/book/>

## 5. EMBEDDED SOFTWARE DESIGN

---

# OVERVIEW

- What are the main challenges of embedded system design?
- What are some methods for design and implementation?
- How do we make sure that what we design is actually going to work (forever)?

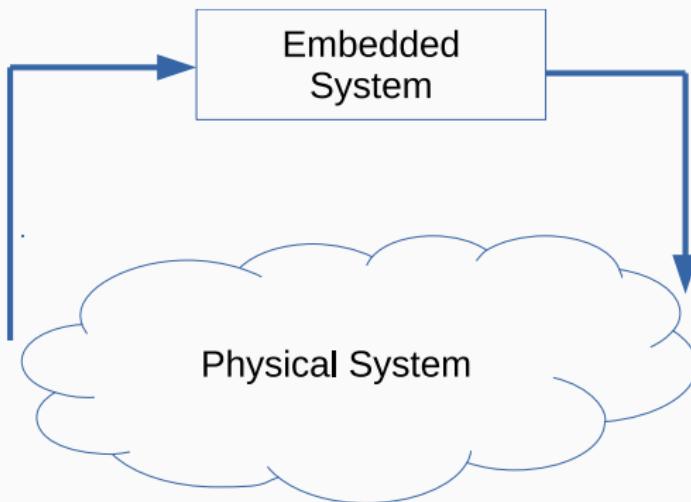
# TRANSFORMATIONAL VIEW OF COMPUTATION

- Transform some input to some output
- Termination is essential
- We try to verify correctness
- Algorithmic: e.g. search algorithm, optimization algorithm, compiler



# REACTIVE VIEW OF COMPUTATION

- Continuously interact with the environment
- Compute outputs i.e. **react** to input state
- Aim is to bring about desirable effects in the environment through a feedback loop



# REACTIVE SYSTEMS

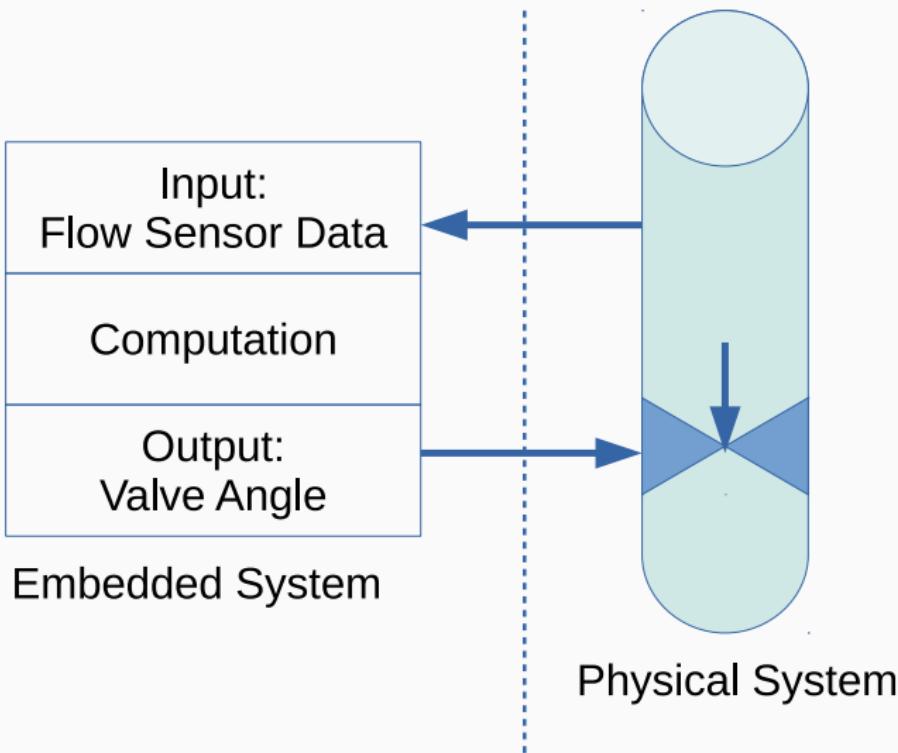
“A reactive system is a system that, when switched on, is able to create desired effects in its environment by enabling, enforcing or preventing events in the environment” - Wieringa

## PROPERTIES OF REACTIVE SYSTEMS

- Reactive Systems typically have most of these properties
  - non-terminating
  - interactive
  - interrupt-driven
  - state-dependent
  - environment-oriented
  - parallel
  - real-time
  - multi-rate

## EXAMPLE: FLOW RATE CONTROL

Control the rate that fluid flows through a pipe



## EXAMPLE: FLOW RATE CONTROL

- Monitor and Control
  - Sense the flow rate and actuate the valve
- Perform a meaningful operation
  - Control valve in safe way, according to dynamics
- Application Specific
  - System designed for this and only this purpose
- Optimized
  - Components and architecture specifically for application

## EXAMPLE: FLOW RATE CONTROL

- Resource Constrained
  - Small inexpensive microcontroller
- Real-time
  - Respond to changes in flow with guarantees
- Multi-rate
  - React to sensor flow, but also external commands

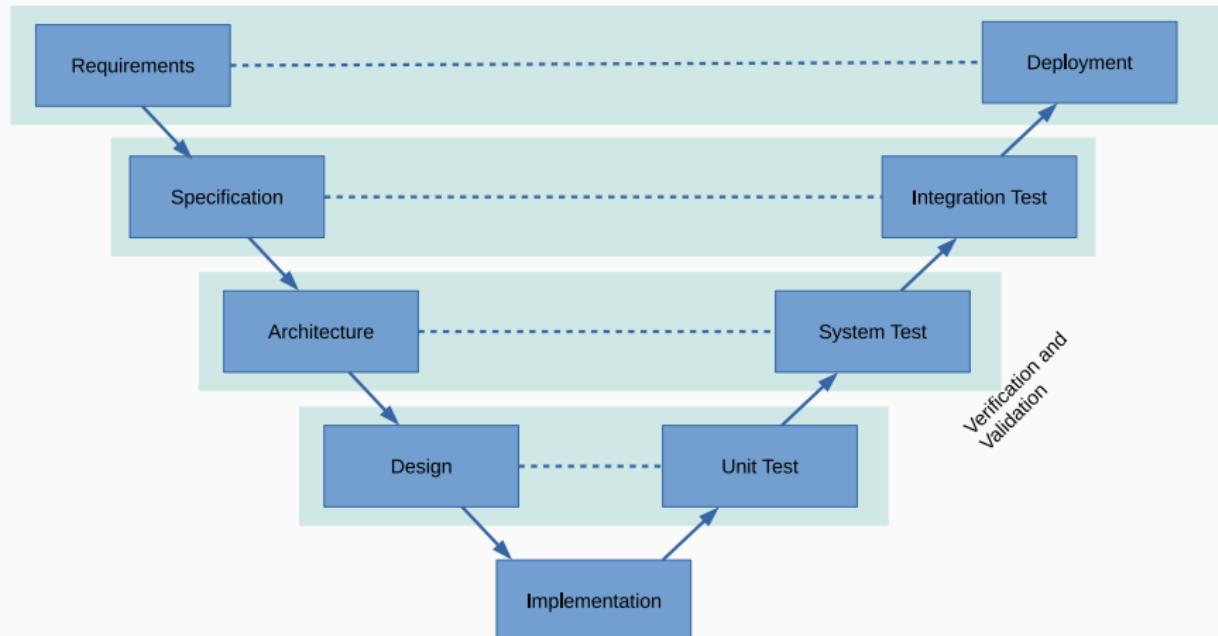
## KEY PROPERTY: TIME

- Correct execution of a program in C/C++/Haskell etc is completely decoupled from how long it takes
- All computing abstractions are built on this premise
- In the embedded world, we have to step outside this abstraction to incorporate time
- In some systems, being late (or even early) is the same as being wrong

# EMBEDDED DESIGN PROCESS

- Follows a similar set of steps to traditional software:
  - Requirements and Specification
  - Architecture and Design
  - Implementation and Integration
  - Verification and Validation (V&V)
  - Deployment and Maintenance

# VEE DIAGRAM OF EMBEDDED DESIGN



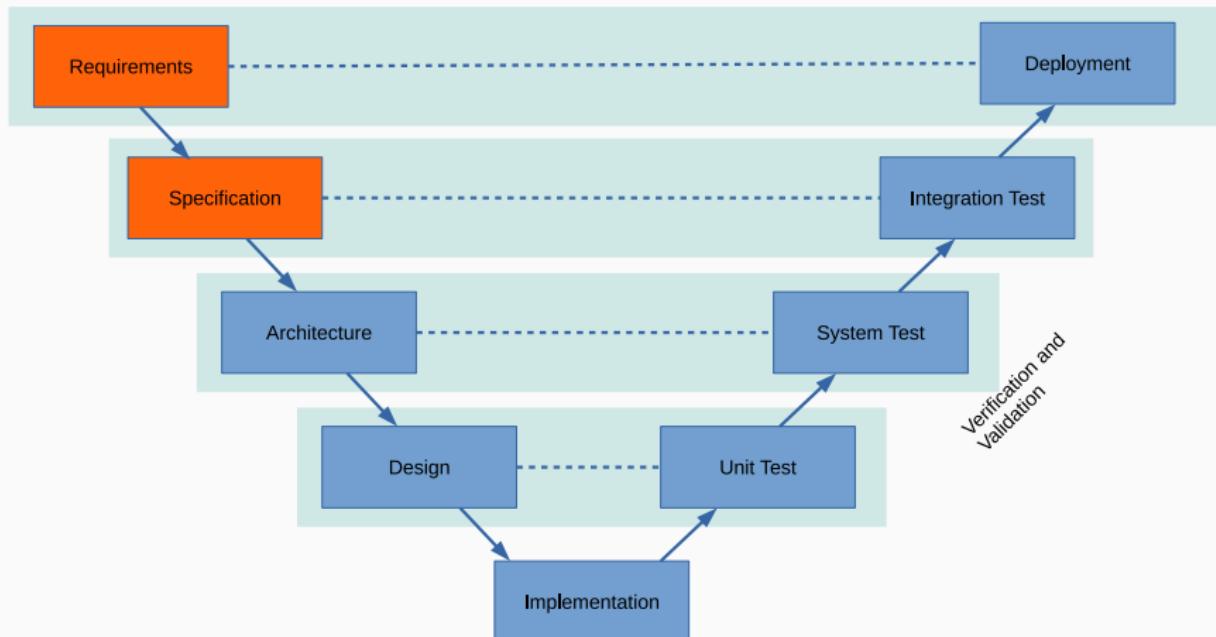
## MAIN CHALLENGES OF EMBEDDED SYSTEM DESIGN

- Very tight SW/HW coupling
- Limited debug functionality
- HW might not be ready when SW process starts
- Support for multiple platforms
- Debugging on hardware is slow and uninformative

# FOUNDATIONAL SOFTWARE ENGINEERING PRINCIPLES

- Rigour
- Separation of concerns
  - Modularity and decomposition
  - Abstraction
- Anticipation of change
- Generality
- Scalability
- From principles to tools

# REQUIREMENTS AND SPECIFICATION



# FUNCTIONAL REQUIREMENTS

- Problem definition in language of domain
- Specify the operation (behaviour) of the system
  - The room temperature must be maintained between 10 and 20 deg C
  - The lift must not move when the doors are open
- Operational specifications typically have the form:
  - IF stimulus (event) s occurs
  - AND system is in state C
  - PRODUCE response r
- Also known as Event-Condition-Action (ECA) rules
- Maps well to State Machines (coming later)

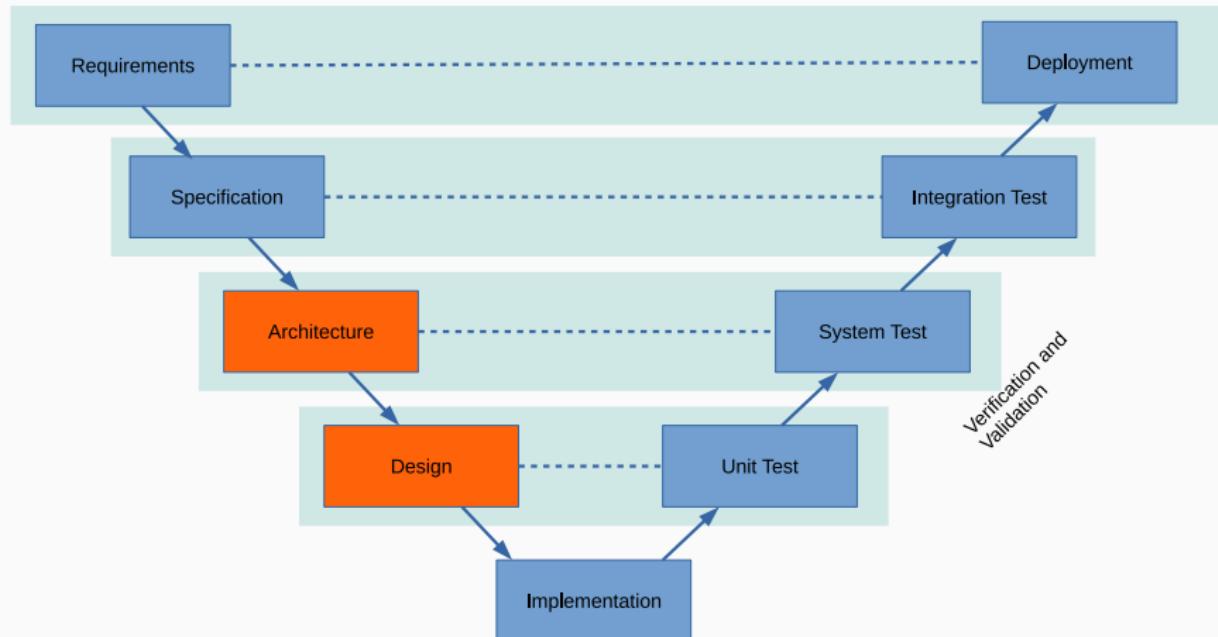
# EXTRA-FUNCTIONAL REQUIREMENTS

- Characterize **quality** of service
  - The device must run for 2 years from 2 AA batteries (for a portable system)
  - Image quality is not degraded (for an encoder)
  - Sensor data must be transmitted securely (for a sensor)
- As important as functional requirements
  - Constraints
  - Robustness
  - Dependability

# CHALLENGES

- Client may not know what is possible (unrealistic expectations)
- Hardware may not exist - have to design a system with hardware and software components
- Efficient integration with user
  - Do we need a touchscreen? Keypad?
- Environment
  - Does device need to be IP-rated?
  - Broad temperature range?

# ARCHITECTURE AND DESIGN



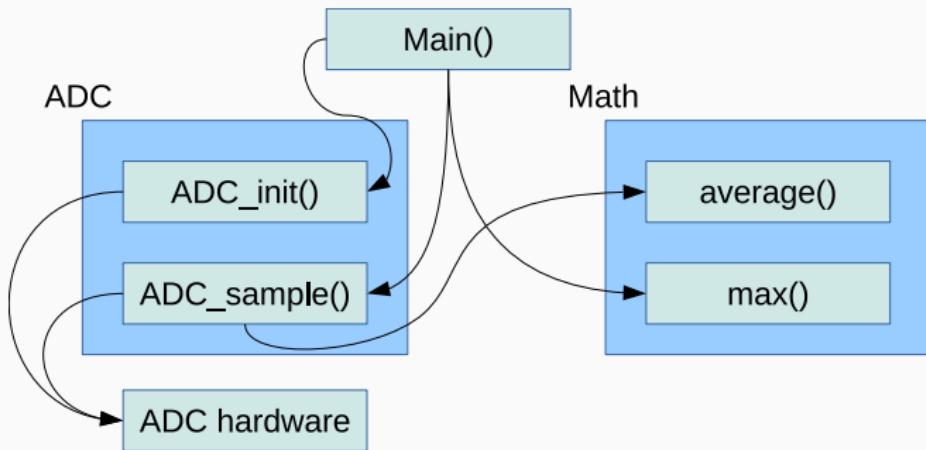
## ARCHITECTURE AND DESIGN

- Given a set of requirements (both functional and extra-functional), a system needs to be designed to meet them
- Hardware and Software teams need to work in parallel
- Many different views of architecture, depending on perspective

## CALL GRAPH

- Shows organization of modules and their dependencies
- Boxes: subroutines or objects, with functions they provide
- Arrows: subroutine or method calls
- Strengths:
  - Shows flow of control
  - Very useful for debugging and testing
  - Useful for shared (concurrent) resources

## CALL GRAPH: EXAMPLE

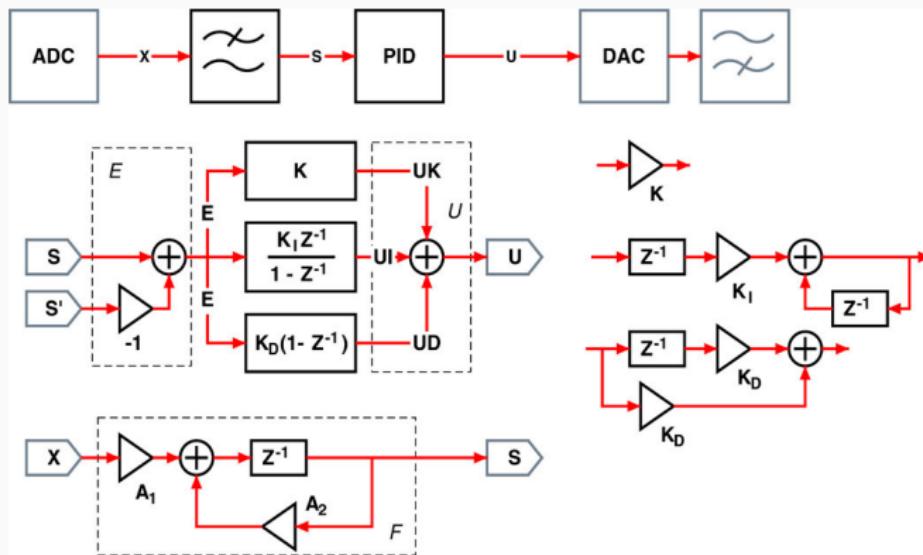


## DATA FLOW DIAGRAM

- Shows how data and signals are communicated between modules
- Boxes: computation or transformation
- Arrows: signal
- Strengths:
  - DSP algorithms and filtering
  - Temporal sequence of events

# DATA FLOW DIAGRAM: EXAMPLE

- Some signal flow diagrams at various levels of detail<sup>1</sup>

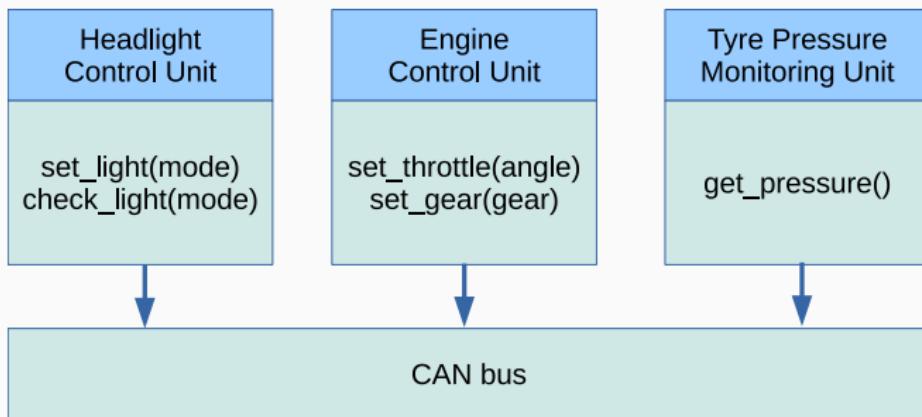


<sup>1</sup><http://spie.org/x48813.xml>

# HARDWARE ALLOCATION DIAGRAM

- Shows how software is mapped to hardware
- Boxes: Processors
- Arrows: communication links and messages
- Strengths:
  - Distributed systems

# HARDWARE ALLOCATION DIAGRAM

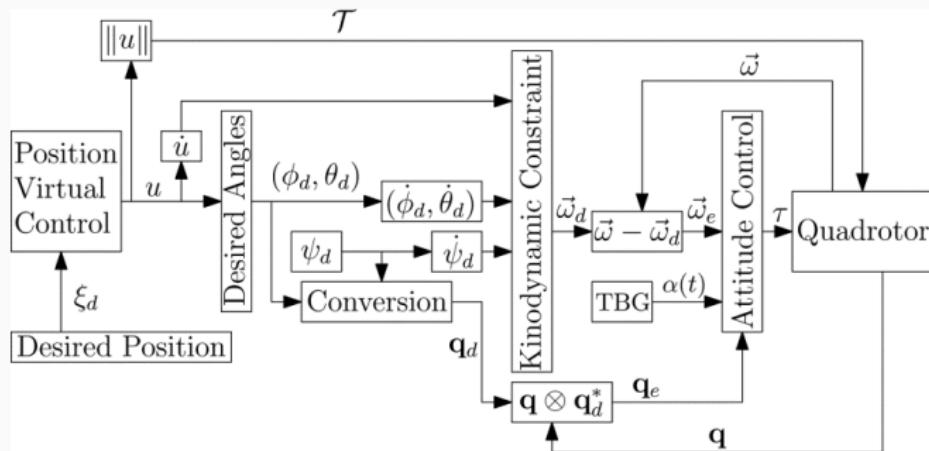


# CONTROL HIERARCHY DIAGRAM

- Shows structure of control loops
- Boxes: controllers
- Arrows: set points, inputs and outputs
- Strengths:
  - Distributed control
  - Interaction between plants

# CONTROL HIERARCHY

- Example for a quadrotor<sup>2</sup>



<sup>2</sup>Position-Yaw Tracking of Quadrotors, Sanchez-Orta et al., Journal of Dynamic System Measurement and Control, 137(6), 2011

- Once architecture has been specified, design system at increasing levels of detail
- Important considerations
  - Compatibility: different hardware platforms
  - Extensibility: new features can be added
  - Robustness: tolerant to faults
  - Modularity: well defined components

# MODELLING

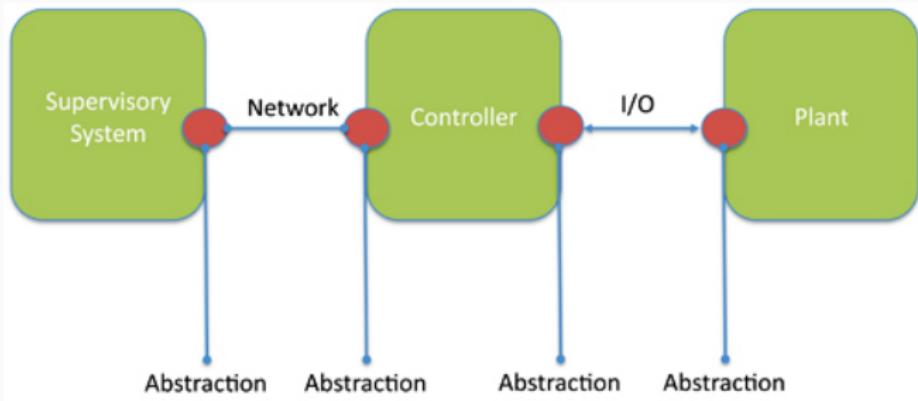
- Formal specification of system and its components
- Unlike conventional software, we often need to model the physical world as well
- Different types of modelling
  - Behavioural
  - Structural
  - Simulation-centric (executable)
- Models can be graphical (UML/SysML) or textual (SystemC)

- Example of modelling a controller to regulate operation of a plant (process)<sup>3</sup>
- Some models are executable, whereas some are more architectural

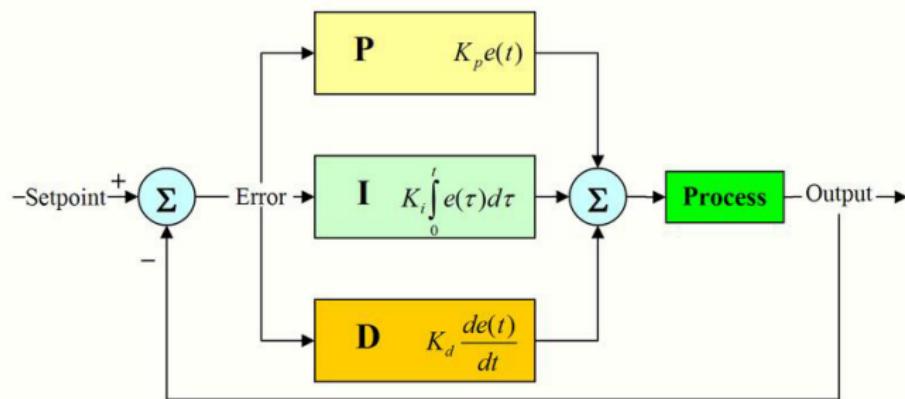
---

<sup>3</sup>For more information, see Ch 3. 'Software Engineering for Embedded Systems', Oshana and Kraeling

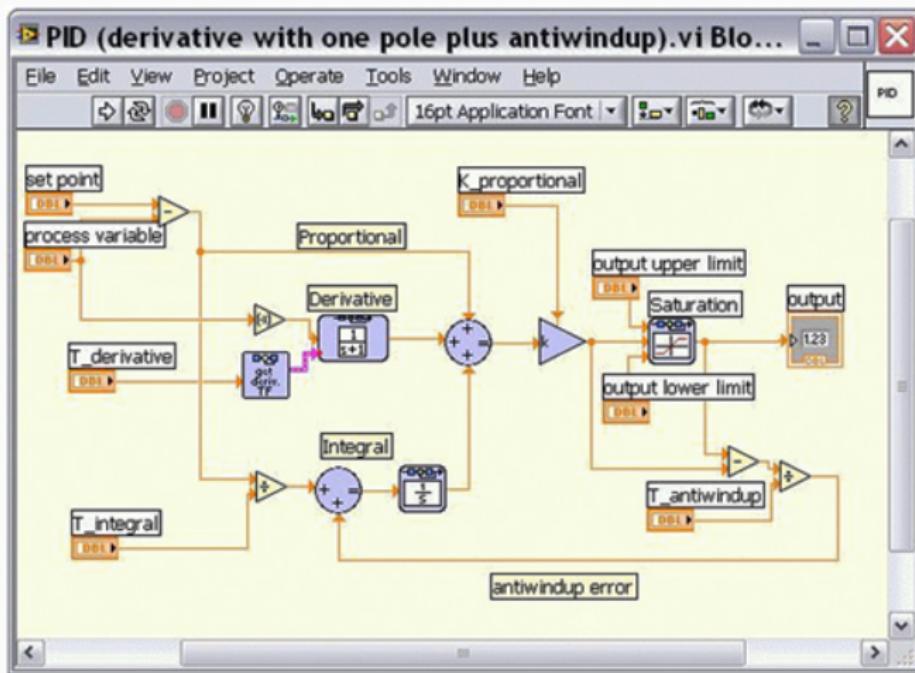
# MODELLING A PLANT: HIGH LEVEL ABSTRACTION



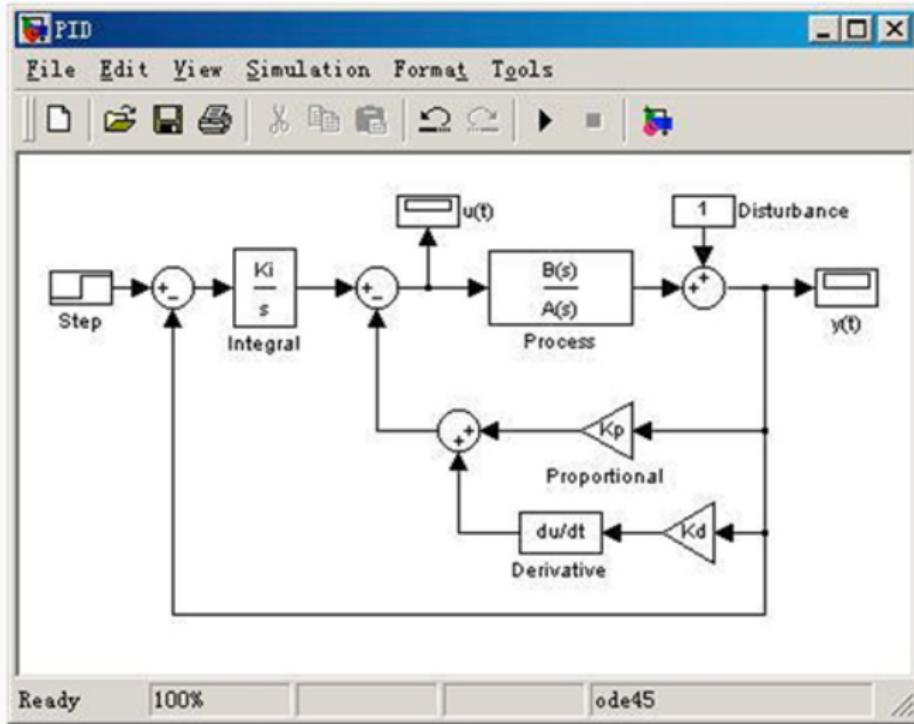
# MODELLING A PLANT: MATHEMATICAL MODEL



# MODELLING A PLANT: DATAFLOW DIAGRAM



# MODELLING A PLANT: SIMULATION DIAGRAM



# MODELLING A PLANT: FIRMWARE IMPLEMENTATION

```
previous_error = setpoint - process_feedback
integral = 0
start:

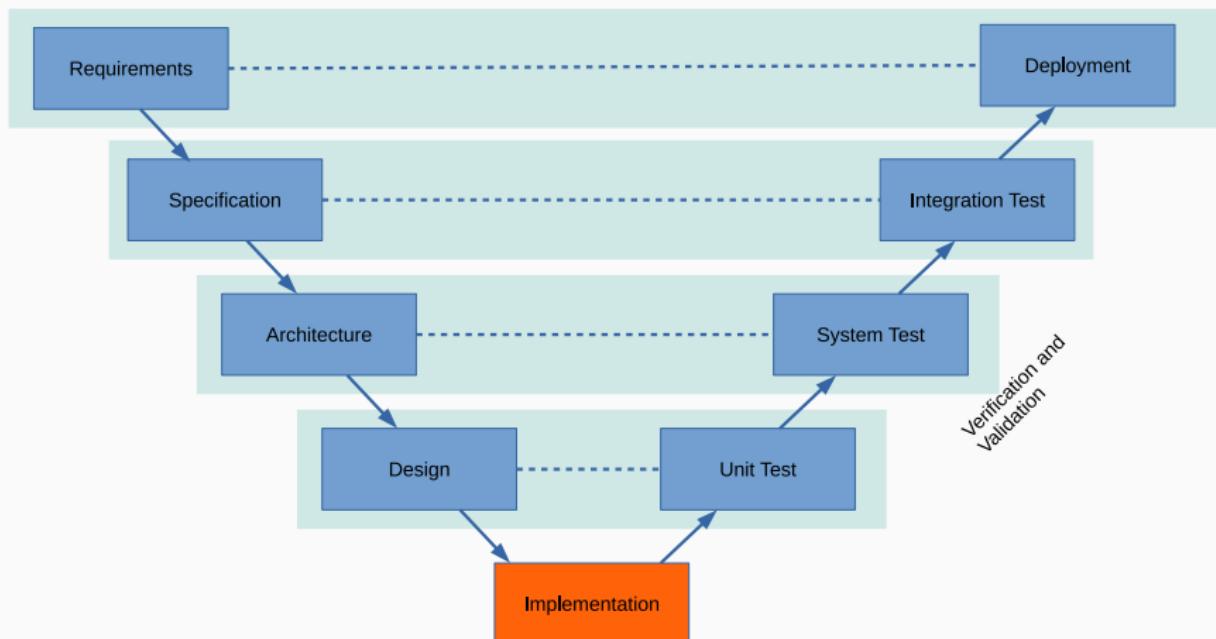
 wait(dt)
 error = setpoint - process_feedback
 integral = integral + (error*dt)
 derivative = (error - previous_error)/dt
 output = (Kp*error) + (Ki*integral) + (Kd*derivative)
 previous_error = error

goto start
```

## MODELLING: SUMMARY

- Many different viewpoints of the same process
- Differing levels of abstraction
- Graphical models
  - Show grouping and structure easily
  - Can be vague
- Textual models
  - Can be difficult to see structure
  - Precise
- Model-based design tools can generate code to match the model...

# IMPLEMENTATION



## IMPLEMENTATION AND INTEGRATION

- Aim for modularity and component reuse
- Avoid strong coupling between components
  - Components do not even have to be in the same physical device
- Use abstraction barriers to break dependencies

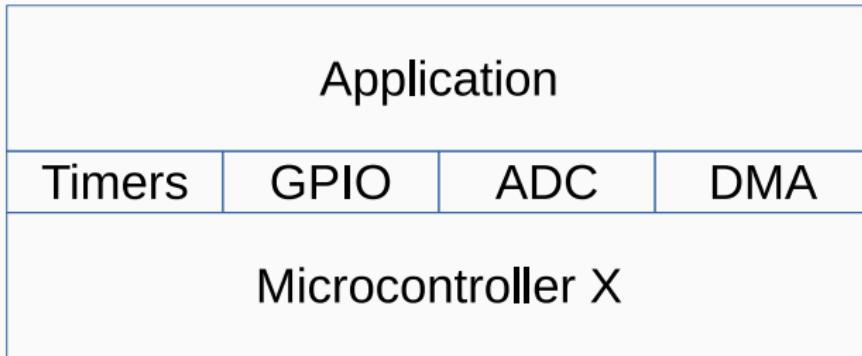
# MODULARITY

- Aims of modular design<sup>4</sup>
  - Modules should have high cohesion
  - Modules should have low coupling
  - Modules should hide implementation details
  - Modules should have good composability
  - Modules should have lowest complexity possible

---

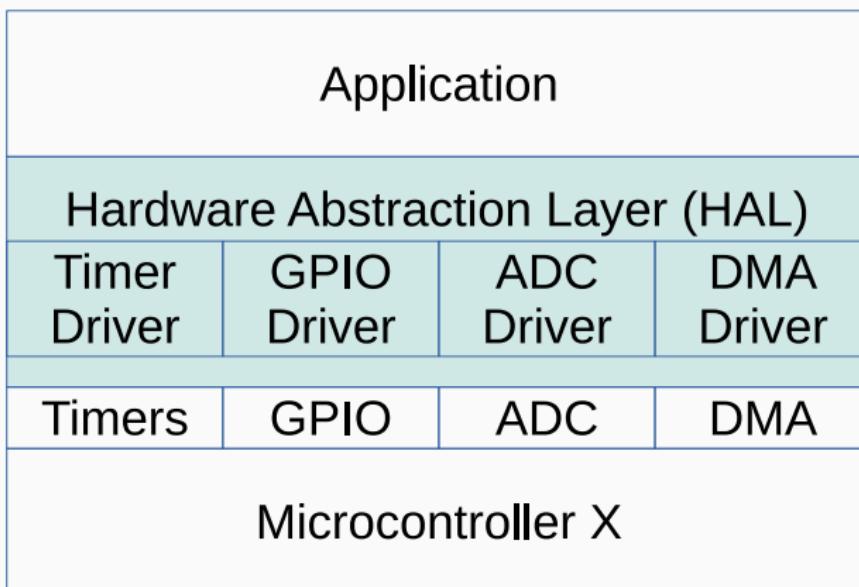
<sup>4</sup>Better Embedded Software, Koopman

# TIGHT COUPLING



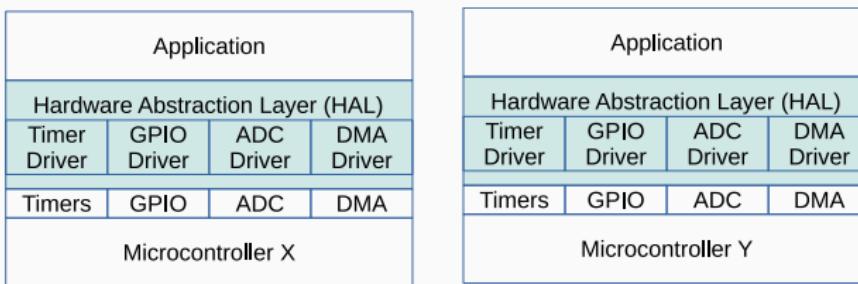
# HARDWARE ABSTRACTION LAYER

- Use a hardware abstraction layer (HAL) to break dependencies on hardware
- Also known as a board support package (BSP)



# HARDWARE ABSTRACTION LAYER

- HALs make migration to other platforms simpler
- ARM devices use CMSIS as a HAL framework to provide vendor neutrality



# COMPLEXITY

- Lower complexity is a good thing
- Different measures of complexity
  - Lines of Code
  - Cyclomatic complexity (knots)
- Refactor code into cohesive modules to reduce complexity

## CODING STANDARDS

- Many different standards exist, depending on application and safety level
- Even if application is not safety critical, using a standard will aid code quality
- MISRA C is one of the most commonly used
- Don't ignore compiler warnings!

## CONTRACTS

- Design By Contract was invented by Meyer for Eiffel
- It has been widely used for OO languages
- It still has relevance for procedural languages like C
- In ADA 2012, DBC has been baked in
- Spark takes it a step further and checks contracts will be valid before it even compiles!

# CONTRACTS

- Capture and check:
  - Preconditions
  - Postconditions
  - Invariants
- Can be used as inputs to formal verification tools

# POOR MAN'S DBC IN C

- Conditions can be checked using `assert()`<sup>5</sup>:

```
#define REQUIRE(a) ((void)(a), assert(a))
#define ENSURE(a) ((void)(a), assert(a))
#define INVARIANT(a) ((void)(a), assert(a))
```

---

<sup>5</sup>we define our own commands so we can remove them from runtime later

# DBC EXAMPLE

```
float root(float a, float b, float c)
{
 float result;
 REQUIRE (a != 0);
 REQUIRE ((b*b - 4*a*c) >=0);

 result= (-b + sqrt(b*b - 4*a*c))/2*a;
 ENSURE (isfinite(result));

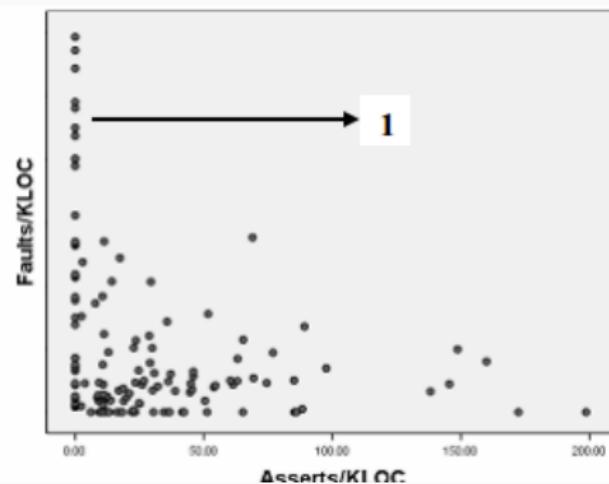
 return result;
}
```

## DBC EXAMPLE: HARDWARE FLAG CHECKS

```
void timer_start(uint32_t period)
{
 REQUIRE (TIMER_INIT != 0);
 ...
 ENSURE (TIMER_STATUS == TIMER_RUNNING);
}
```

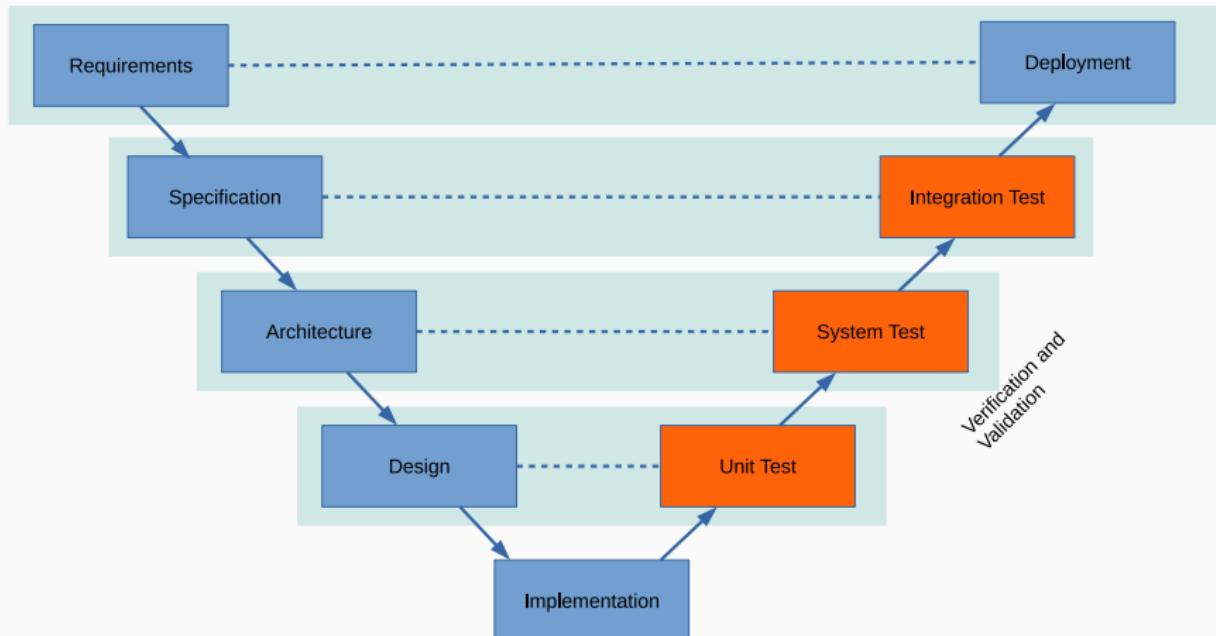
# POWER OF ASSERTIONS

- Using the principles of DBC with sanity checks makes a vast difference to code quality<sup>6</sup>
- In some ways, this could be because they force the programmer to think more actively about failure modes

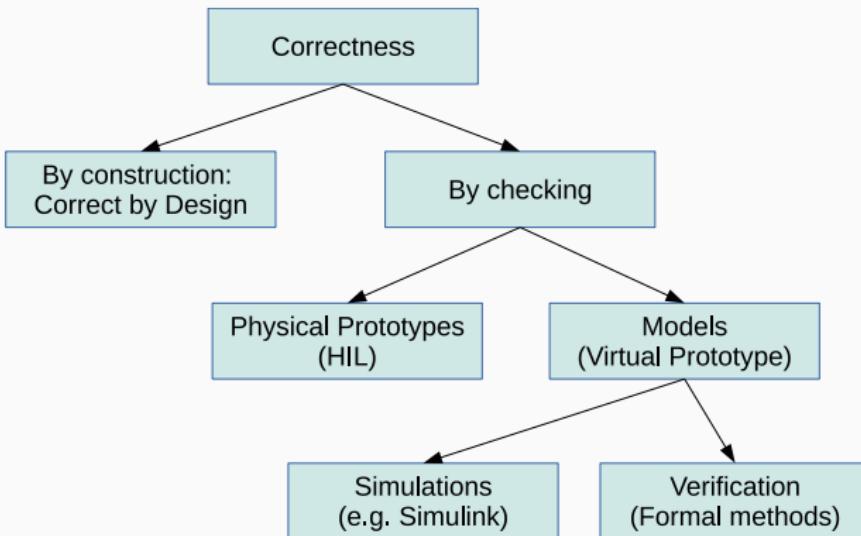


<sup>6</sup>Microsoft Research

# VERIFICATION AND VALIDATION



# VERIFICATION AND VALIDATION



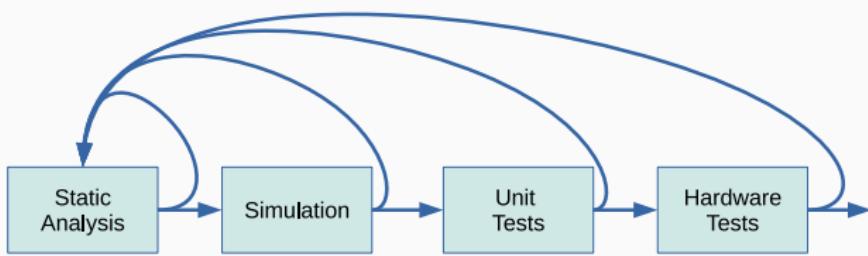
## CORRECT BY CONSTRUCTION

- One way to make sure there are no bugs is to formally specify the program
- This is the flip side of using an assert
  - We essentially prove that an assert cannot fail
- Predicate Logic is one class of formal method
  - Z - partially developed at Oxford
- CSP is another approach that captures temporal relationships
  - Invented by Sir Tony Hoare

## CHECKING CODE

- Formal methods however are not widely used in industry
  - Niche areas in safety-critical systems
  - They do not scale well to large systems
  - The alternative is to write code and then check that it will do what it says on the tin
  - This can be by inspection (static analysis) or by execution (dynamic analysis)

# CHECKING CODE



## CODE INSPECTION

- This is one of the simplest methods for ensuring code quality
  - “Many eyes make all bugs shallow”
- Peer review of code ensures that it adheres to standards
- It also works as a means of communicating to team members what has been done
- Pair-programming could be viewed as an extreme form of code-inspection

# STATIC ANALYSIS

- There are many automated tools to check code for issues
  - Lint, Clang, Splint
- Some tools also check for compliance to standards e.g. MISRA-C

```
void do_something(uint8_t a)
{
 if (a == 3)
 {
 do_something_else();
 };
}
```

{misra3.c 3 Warning 506: Constant value Boolean [MISRA 2012 Rule 2.1, required]}

## DYNAMIC ANALYSIS

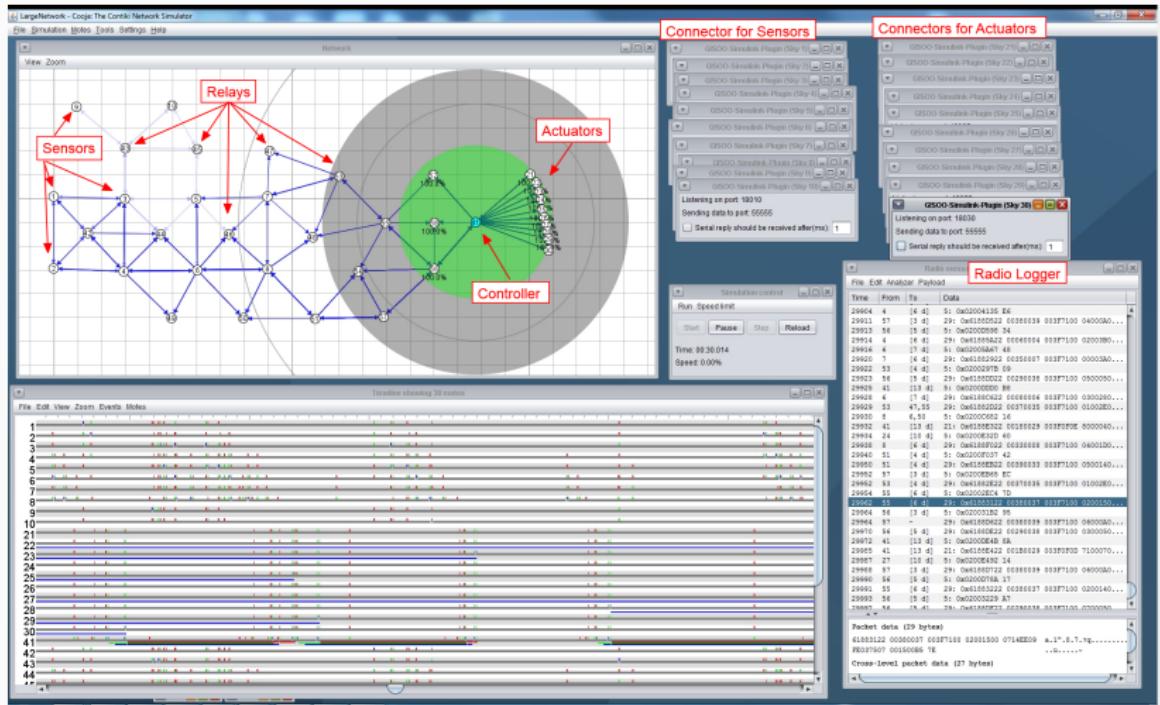
- In dynamic analysis, we actually execute the code to check that it meets the specifications
- We don't have to execute on the target:
  - Build a model and check that this works
  - Simulate
  - Cross-compile and place in test-harness
- Different approaches are suited to different phases of the project
- Last step: test on the hardware

- Construct a model of the software
  - From specification
  - Directly from code
- Check that the model works correctly
- Many model checking tools:
  - CBMC
  - PRISM: probabilistic model checking

# SIMULATION/EMULATION

- Run the code on a virtual prototype
  - Android ADK widely used example
  - COOJA is a Java simulator for IoT devices
  - Simics is a general purpose simulator
  - ReNode is an ARM simulator (open-source)
- Cycle-accurate emulators are typically slow
  - Abstraction layers can replace real components with emulated components

# COOJA: IOT SIMULATION



# TESTING

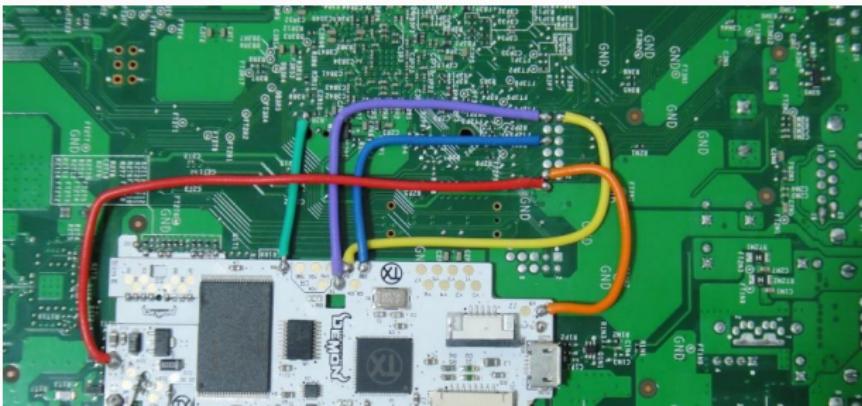
- Adopt conventional testing strategies
  - Test-driven design
  - Regression testing
- Cross-compile and unit test code on PC
- Much faster and easier to debug
- Crashes can actually give enough information to determine cause

## TESTING ON TARGET PLATFORM

- For system integration or low-level hardware, it is necessary to test on target platform
- This should only be done when all other tests have passed
- Debug output can be provided by:
  - LEDs (worst)
  - printf() if available (not much better)
  - JTAG (best)

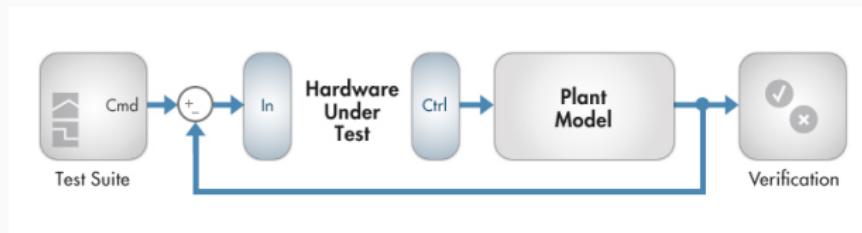
# JTAG DEBUGGING

- Virtually all microcontrollers typically have a JTAG interface
- This is a special standardized peripheral that provides direct access to the CPU core
  - Four wires (TCK, TMS, TDO, TDI)
- Allows registers to be inspected, modified
- Allows CPU to hit breakpoints, single step

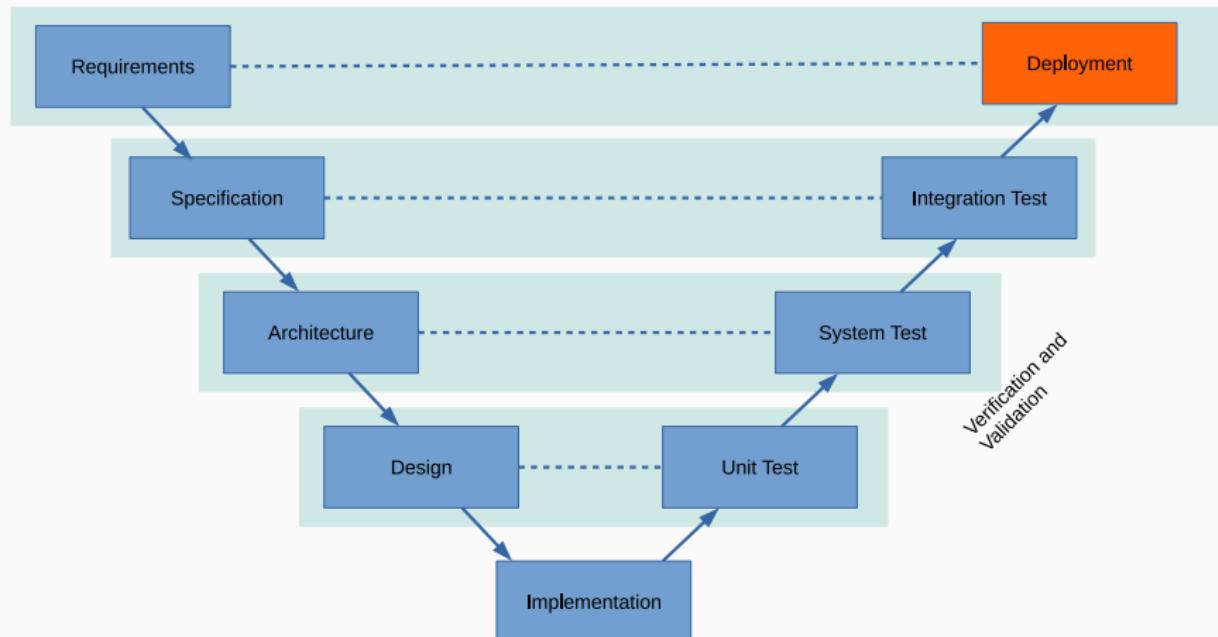


# HARDWARE-IN-THE-LOOP SIMULATION

- PC equipped with ADC and DAC running simulation of physical system
- Hardware thinks it is controlling a real plant/system
- Very useful for rapid prototyping



# DEPLOYMENT AND MAINTENANCE



## DEPLOYMENT AND MAINTENANCE

- The process is not over once the device is out in the world
- It is vital to provide some sort of logging or diagnostics to trace failures
- Logs are key to postmortems and continuous improvement
  - Black-boxes in avionics close the quality feedback loop

# LOGGING

- Important questions
  - What to log
  - How often to log?
  - What is the cost of logging on the system?
  - What is the cost of recovery?

## CODE UPDATES

- Although firmware is meant to be perfect the moment it leaves the factory, it is likely to need some improvements over the years
- Options:
  - Hardware reprogramming (e.g. via JTAG)
  - Over The Air (OTA) updates (e.g. via 3G)
- Be very careful of introducing new bugs with an update
  - Worst case scenario: bricked device
  - Use a golden image as a fail-safe

## SUMMARY

- Embedded design is much more than software design, it is **system** design
- Components are part of a much larger whole
  - Use abstraction and modularization to focus on parts
  - Compose parts together to make the system
- Embedded devices are invisible until they fail
  - Use discipline and software engineering principles to reduce risks

## FURTHER READING

- “Software Engineering for Embedded Systems” - Oshana and Kraeling
- “The Art of Designing Embedded Systems” - Ganssle
- “Embedded System Design: Modeling, Synthesis and Verification” - Gajski, Abdi and Gerstlauer

## 6. TIMERS AND INTERRUPTS

---

# OVERVIEW

- How do we keep track of time?
- How do we react to triggers and events in a timely fashion?

# THE REAL-WORLD

- The fundamental property of the real-world is that time waits for no device
- Delays can be built in software, but they are imprecise and vary across targets

```
void sw_delay(uint32_t d)
{
 while (d-- > 0)
 {
 // do nothing
 }
}
```

- They also block the processor whilst executing the delay

# TIMERS

- One of the most important peripherals in a microcontroller
- Allow a device to keep track of real-world time
- Enable an embedded system to **deterministically**
  - sense
  - control/actuate
- Modern microcontrollers have a number of timers
  - e.g. ARM based STM32F4 has up to 14 timers
- Timers can also be external (i.e. a separate chip)

# USE OF TIMERS

- Output:
  - Keep traffic light green for 10 seconds
  - Generate spark in engine every 20 msec
- Input:
  - Measure car's speed
- Timers are just counters which are clocked at a regular rate

## ELAPSED TIME

- Elapsed time counters simply count the number of seconds since something happened
  - since bootup
  - since last time the timer was reset
- They will overflow at some stage though, depending on the timer resolution (e.g. 16 bit) and update rate
- Be very careful that this does not lead to Y2K or 2038 problem...

# REAL TIME CLOCKS

- Real<sup>1</sup> Time Clocks (RTCs) have a concept of “human” time i.e. they are referenced to some epoch
  - e.g. 1 Jan 1970 is the start of the UTC epoch
- Modern microcontrollers often have a built-in RTC
- To maintain time across resets and power-down, RTC's are backed up:
  - battery
  - super-capacitor

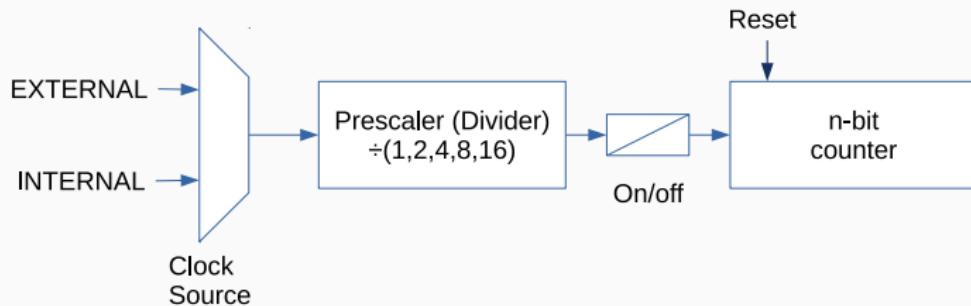
---

<sup>1</sup>Real time is only real at time of setting timer. Without periodic synchronization, they will drift.

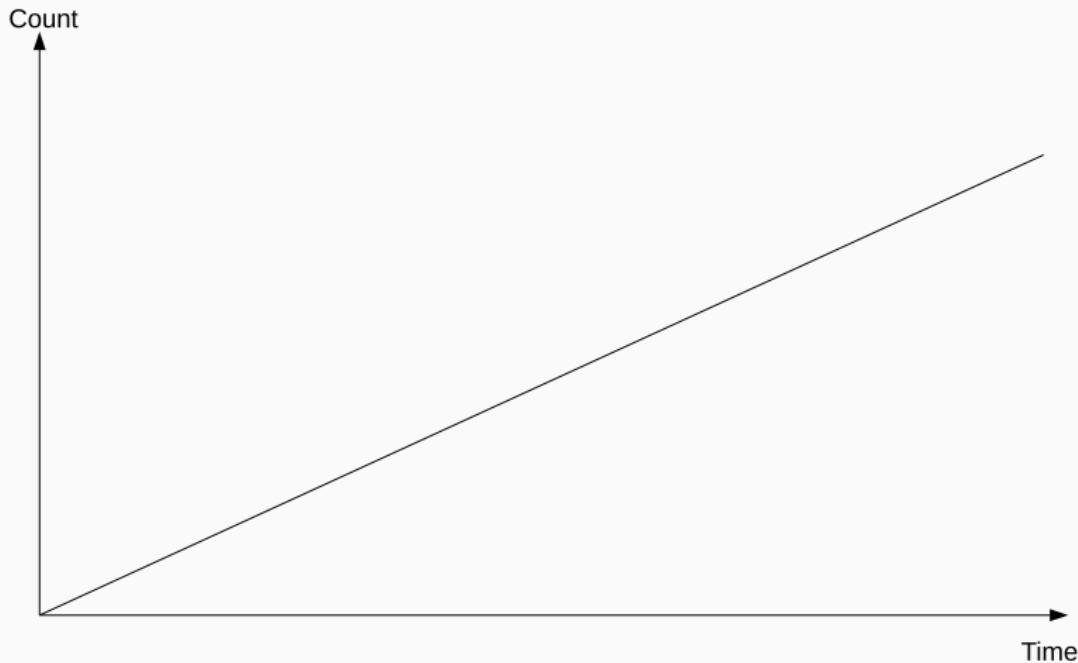
## PERIOD TIMERS

- Timers can also be configured to reset when they reach some value
- When they reset, they set a flag that indicates that it overflowed
- For example, a timer for sampling audio could be configured to reset 16 000 times per second

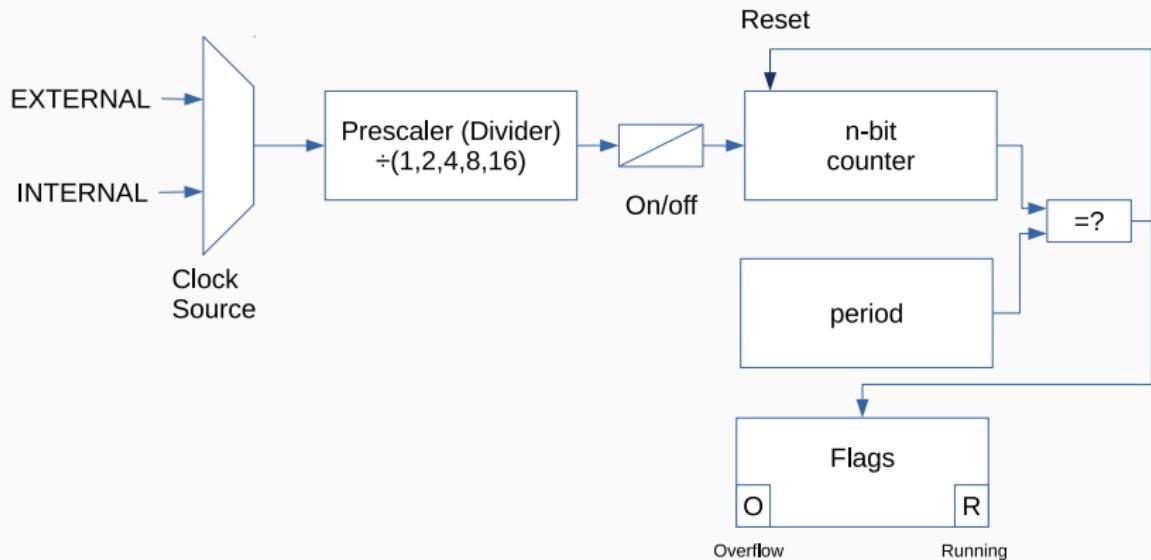
# TIMER BLOCK DIAGRAM



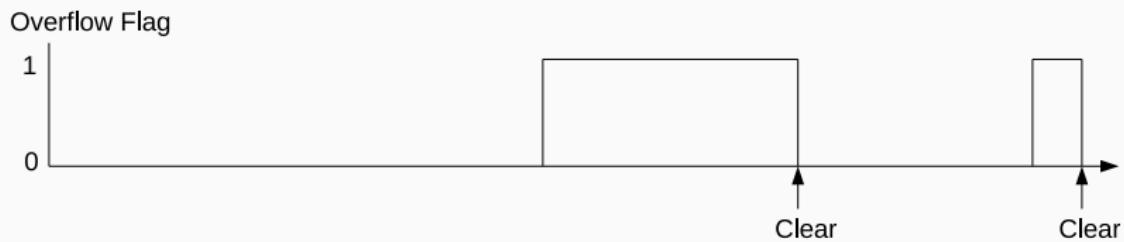
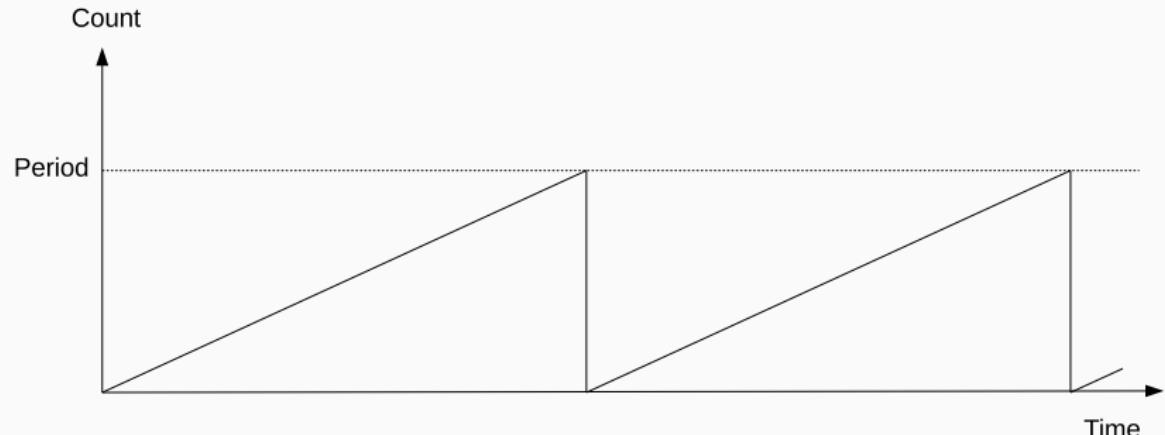
# TIMER OPERATION



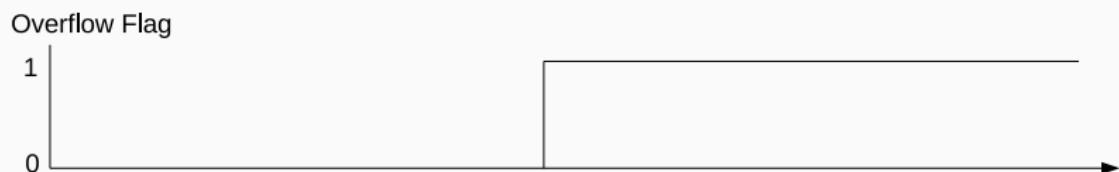
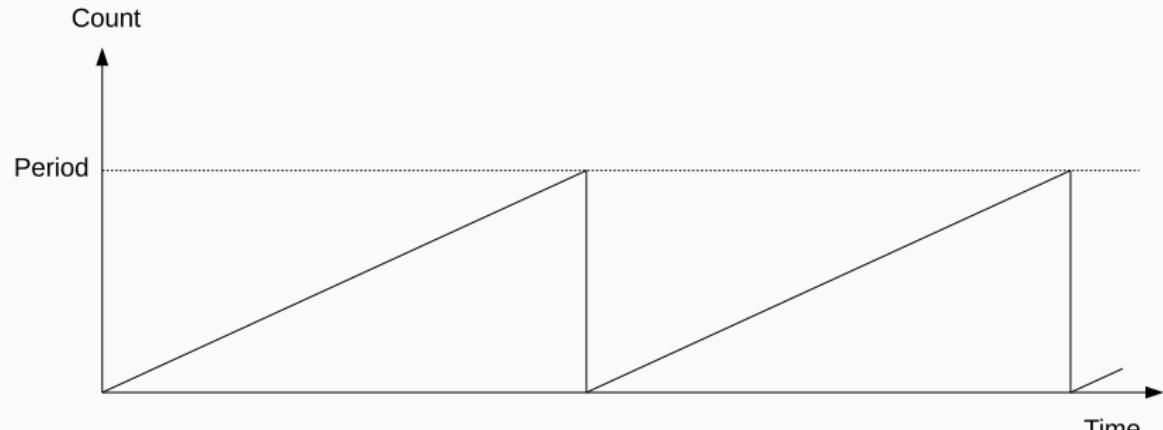
# TIMER OPERATION



# TIMER OPERATION



# TIMER OPERATION



# INTERVAL TIMER API

```
// Initialize and start timer from 0
void Timer_start(void);
// Get the current reading
uint32_t Timer_read(void);
// Set the timer to a particular value
void Timer_set(uint32_t value);
```

# REAL TIME CLOCK STRUCT

```
// Is this a good representation for a RTC?
struct rtc_time
{ uint8_t year;
 uint8_t month;
 uint8_t day;
 uint8_t hour;
 uint8_t minute;
 uint8_t second;
};
```

# PERIODIC TIMER API: FIRST THOUGHTS

```
// Initialize a timer with a certain period
void Timer_init(uint16_t period);
// Start a timer
void Timer_start(uint32_t delay);
// Check if a timer is expired
uint8_t Timer_expired(void);
// Stop a timer
void Timer_stop(void);
```

# HANDLING MULTIPLE TIMERS: ONE APPROACH

```
// Initialize timer x with a certain period
void Timer_x_init(uint16_t period);
// Start timer x
void Timer_x_start(uint32_t delay);
// Check if timer x is expired
uint8_t Timer_x_expired(void);
// Stop timer x
void Timer_x_stop(void);
```

## HANDLING MULTIPLE TIMERS

- Having multiple timer functions with different numbers:
  - e.g. Timer\_1\_init();
  - e.g. Timer\_9\_init();
- Not very elegant
- Changes to API will require changes to all timer code

# MORE FLEXIBILITY WITH MULTIPLE TIMERS

```
typedef struct TimerObject
{
 uint8_t identifier;
}timer_t;
// Initialize a timer with a certain period
void Timer_init(timer_t * t,uint8_t identifier,uint16_t period);
// Start a timer
void Timer_start(timer_t * t,uint32_t delay);
// Check if a timer is expired
uint8_t Timer_expired(timer_t * t,void);
// Stop a timer
void Timer_stop(timer_t * t,void);
```

# USING TIMER OBJECTS

```
timer_t tickTimer;
timer_t tockTimer;
Timer_init(&tickTimer,1,100); // Use timer1 for Tick
Timer_init(&tockTimer,2,100); // Use timer2 for Tock
Timer_start(&tock_Timer,200); // Start Tock Timer
```

## ADVANTAGES WITH USING A TIMER OBJECT

- API looks the same, regardless of number of timers in the system
- We can easily swap timers and all further references by changing the “constructor”
- The user is not aware of the implementation
  - Timers could be external
- Makes testing much simpler
  - Use a mock timer to simulate timer calls
- We are essentially using an OO design pattern to allow multiple physical objects to share a common interface

# USING A PERIODIC TIMER: POLLING

```
int main(void)
{
 Timer_start(&tick_timer,1000);
 while(1){
 LED_toggle();
 while (! Timer_expired(&tick_timer))
 {
 // do nothing
 }
 }
 return 0;
}
```

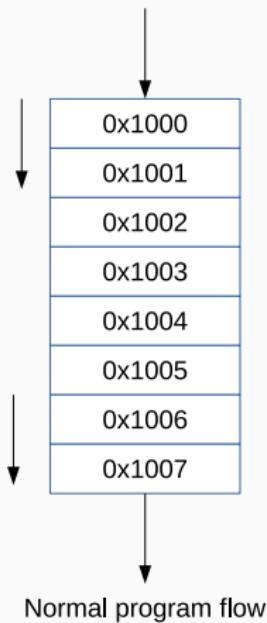
## ISSUES WITH POLLING

- Polling blocks the CPU from doing anything else: blocking-wait
  - In some ways, this is like continually checking your e-mail
- A better way would be for the timer to signify the CPU that it is done
  - This is like push-notifications
- Additionally, if we are doing something else and don't notice that the timer has expired, we could accumulate a large, possibly variable, delay
- We can also miss an arbitrary number of rollovers, as we only check the flag

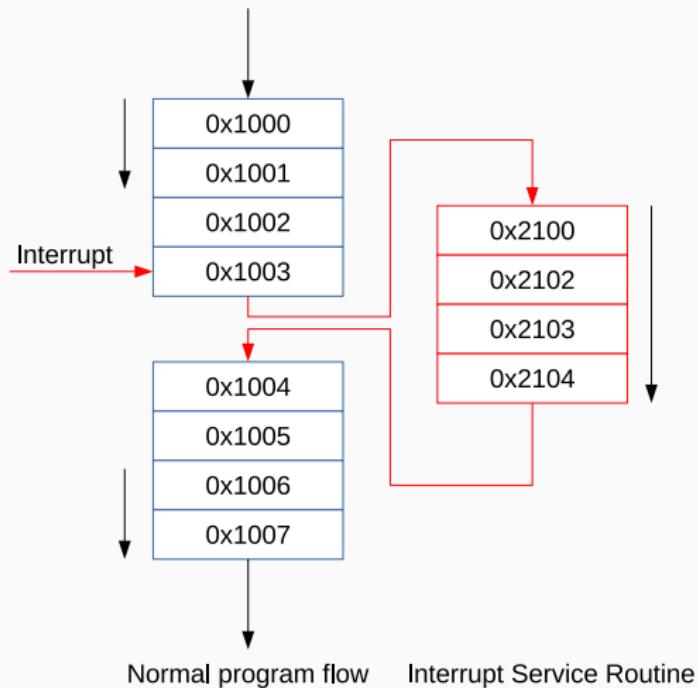
# INTERRUPTS

- Interrupts are the basis for multi-tasking
- They allow peripherals to generate **events** and alert the CPU
- Interrupts can come from multiple sources, not just timers:
  - e.g. pin change interrupt (button pressed)
  - e.g. serial port receive (character received)
- Use with caution however, as debugging interrupts can be challenging

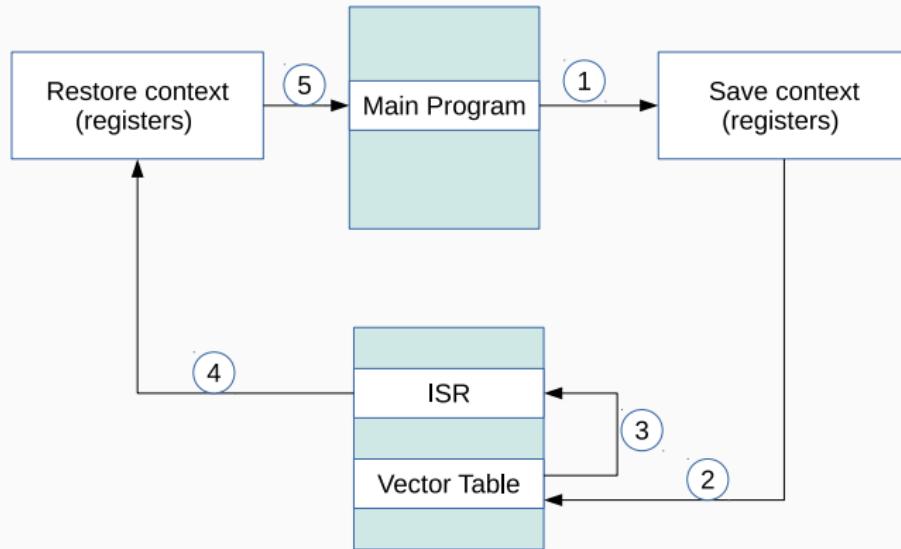
# INTERRUPT PROCESS



# INTERRUPT PROCESS



# SERVICING AN INTERRUPT



# CONTEXT SWITCHING

1. Preserve the context (registers, stack pointers and program counters)
2. Jump to the interrupt vector table
3. Vector to the ISR address
4. Restore context
5. Continue executing main program

# INTERRUPT VECTORS



## UNDER THE HOOD: OVERHEAD AND LATENCY

- Because of context switching, interrupts are not free
  - ARM Cortex M3: 12 cycles
  - PIC18F: 3 cycles
- Housekeeping (such as storing more registers for high level languages like C) can make them even more expensive
- Too frequent high priority interrupts can starve low priority routines

# WHAT COULD GO WRONG HERE?

```
// global variable which keeps current time
DateTimeType TimerVal;
// interrupt handler
void TIM1_IRQHandler(void)
{
 // code to update TimerVal
}
// Access function
void getTime(DateTimeType * dt)
{
 dt->day=TimerVal.day;
 dt->hour=TimerVal.hour;
 dt->minute=TimerVal.minute;
 dt->second=TimerVal.second;
}
```

# AND HERE?

```
// global variable to store the number of samples
uint32_t num_samples=0;
// interrupt handler
void TIM1_IRQHandler(void)
{
 //...
 push(&sample); // push into buffer
 num_samples++; // increment total number
 //...
}
// main
void main(void)
{
 //...
 if (num_samples > 16)
 {
 pop(&new_sample); // take a sample from buffer
 num_samples--; // decrement total samples
 }
 //...
}
```

## UNDER THE HOOD:

- The instruction `num_samples--`; is not necessarily **atomic**
- Atomic means that the instruction can complete without interruption (e.g. single cycle)
- This is because in many CPUs, it is necessary to perform a Read-Modify-Write (RMW) operation to alter variables

```
mov r1,[num_samples]
dec r1
mov [num_samples],r1
```

# RE-ENTRANCY

- If a function is re-entrant, then it can be interrupted in the middle of its execution and safely called again and carry on as before
- Any function that can be interrupted should be made re-entrant
- For example:

```
uint32_t count;
uint32_t counter(void)
{
 count = count + 1;
 printf("%lu\n",counter); // if printf() is called somewhere else - interleaved output
}
```

## MAKING A FUNCTION RE-ENTRANT

- Avoid shared variables (easiest way, but not always possible)
- Protect critical sections of code by disabling interrupts:

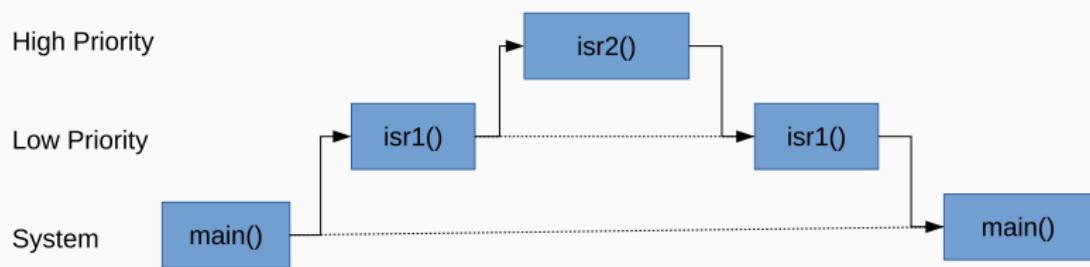
```
INTERRUPT_DISABLE();
num_samples--;
INTERRUPT_ENABLE();
```

- Disabling interrupts works, but it increases latency, which can be an issue

## MUTEXES AND SEMAPHORES

- A **mutex** is a special flag (a binary semaphore) that allows one and only one function to modify a shared variable
- mutex: mutually exclusive
- This allows a function to take a “lock” on a variable and guarantee that it is the only one modifying it

## \*NESTED INTERRUPTS



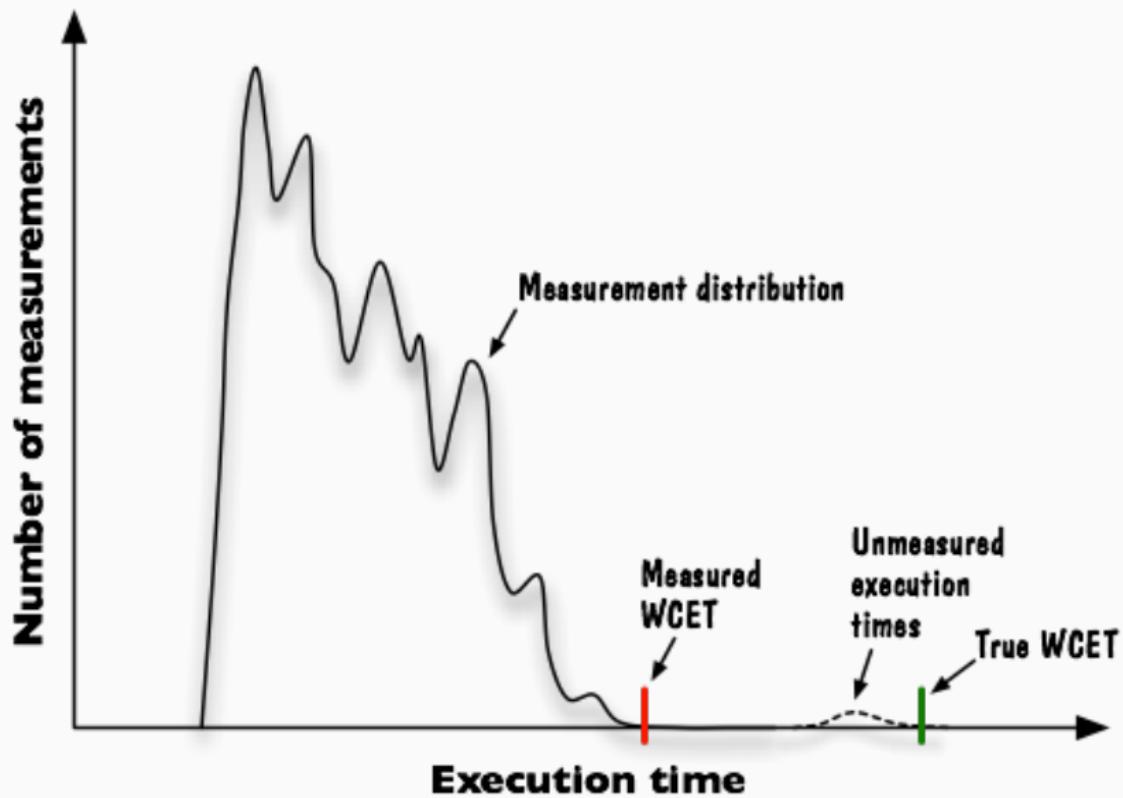
## \*NESTED INTERRUPTS

- Should be used with caution, as ISRs also need to be re-entrant safe
- This is because there is only one hardware stack frame
- Nesting does however allow for better guarantees on timing

# INSTRUMENTING AN ISR

- To measure how long an ISR takes:
  - Use formal methods
  - Run in a cycle-accurate simulator
  - Run in production hardware (toggle pin)
- Important parameter: Worst Case Execution Time (WCET)

# WORST CASE EXECUTION TIME



## OPTIMIZING AN ISR

- An ISR is one area where using inline assembler might be necessary
- Especially if ISR is triggered frequently
- E.g. ISR called at 1kHz, WCET 200us
- What is system load?
  - $\text{Load} = 1000 \times 0.0002 = 0.2 = 20\%$
- Halve ISR runtime:
  - $\text{Load} = 1000 \times 0.0001 = 0.1 = 10\%$

# MULTIPLE INTERRUPT SOURCES: INTERRUPT MAP

- Helps to dimension the system

---

| Source | WCET Max. | Rate  | Description       |
|--------|-----------|-------|-------------------|
| TMR1   | 10us      | 10kHz | System Tick Timer |
| USART0 | 100us     | 1kHz  | Serial Data In    |

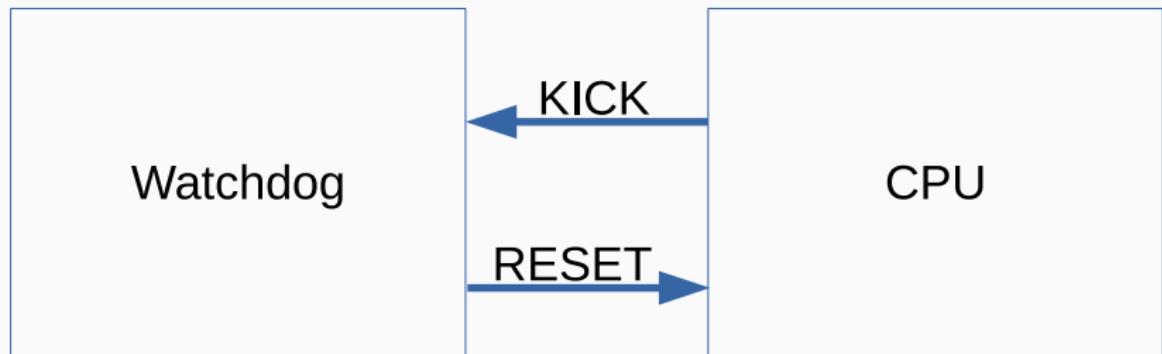
---

# GUIDELINES FOR ISRS

- Keep ISRs short
- Keep ISRs simple
- Only do what **has** to be done in the ISR
  - Set a flag or add to a queue for a normal routine to take care of
  - Keep non-critical code out of ISRs

## ★ A SPECIAL TIMER: THE WATCHDOG

- A watchdog timer is a special timer
  - Generally it is integrated with the microcontroller core
  - Sometimes it makes sense to have a physically separate watchdog chip
- It starts from some value and counts down
- If the CPU does not kick (pet) it, then it resets the CPU



## ★ USING THE WATCHDOG

- All embedded systems should use a watchdog to recover from system hangs and crashes
- Watchdogs often can't be turned off once turned on
  - This prevents runaway code from inadvertently disabling the timer
- The watchdog should only be kicked from one place to help aid debugging
- The watchdog only protects against processes taking too long, it cannot protect against algorithmic errors

## SUMMARY

- Interrupts are a powerful mechanism to enable reactive concurrency
- Keep interrupt routines short to prevent CPU blocking
- Avoid shared variables in ISRs if at all possible
- If not possible, use mutex/flags to prevent concurrent access

## 7. PERIPHERALS AND I/O

---

# OVERVIEW

- How do we interface to the real-world?
- How do we deal with signals over a continuous domain?
- How we communicate information between devices and systems?

# PERIPHERALS

- The key differentiator between different manufacturers is the variety of peripherals provided
- More peripherals means that less “glue” logic is required, which reduces the hardware complexity and BOM
- Microcontrollers have a vast array of peripherals, both analog and digital

# SAMPLE DATASHEET: PIC10F 8 BIT (£0.42)



## PIC10F200/202/204/206

### 6-Pin, 8-Bit Flash Microcontrollers

#### Devices Included In This Data Sheet:

- PIC10F200
- PIC10F204
- PIC10F202
- PIC10F206

#### High-Performance RISC CPU:

- Only 33 Single-Word Instructions to Learn
- All Single-Cycle Instructions except for Program Branches, which are Two-Cycle
- 12-Bit Wide Instructions
- 2-Level Deep Hardware Stack
- Direct, Indirect and Relative Addressing modes for Data and Instructions
- 8-Bit Wide Data Path
- Eight Special Function Hardware Registers
- Operating Speed:
  - 4 MHz internal clock
  - 1  $\mu$ s instruction cycle

#### Special Microcontroller Features:

- 4 MHz Precision Internal Oscillator:
  - Factory calibrated to  $\pm 1\%$
- In-Circuit Serial Programming™ (ICSP™)
- In-Circuit Debugging (ICD) Support
- Power-on Reset (POR)
- Device Reset Timer (DRT)
- Watchdog Timer (WDT) with Dedicated On-Chip RC Oscillator for Reliable Operation
- Programmable Code Protection
- Multiplexed MCLR Input Pin
- Internal Weak Pull-ups on I/O Pins
- Power-Saving Sleep mode
- Wake-up from Sleep on Pin Change

#### Low-Power Features/CMOS Technology:

- Operating Current:
  - < 175  $\mu$ A @ 2V, 4 MHz, typical
- Standby Current:
  - 100 nA @ 2V, typical
- Low-Power, High-Speed Flash Technology:
  - 100,000 Flash endurance
  - > 40 year retention
- Fully Static Design
- Wide Operating Voltage Range: 2.0V to 5.5V
- Wide Temperature Range:
  - Industrial: -40°C to +85°C
  - Extended: -40°C to +125°C

#### Peripheral Features (PIC10F200/202):

- Four I/O Pins:
  - Three I/O pins with individual direction control
  - One input-only pin
  - High current sink/source for direct LED drive
  - Wake-on-change
  - Weak pull-ups
- 8-Bit Real-Time Clock/Counter (TMR0) with 8-Bit Programmable Prescaler

#### Peripheral Features (PIC10F204/206):

- Four I/O Pins:
  - Three I/O pins with individual direction control
  - One input-only pin
  - High current sink/source for direct LED drive
  - Wake-on-change
  - Weak pull-ups
- 8-Bit Real-Time Clock/Counter (TMR0) with 8-Bit Programmable Prescaler
- One Comparator:
  - Internal absolute voltage reference
  - Both comparator inputs visible externally
  - Comparator output visible externally

TABLE 1: PIC10F20X MEMORY AND FEATURES

| Device    | Program Memory | Data Memory  | I/O | Timers<br>8-bit | Comparator |
|-----------|----------------|--------------|-----|-----------------|------------|
|           | Flash (words)  | SRAM (bytes) |     |                 |            |
| PIC10F200 | 256            | 16           | 4   | 1               | 0          |
| PIC10F202 | 512            | 24           | 4   | 1               | 0          |

# SAMPLE DATASHEET: PIC24FJ 16 BIT (£2.70)



MICROCHIP

## PIC24FJ128GA310 FAMILY

### 64/80/100-Pin, General Purpose, 16-Bit Flash Microcontrollers with LCD Controller and XLP Technology

#### Extreme Low-Power Features:

- Multiple Power Management Options for Extreme Power Reduction:
- VBAT allows the device to transition to a backup battery for the lowest power consumption with RTCC
- Deep Sleep allows near total power-down with the ability to wake-up on external triggers
- Sleep and Idle modes selectively shut down peripherals and/or core for substantial power reduction and fast wake-up
- Doze mode allows CPU to run at a lower clock speed than peripherals
- Alternate Clock modes Allow On-the-Fly Switching to a Lower Clock Speed for Selective Power Reduction
- Extreme Low-Power Current Consumption for Deep Sleep:
  - WDT: 270 nA @ 3.3V typical
  - RTCC: 400 nA @ 32 kHz, 3.3V typical
  - Deep Sleep current, 40 nA, 3.3V typical

#### Peripheral Features:

- LCD Display Controller:
  - Up to 60 segments by 8 commons
  - Internal charge pump and low-power, internal resistor biasing
  - Operation in Sleep mode
- Up to Five External Interrupt Sources
- Peripheral Pin Select (PPS): Allows Independent I/O Mapping of Many Peripherals
- Five 16-Bit Timers/Counters with Prescaler:
  - Can be paired as 32-bit timers/counters
- Six-Channel DMA supports All Peripheral modules:
  - Minimizes CPU overhead and increases data throughput

#### Peripheral Features (Continued):

- Seven Input Capture modules, each with a Dedicated 16-Bit Timer
- Seven Output Compare/PWM modules, each with a Dedicated 16-Bit Timer
- Enhanced Parallel Master/Slave Port (EPM/PESP):
  - Runs in Deep Sleep and VBAT modes
- Two 3-Wire/4-Wire SPI modules (support 4 Frame modes) with 8-Level FIFO Buffer
- Two I<sup>2</sup>C™ modules Support Multi-Master/Slave mode and 7-Bit/10-Bit Addressing
- Four UART modules:
  - Support RS-485, RS-232 and LIN/J2602
  - On-chip hardware encoder/decoder for IrDA®
  - Auto-wake-up on Auto-Baud Detect
  - 4-level deep FIFO buffer
- Programmable 32-Bit Cyclic Redundancy Check (CRC) Generator
- Digital Signal Modulator Provides On-Chip FSK and PSK Modulation for a Digital Signal Stream
- Configurable Open-Drain Outputs on Digital I/O Pins
- High-Current Sink/Source (18 mA/18 mA) on All I/O Pins

#### Analog Features:

- 10/12-Bit, 24-Channel Analog-to-Digital (A/D) Converter:
  - Conversion rate of 500 kops (10-bit), 200 kops (12-bit)
  - Conversion available during Sleep and Idle
- Three Rail-to-Rail Enhanced Analog Comparators with Programmable Input/Output Configuration
- On-Chip Programmable Voltage Reference
- Charge Time Measurement Unit (CTMU):
  - Used for capacitive touch sensing, up to 24 channels
  - Time measurement down to 1 ns resolution
  - CTMU temperature sensing

| Device          | Pins | Memory                |                   | Remappable Peripherals |   |               |                    |             |     | I <sup>2</sup> C™ | 10/12-Bit ADC (16 bit) | EPROM/EPPS | LCD (pixels) | JTAG | Deep Sleep w/VBAT |   |
|-----------------|------|-----------------------|-------------------|------------------------|---|---------------|--------------------|-------------|-----|-------------------|------------------------|------------|--------------|------|-------------------|---|
|                 |      | Flash Program (bytes) | Data SRAM (bytes) | 16-Bit Timers          |   | Capture Input | Compare/PWM Output | UART/wIrDA® | SPI |                   |                        |            |              |      |                   |   |
| PIC24FJ128GA310 | 100  | 128K                  | 8K                | 5                      | 7 | 4             | 2                  | 2           | 24  | 3                 | 24                     | Y          | 480          | Y    | Y                 |   |
| PIC24FJ128GA308 | 80   | 128K                  | 8K                | 5                      | 7 | 4             | 2                  | 2           | 16  | 3                 | 16                     | Y          | 368          | Y    | Y                 |   |
| PIC24FJ128GA306 | 64   | 128K                  | 8K                | 5                      | 7 | 7             | 4                  | 2           | 2   | 16                | 3                      | 16         | Y            | 240  | Y                 | Y |
| PIC24FJ64GA310  | 100  | 64K                   | 8K                | 5                      | 7 | 4             | 2                  | 2           | 24  | 3                 | 24                     | Y          | 480          | Y    | Y                 |   |
| PIC24FJ64GA308  | 80   | 64K                   | 8K                | 5                      | 7 | 4             | 2                  | 2           | 16  | 3                 | 16                     | Y          | 368          | Y    | Y                 |   |

# SAMPLE DATASHEET: STM32F407 32 BIT (£7.51)



**STM32F405xx**  
**STM32F407xx**

ARM Cortex-M4 32b MCU+FPU, 210DMIPS, up to 1MB Flash/192+4KB RAM, USB OTG HS/FS, Ethernet, 17 TIMs, 3 ADCs, 15 comm. interfaces & camera

Datasheet - production data



## Features

- Core: ARM 32-bit Cortex™-M4 CPU with FPU, Adaptive real-time accelerator (ART Accelerator™) allowing 0-wait state execution from Flash memory, frequency up to 168 MHz, memory protection unit, 210 DMIPS/1.25 DMIPS/MHz (Dhrystone 2.1), and DSP instructions
- Memories
  - Up to 1 Mbyte of Flash memory
  - Up to 192+4 Kbytes of SRAM including 64-Kbyte of CCM (core coupled memory) data RAM
  - Flexible static memory controller supporting Compact Flash, SRAM, PSRAM, NOR and NAND memories
- LCD parallel interface, 8080/6800 modes
- Clock, reset and supply management
  - 1.8 V to 3.6 V application supply and I/Os
  - POR, PDR, PVD and BOR
  - 4-to-26 MHz crystal oscillator
  - Internal 16 MHz factory-trimmed RC (1% accuracy)
  - 32 kHz oscillator for RTC with calibration
  - Internal 32 kHz RC with calibration
- Low power
  - Sleep, Stop and Standby modes
  - V<sub>BAT</sub> supply for RTC, 20×32 bit backup registers + optional 4 KB backup SRAM
- 3×12-bit, 2.4 MSPS A/D converters: up to 24 channels and 7.2 MSPS in triple interleaved mode
- 2×12-bit D/A converters
- General-purpose DMA: 16-stream DMA controller with FIFOs and burst support
- Up to 17 timers, up to twelve 16-bit and two 32-bit timers

IC/OC/PWM or pulse counter and quadrature (incremental) encoder input

- Debug mode
  - Serial wire debug (SWD) & JTAG interfaces
  - Cortex-M4 Embedded Trace Macrocell™
- Up to 140 I/O ports with interrupt capability
  - Up to 136 fast I/Os up to 84 MHz
  - Up to 138 5 V-tolerant I/Os
- Up to 15 communication interfaces
  - Up to 3 I<sup>2</sup>C interfaces (SMBus/PMBus)
  - Up to 4 USARTs/2 UARTs (10.5 Mbit/s, ISO 7816 Interface, LIN, IrDA, modem control)
  - Up to 3 SPIs (42 Mbit/s), 2 with muxed full-duplex I<sup>2</sup>S to achieve audio class accuracy via internal audio PLL or external clock
  - 2 × CAN interfaces (2.0B Active)
  - SDIO interface
- Advanced connectivity
  - USB 2.0 full-speed device/host/OTG controller with on-chip PHY
  - USB 2.0 high-speed/full-speed device/host/OTG controller with dedicated DMA, on-chip full-speed PHY and ULP/I
  - 10/100 Ethernet MAC with dedicated DMA: supports IEEE 1588v2 hardware, MII/RMII
- 8- to 14-bit parallel camera interface up to 54 Mbytes/s
- True random number generator
- CRC calculation unit
- 96-bit unique ID
- RTC: subsecond accuracy, hardware calendar

Table 1. Device summary

| Reference   | Part number                                                                     |
|-------------|---------------------------------------------------------------------------------|
| STM32F405xx | STM32F405RG, STM32F405VG, STM32F405ZG,<br>STM32F405OG, STM32F405OE              |
| STM32F407xx | STM32F407VG, STM32F407IG, STM32F407ZG,<br>STM32F407VE, STM32F407ZE, STM32F407IE |

# DIGITAL INPUT/OUTPUT (GPIO)

- Microcontrollers have many digital pins that can be configured as input or output
- GPIO: General Purpose Input Output
- These are typically mapped to registers (ports) in memory according to CPU width
- E.g. 8 bit microcontroller will have:
  - Ports PORTA, PORTB, PORTC
  - Pins PA0...PA7, PB0...PB7 etc
- They have a configuration register which sets whether each pin will be input or output
- On power up, pins typically default to inputs for safety

## ACCESSING A PORT

- A port can be written to like any other register in memory
- The port will reflect or shadow the value written to it
  - e.g. PORTB = 0x01 will set pin B0 to 1
- To access individual pins, we use a Read-Modify-Write sequence (assembler)

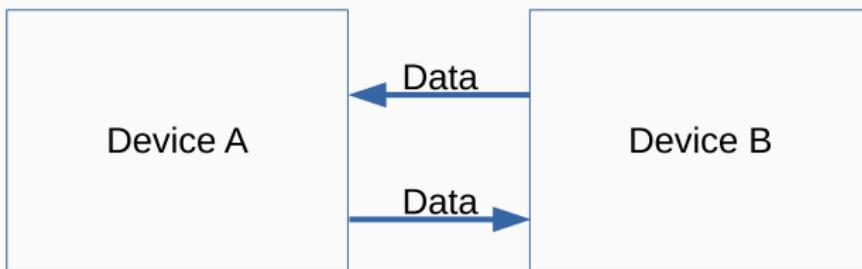
```
mov R1, PORTB // read
bset R1, 0 // modify - set pin B0
mov PORTB, R1 // write
```

## EXAMPLES

- Output
  - Turn on and off a motor in a lift
  - Flash an LED to show an error
- Input
  - Check whether a button has been pressed
  - Turn on the car's ignition

# DIGITAL COMMUNICATION

- Digital devices need to communicate data
- Large number of techniques, but some are standardised

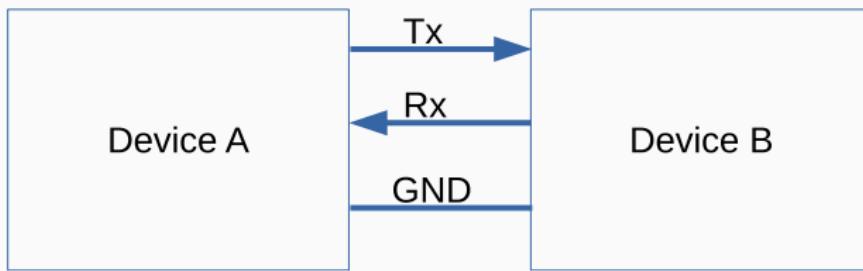


# DIGITAL COMMUNICATIONS: CLASSIFICATION

- Directionality:
  - Simplex: Unidirectional
  - Half-Duplex: Bidirectional, but only in one direction at a time
  - Full-Duplex: Bidirectional, simultaneous
- Synchronicity
  - Synchronous: Tx and Rx share a common clock
  - Asynchronous: Tx and Rx have independent clocks
- Width
  - Serial: Bits are sent one after the other
  - Parallel: Bits are sent simultaneously

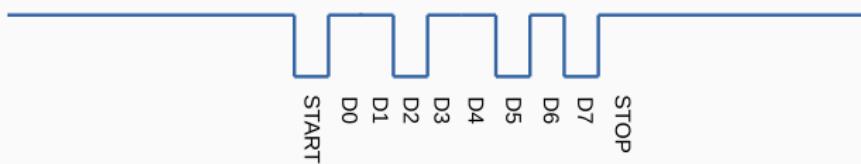
# DIGITAL COMMUNICATIONS: USART

- USART: Universal Serial Asynchronous Receiver Transmitter
- One of the oldest protocols for digital communication



# DIGITAL COMMUNICATIONS: USART

- Asynchronous, serial protocol
- Optional parity bit for simple checksum



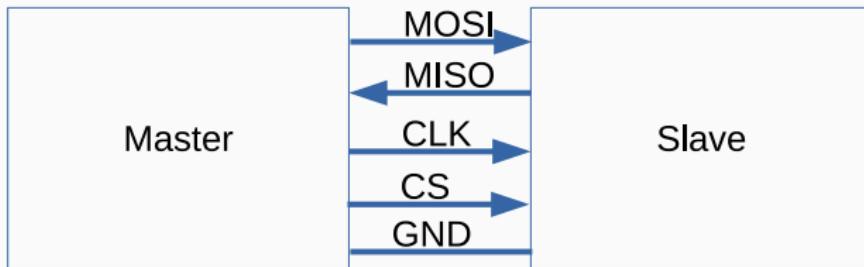
- Common baud rates:
  - 9600
  - 19200
  - 57600
  - 115200

## DIGITAL COMMUNICATIONS: USART

- USART is widely supported on many devices
- USART is not ideal for low power devices, as they need to perform precise timing which prevents a device from going to sleep
- Both devices must also agree on the protocol beforehand, although there are some techniques for autobauding
- Timing needs to be relatively precise to prevent bit errors

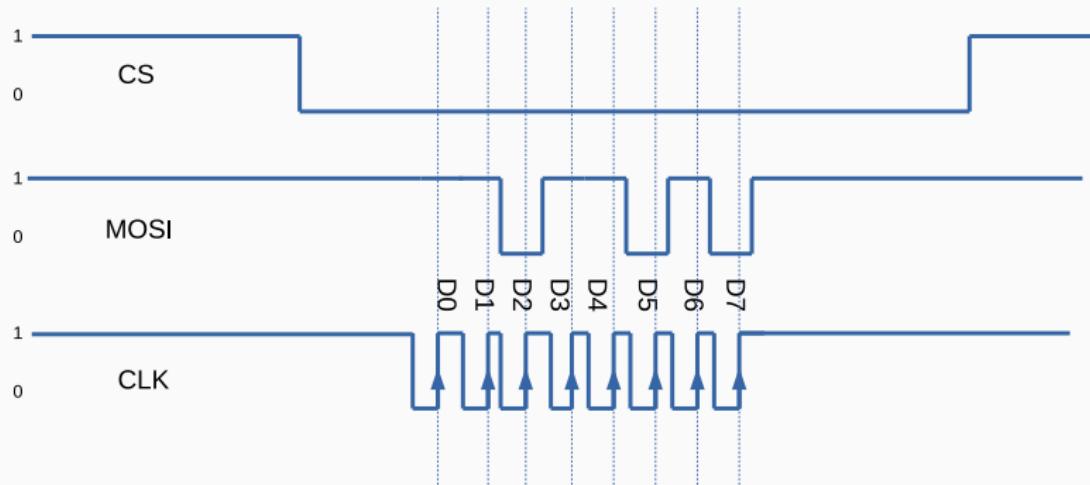
# DIGITAL COMMUNICATIONS: SPI

- SPI: Serial Peripheral Interface (3-wire)
- Widely used as bus between integrated circuits
- Wires:
  - CS: Chip Select (pull low to talk to slave)
  - MOSI: Master-Out Slave-In
  - MISO: Master-In Slave-Out
  - CLK: Clock



# DIGITAL COMMUNICATIONS: SPI

- Synchronous, serial protocol
- Full duplex, master initiated
- Master drives the clock at an arbitrary rate
- Data is valid on rising edge of clock

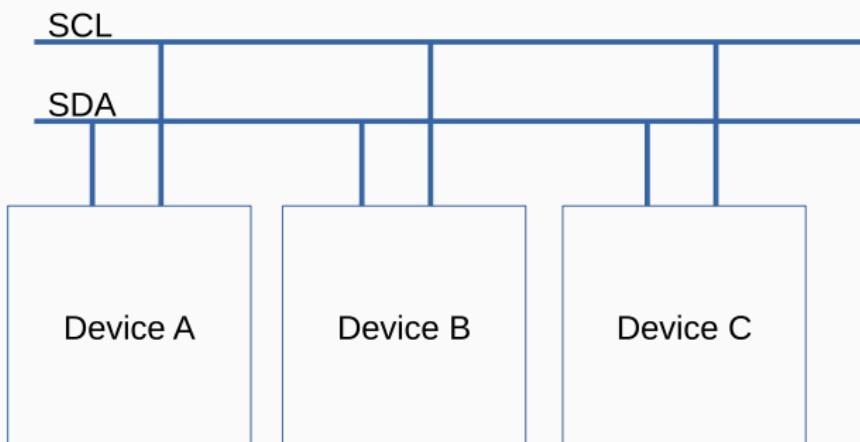


## DIGITAL COMMUNICATIONS: SPI

- SPI is used as a communication protocol between integrated circuits on a board
- Clock can be stretched or even paused mid-transaction
- Simple protocol
- One master can address multiple slaves, each with a dedicated CS line
- Needs at least three signal wires (CS is optional)

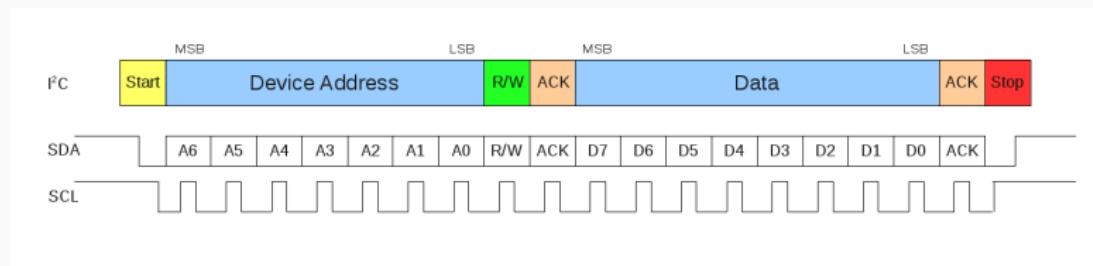
# DIGITAL COMMUNICATIONS: I<sup>2</sup>C

- I<sup>2</sup>C: Inter-IC communication (2 wire)
- Allows any device to become a master
- Each device has a unique address on the bus
- Wires:
  - SDA: Serial Data
  - SCL: Serial Clock



# DIGITAL COMMUNICATIONS: I<sup>2</sup>C

- Synchronous, serial protocol
- Half duplex, master initiated
- Devices contend the bus to become master



# DIGITAL COMMUNICATIONS: I<sup>2</sup>C

- I<sup>2</sup>C<sup>1</sup> is used as a communication protocol between integrated circuits on a board
- Clock can be stretched or even paused mid-transaction
- Complex, bidirectional protocol
- Only two signal wires needed, regardless of number of devices
- Bus speeds are limited to 400kHz

---

<sup>1</sup>I<sup>3</sup>C is a revised, faster variant

## DIGITAL COMMUNICATIONS: OTHERS

- Other widely used interfaces include:
  - CANbus: Mainly used in vehicles
  - USB: Complex, high bandwidth protocol
  - I<sup>2</sup>S: Specialized variant of I<sup>2</sup>C for low jitter audio
  - 1-Wire: Single-signal wire, bidirectional interface
  - SDIO: Interface for SD memory cards

## ANALOG INTERFACE

- Most of the information in the world is not nicely quantized, but continuous
  - Temperature
  - Acceleration
  - Pressure
  - Force
- We need interfaces that can convert
  - Digital to Analog (DAC)
  - Analog to Digital (ADC)

# DIGITAL TO ANALOG CONVERTER (DAC)

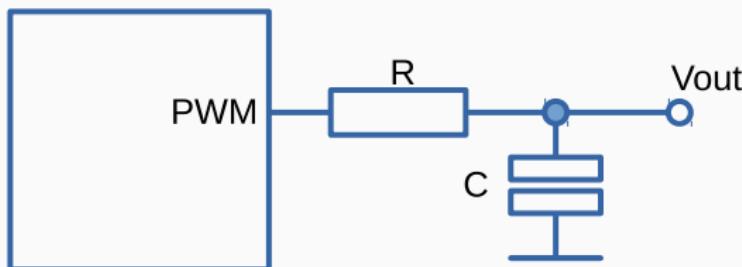
- A DAC takes in a digital word and converts it to an analog voltage
- Important Parameters:
  - Resolution: Number of unique output voltages e.g. 256 for 8 bit DAC
  - Sampling Rate: Maximum rate of output change
  - Non-linearity: How much the output deviates from a straight line
  - Noise: How much jitter there is on the output voltage

## DAC: PWM

- One of the simplest forms of DAC is a Pulse Width Modulator
- This is widely used for motor control and as a low-cost DAC
- Many microcontrollers have built in PWM modules
- Simple principle: vary the on-time (duty cycle) of a constant frequency clock

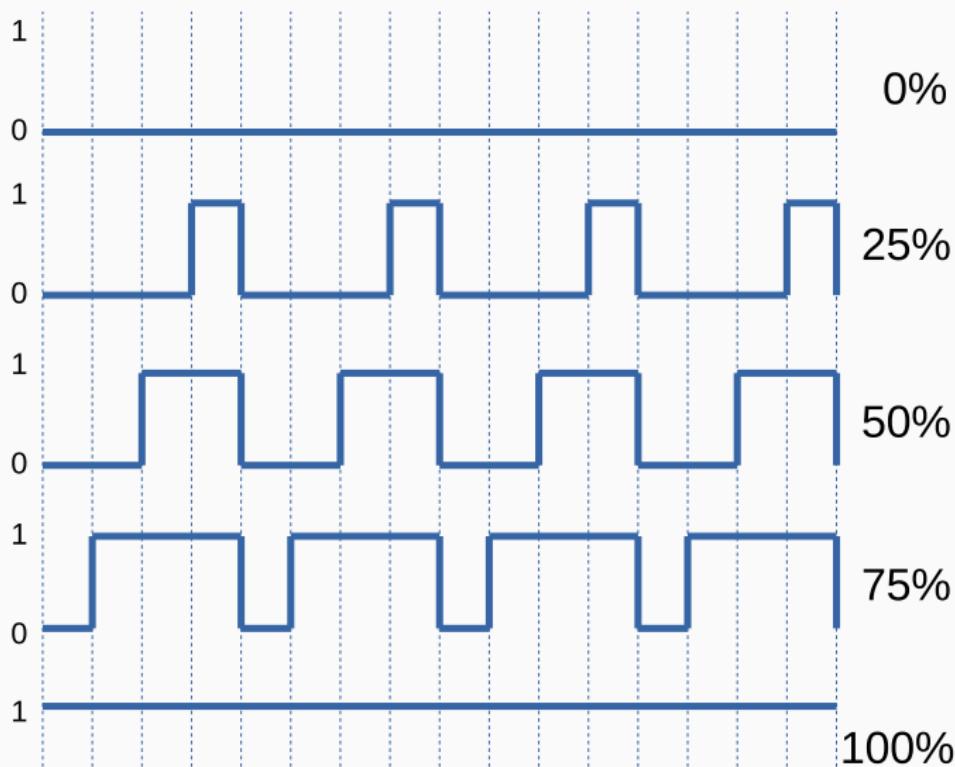
## DAC: PWM

- A low-pass filter is used to remove the switching harmonics and retain the low-frequency average



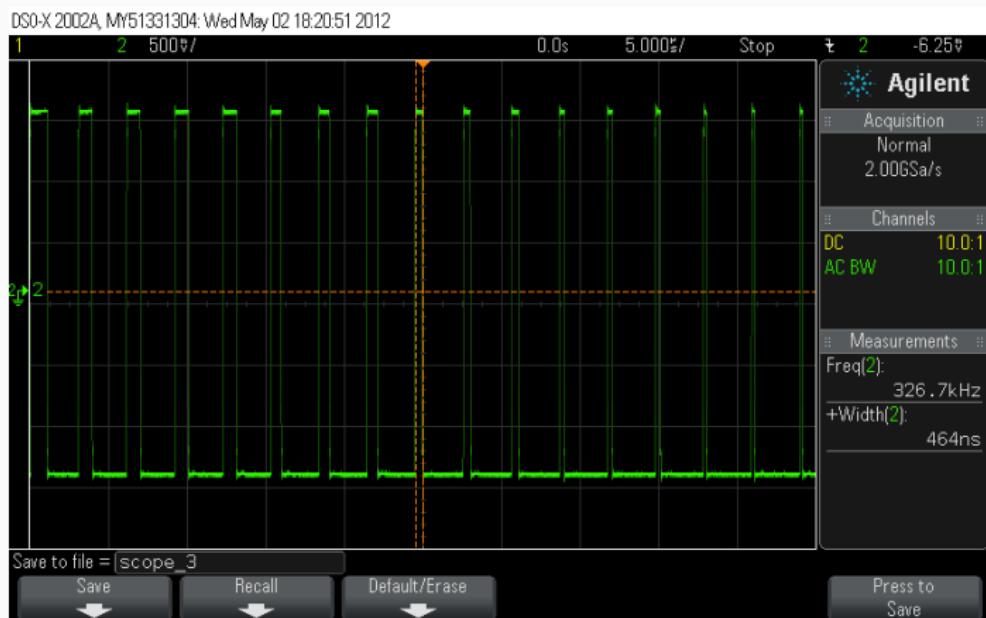
## DAC: PWM

- Varying the pulse width alters the voltage



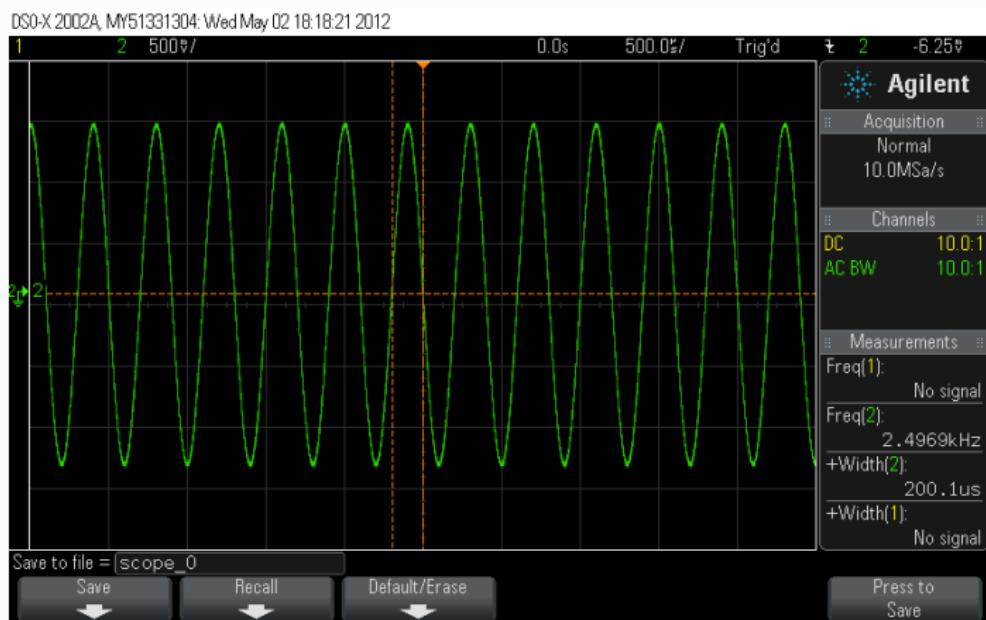
# DAC: PWM

- Pulse train



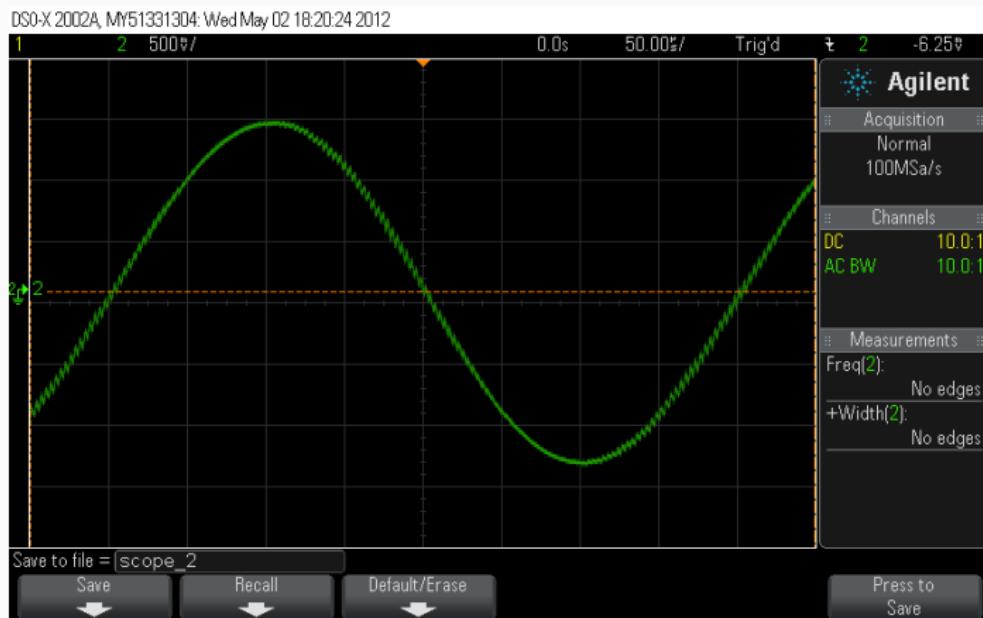
# DAC: PWM

- After being filtered



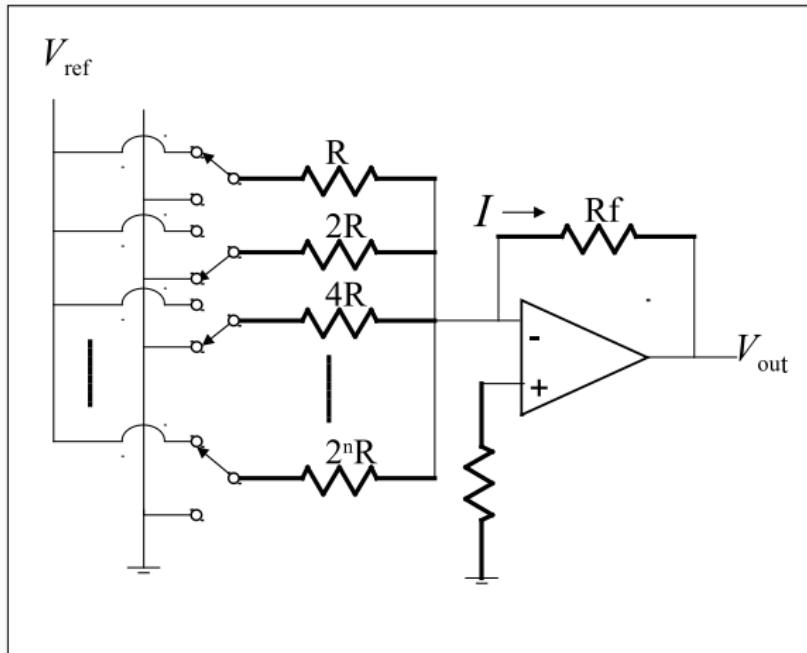
## DAC: PWM

- High frequency hash noise



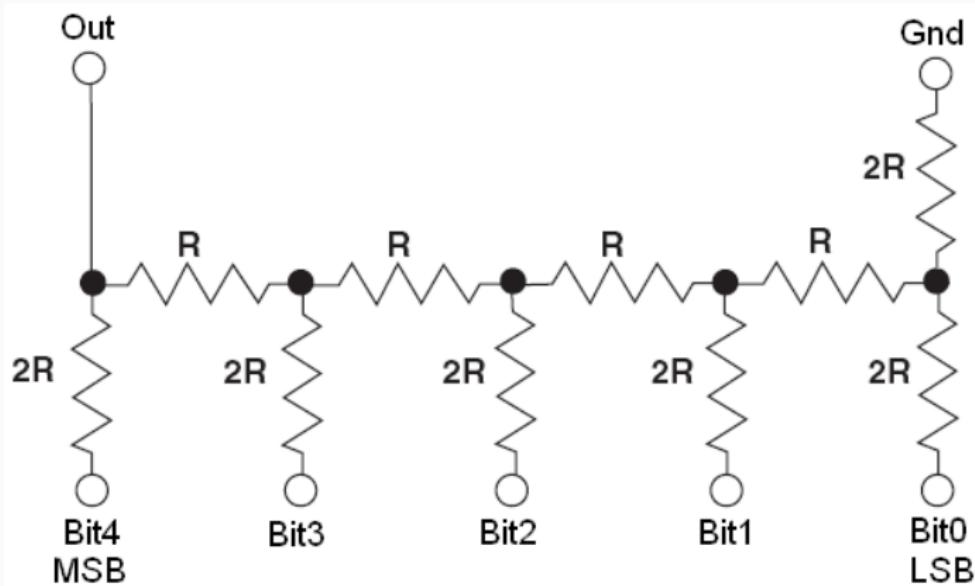
## DAC: RESISTOR LADDER

- Alternative is to weight digital signals differently (linear combination)



## DAC: R2R LADDER

- For high resolutions, weighted resistor ladder becomes problematic
- Alternative approach just using two resistor values:  
R2R ladder



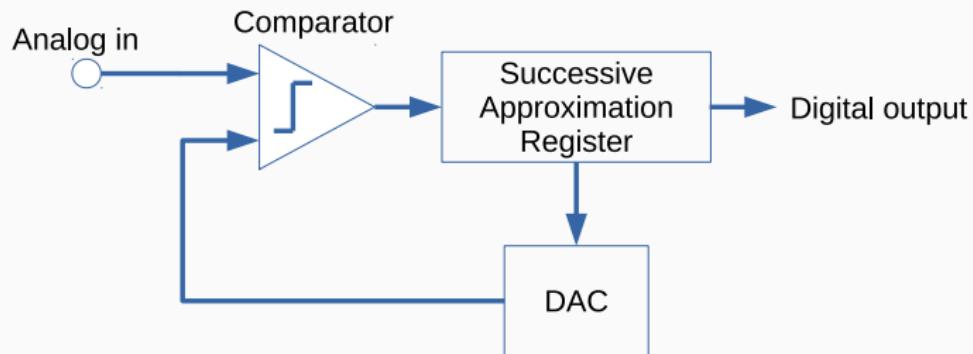
## DAC: OTHER VARIANTS

- Sigma-delta: rather than pulse width, integrates pulse-density (number of pulses per unit time)
  - Widely used in audio as a 1-bit oversampling DAC
- Segmented DAC:
  - Combines two different types of DAC for high and low bits to optimize performance

## ANALOG TO DIGITAL CONVERTER (ADC)

- To convert an analog signal to digital is a little harder.
- However, we can exploit the fact that we can generate a variable analog voltage using a DAC
- Compare the generated voltage to the input voltage
  - Too high: decrease the digital word
  - Too low: increase the digital word
- We can find the point where it toggles - this will be the digital representation
- Called a Successive Approximation Register (SAR) ADC

# SAR ADC



## ADC: OTHER VARIANTS

- Flash converter: instead of one comparator, have comparators for every digital word combination
  - Extremely fast
  - Limited resolution (8 bit = 256 comparators and reference voltages)
- Switched capacitor ADC
  - Based on measuring how long it takes to charge a capacitor
  - Variable conversion time
- Tracking ADC
  - Follows analog waveform with DAC
  - Counts the number of up or down pulses needed to keep track of input waveform

## OTHER PERIPHERALS

- Charge Time Measurement Unit (CTMU): Used for capacitive sensing
- Random Number Generator: Used for security applications
- Input Capture: Measure input pulse width
- Analog Comparator: Determine whether an input voltage is above a threshold
- Brown-out detector: Resets processor if voltage dips
- LCD driver: generates voltages and signals for LCD displays

## \*SENSORS

- To measure parameters in the real world (temperature, acceleration) we need a **transducer** which converts that measurand:
  - To a voltage - digitize with ADC
  - To a digital signal - interface directly to microcontroller
- Sensors have a wide range of parameters
  - Noise
  - Response time
  - Accuracy
  - Precision
  - Repeatability
  - Linearity

## \*ACTUATORS

- To make changes in the real world, we use an **actuator**.
- Some actuators can be driven directly from a microcontroller
  - LED
  - Buzzer
- However, most actuators need some sort of high-current buffer
  - Motors
  - Solenoids
  - Relays

## \*CONNECTIVITY

- To interface to the wider world, we use some form of communication
- This can be wired (e.g. USART)
- Wireless is becoming more dominant
  - GSM/3G/LTE
  - Bluetooth/BLE (iBeacon)
  - WiFi
  - LoRa/SigFox/NB-IOT (IoT standards)
- We typically talk to a wireless peripheral using SPI or I<sup>2</sup>C

## SUMMARY

- Microcontrollers are packed full of peripherals
- Their purpose is to allow the CPU to sense and control the external environment
- Digital Interfaces (SPI/I<sup>2</sup>C) allow a microcontroller to communicate with other devices
- The analog domain is more challenging, but ADCs and DACs exist to bridge the divide

## FURTHER READING

- “The Art of Electronics” - Horowitz and Hill
- “Making Embedded Systems” - White
- “Embedded Systems Hardware for Software Engineers”  
- Lipiansky
- “Practical Electronics for Inventors, Third Edition” - Scherz

## 8. REACTIVE EMBEDDED SYSTEMS

---

# OVERVIEW

- How do we actually make embedded systems work?
- How do we deal with (asynchronous) real-time?
- How do we simplify the programming model?

# SUPER-LOOPS

A “super-loop” model is suited to simple, sequential program flow

```
// ****
// ** Smoke Detector **
// ****
while (1)
{
 smoke_level = check_sensor();
 if (smoke_level > THRESHOLD)
 {
 alarm_on();
 }
 else
 {
 alarm_off();
 }
 sleep(1000); // sleep for 1s
}
```

## SUPER-LOOPS

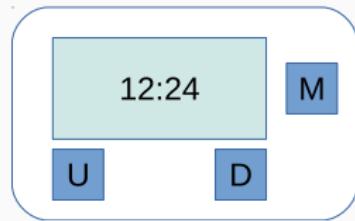
- Efficient for trivial flow
- As more branches are required, it becomes cumbersome
- If your model is simple enough to allow it, use a super-loop!

# FINITE STATE MACHINE

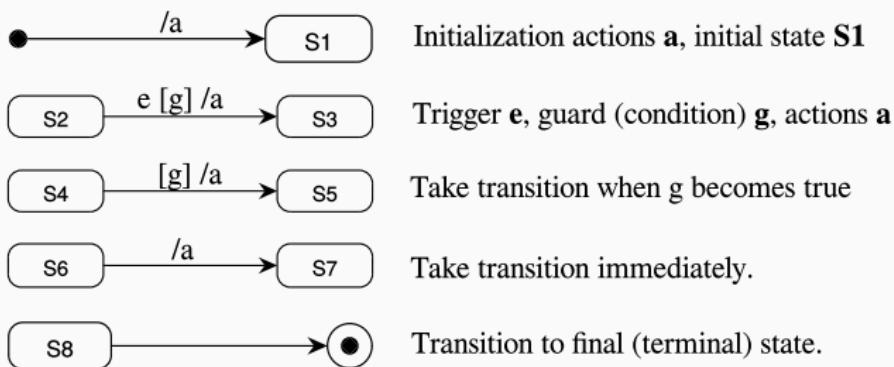
- The FSM model is well suited to embedded systems, as a system is:
  - in a particular state (red light)
  - remains in the state while waiting for an event (car sensor)
  - to transition to a new state (green light)
- FSM models are often continuous loops i.e. there is not an end to the computation
- However, they need an initial state to specified

## EXAMPLE: CHANGE TIME AND ALARM ON A WATCH

- Watch has three buttons:
  - Mode
  - Up
  - Down
- Mode toggles between displaying
  - Current time
  - Alarm time
- Up and Down increment and decrement the shown time respectively

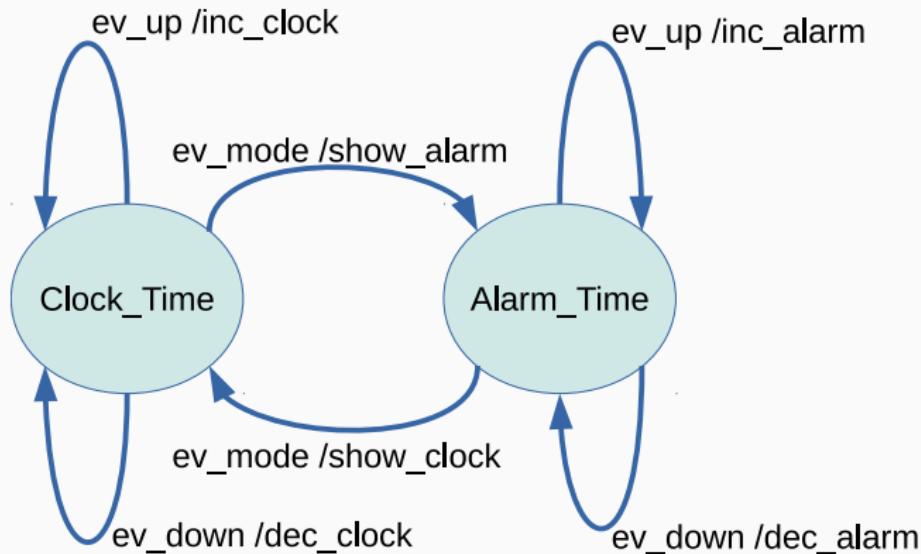


# UML REPRESENTATION (GRAPHICAL)



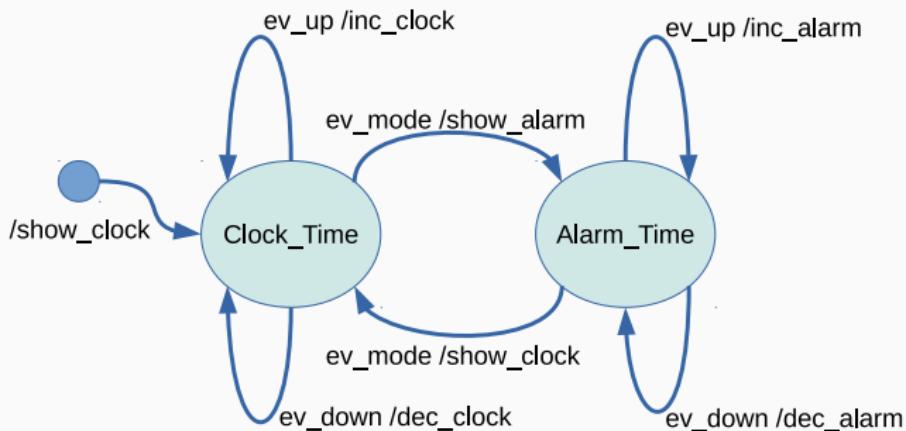
## EXAMPLE: GRAPHICAL REPRESENTATION

- What is missing?



## EXAMPLE: GRAPHICAL REPRESENTATION

- Initial state



## REPRESENTATION: STATE TRANSITION TABLE (STT)

| State      | Event   | Action     | Next State |
|------------|---------|------------|------------|
| CLOCK_TIME | EV_UP   | inc_clock  | CLOCK_TIME |
| CLOCK_TIME | EV_DOWN | dec_clock  | CLOCK_TIME |
| CLOCK_TIME | EV_MODE | show_alarm | ALARM_TIME |
| ALARM_TIME | EV_UP   | inc_clock  | ALARM_TIME |
| ALARM_TIME | EV_DOWN | dec_clock  | ALARM_TIME |
| ALARM_TIME | EV_MODE | show_clock | CLOCK_TIME |
| INIT       | NIL     | show_clock | CLOCK_TIME |

# IMPLEMENTATION: SWITCH CASE

```
// Dictionary of states
typedef enum{
 ST_CLOCK_STATE;
 ST_ALARM_STATE;
} state;
// Dictionary of events
typedef enum{
 EV_UP;
 EV_DOWN;
 EV_MODE;
} event;
```

# IMPLEMENTATION: SWITCH CASE

```
switch(current_state)
{
 case ST_CLOCK_STATE:
 switch(ev)
 {
 EV_MODE:
 show_alarm();
 new_state = ST_ALARM_STATE;
 break;
 EV_UP:
 inc_clock();
 break;
 EV_DOWN:
 dec_clock();
 break;
 DEFAULT:
 error(); // should never be here - flag an error
 }
 // more code for ST_ALARM_STATE and default
}
```

## AN ALTERNATIVE IMPLEMENTATION

- Switch-case is a pretty messy construction technique - soon it becomes too big to handle
- One alternative is to code the table as a set of actions to take given an event
  - This then simply turns into a lookup
  - Current state on one axis, Current event on other axis
  - Table can be easily generated and maintained in one place
- This is sometimes known as the State pattern

```
stateMatrix stateTable[2][3] = {
 { {ST_CLOCK,inc_clock}, {ST_CLOCK,dec_clock}, {ST_ALARM,show_alarm} },
 { {ST_ALARM,inc_alarm}, {ST_ALARM,dec_alarm}, {ST_CLOCK,show_clock} }
};
```

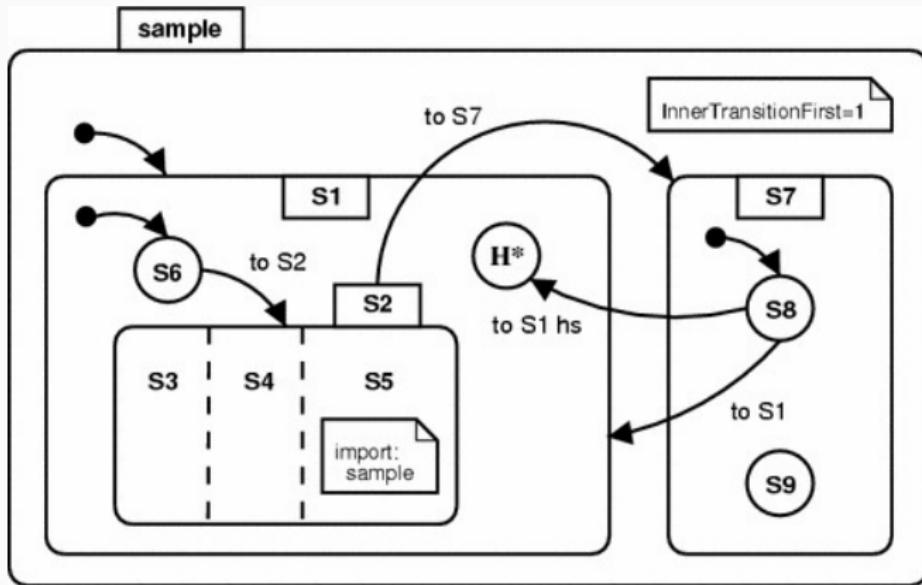
## FSMS: SUMMARY

- FSMs provide a formalism to specifying program operation in reactive systems
- The next action to take depends on:
  - current state
  - incoming event
- FSMs however do not scale to large, complex systems
  - combinatorial state explosion
  - many states react to the same event

# HIERARCHICAL STATE MACHINES

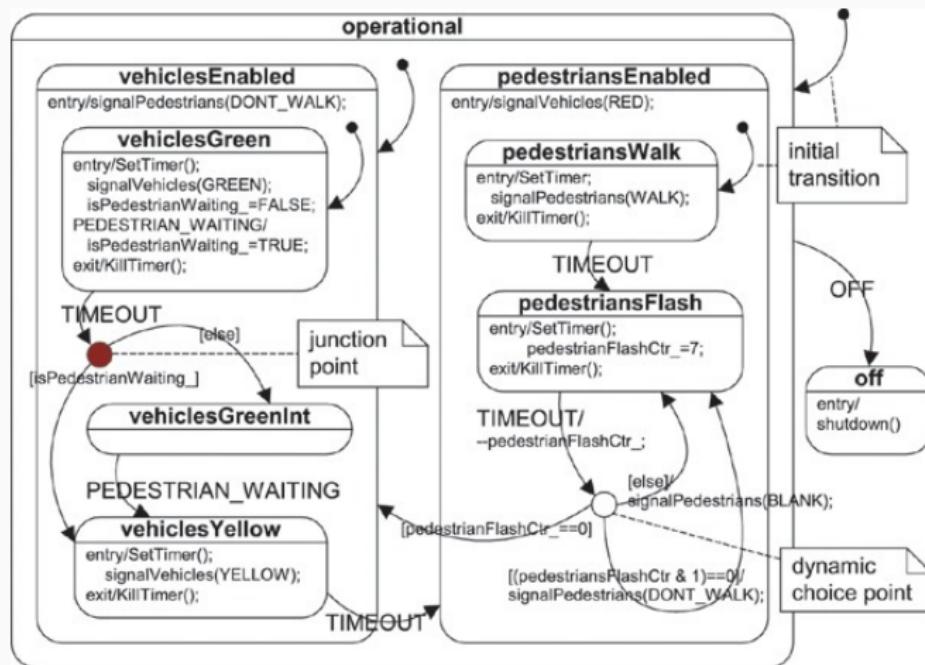
- Hierarchical State Machines tackle the problem of state explosion
- Introduce hierarchically nested states
  - Superstates
  - Substates
- Parallel flow
  - Branching
  - Joining
- History
  - Memory of prior internal state
- Formalized as StateCharts (UML notation)
  - Invented by Professor David Harel in the 1980's

# STATECHARTS



## STATECHART EXAMPLE

- Pelican Crossing<sup>1</sup>



<sup>1</sup><http://www.barrgroup.com/Embedded-Systems/How-To/Introduction-Hierarchical-State-Machines>

## HIERARCHICAL STATE MACHINES: SUMMARY

- HSMs extend FSMs to allow for constructs like parallelism and inheritance
- Ease the design process
- However, there are some subtleties about nesting of events and handlers
- Many tools are available that automate the translation of the UML model into C code

# OPERATING SYSTEMS

- Operating Systems provide a layer of abstraction to:
  - ease programming in complex systems
  - provide the illusion of concurrency
  - provide modularity and isolation
- In embedded systems the emphasis is not on fairness, but on timeliness

# REAL-TIME OPERATING SYSTEM

- Typically, most operating systems used in embedded systems are designed for real-time
- **RTOS:** Real-Time Operating System
- Many different vendors, implementations, requirements and costs (£0 upwards)
  - freeRTOS
  - Micrium uCOS
  - Keil RTX
  - RTLinux
  - TinyOS, Contiki, RIOT (specially designed for IoT)

## DIMENSIONS OF AN RTOS

- Criticality: is this a safety critical system?
  - ABS brakes vs electronic toy
- Functionality: is this a resource-constrained system?
  - spaceshuttle vs pacemaker
- Power: is this an energy-limited system?
  - microwave oven vs fitness monitor

## REAL-TIME KERNELS

- The **kernel** is software that manages the time and resources of the CPU
- Work is split into **tasks**, each responsible for a proportion of the total
- A task or thread is a simple program which thinks it has the CPU to itself
- On a single CPU, only one task can execute at a time
- Multi-tasking is the process of scheduling and switching between tasks

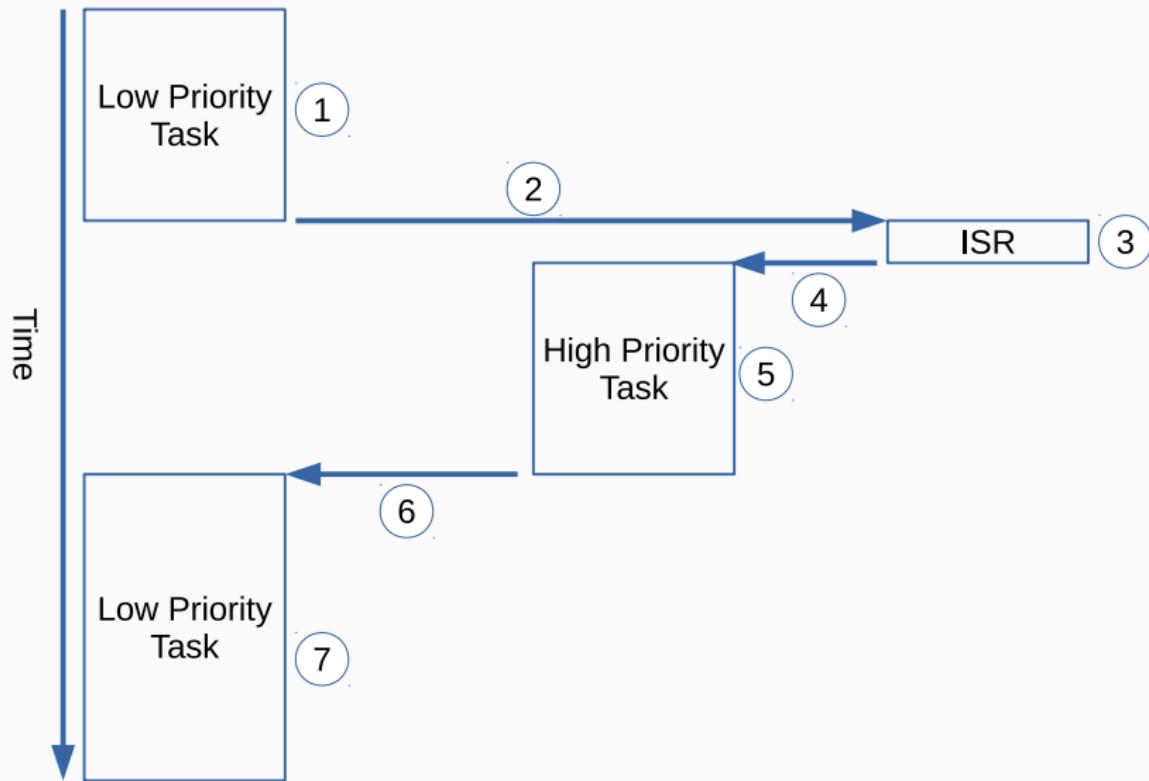
# BUILDING BLOCKS OF REAL-TIME

- Events
  - An event is a message or semaphore indicating that something has happened
- Tasks (Action)
  - A work-package

## PRE-EMPTION

- Most kernels are **pre-emptive**
- More important tasks are run first, possibly suspending a less important task
- ISRs tell the kernel that a particular event has occurred
- The kernel can then switch out the low priority task
- Typically low values have a higher priority

# PRE-EMPTION ILLUSTRATION



## PRE-EMPTION

1. A low priority task is running
2. An interrupt occurs and is serviced in hardware by an ISR
3. The ISR signals that the event that a high-priority task is waiting for happened
4. The kernel suspends the low-priority task and starts the high-priority task
5. The high-priority task completes its processing. It then will wait for a new event
6. The kernel is notified that the high priority task is waiting, so will switch to the low priority one
7. The low-priority task resumes exactly at the point where it was interrupted

# CONTEXT SWITCHING

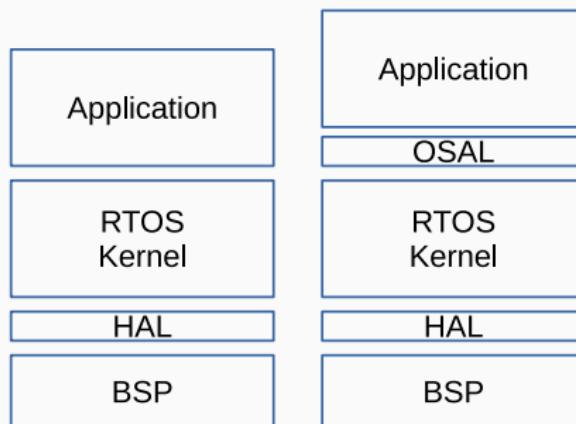
- When a function is running, it will set CPU registers to values depending on its state
- It can be suspended at any time
- When this happens, its context (state) needs to be preserved
- Unlike an ISR, context switching is not provided by hardware, but by software

```
// Task suspend - store context
push(r0)
push(r2)
...
push(r15)

// Task resume - restore context
pop(r0)
pop(r2)
...
pop(r15)
```

# RTOS ABSTRACTION LAYER

- There are literally hundreds of operating systems
  - Which to choose?
  - Once you have chosen, you are pretty stuck
- Unless you use an RTOS abstraction layer (OSAL)
- OSALs also allow easy cross-platform testing



## OSAL: KEY FUNCTIONALITY

- Task management
- Timers (delays)
- Communication (events)
- Synchronization (mutex semaphores)
- Interrupt handlers

# OSAL: TASK CREATION

```
OSTaskCreate(osTask * task, uint8_t priority_level, uint32_t stack_space);
```

- Create a task
  - Provide the name of the task function
  - Provide the priority level
  - Specify how much stack space is required
- This creates a Task Control Block (TCB)
- Creating tasks is expensive...

## OSAL: TASK DELETION

```
OSTaskDelete(osTask * task);
```

- Kill a running task
  - Provide the name of the task function
  - This will destroy a task and release all its assets

## OSAL: TASK DELAY

```
OSTaskDelay(uint32_t delay);
```

- Make a task wait for a set amount of time
- On low power systems, this delay can put the CPU to sleep

# OSAL: EVENT SIGNALLING

```
OSCreateEvent(osEvent *event);
```

- Create an event
  - Type of event: e.g. conversion\_complete
  - [Optional] data/message to communicate
- Can be created from another task or from an ISR

## OSAL: EVENT CONDITIONAL

```
OSWaitEvent(osEvent *event);
```

- Wait for a particular event
  - Type of event e.g. button pressed
- Kernel will suspend execution until an event of this type occurs

## OSAL: SYNCHRONIZATION

```
OSCreateMutex(osMutex *mutex);
OSAcquireMutex(osMutex *mutex);
OSReleaseMutex(osMutex *mutex);
```

- Recall that concurrent access to shared variables can lead to major issues
- A sensible approach is to make sure that one and only one task has access to a resource
- This is typically done with a lock for which only one task holds the key

## OSAL: ADDITIONAL FUNCTIONS

- Functions to setup interrupt handlers
- Functions to protect critical sections of code by disabling the kernel

## EXAMPLE: SIMPLE TASK

```
void task_flash(void)
{
 while (1)
 {
 LED_on();
 OSTaskDelay(100);
 LED_off();
 OSTaskDelay(1000);
 }
}
```

## EXAMPLE: RUN-TO-COMPLETION TASK

```
void task_10flash(void)
{
 uint8_t count = 0;
 while (count<10)
 {
 LED_on();
 OSTaskDelay(100);
 LED_off();
 OSTaskDelay(1000);
 count++;
 }
 OSTaskExit(); // Like OSTaskDelete, but called within a task
}
```

## EXAMPLE: COMMUNICATION

```
// ADC Interrupt Routine
void ADC_ISR(void)
{
 OSCreateEvent(adc); // send an event
}

// Task to check sensor readings
void task_sample(void)
{
 HAL_adc_start();
 while (1)
 {
 OSWaitEvent(adc); // wait until there is a new conversion
 sample = adc->data; // get the reading
 if (sample > VOLTAGE_MAX)
 {
 OSCreateEvent(alarm); // signal the ALARM event to another process
 }
 }
}
```

## EXAMPLE: SHARED RESOURCE

```
void task_print
{
 OSCreateMutex(&printer); // create a named mutex to access "printer" resource
 while(1)
 {
 OSAcquireMutex(&printer); // capture the mutex
 printf("Task1\n");
 OSReleaseMutex(&printer); // release the mutex
 OSTaskDelay(1000);
 }
}
```

# SCHEDULING

- Scheduling is the issue of how to order access to the CPU
- This is the key differentiator between RTOS kernels
- Common classes:
  - Rate Monotonic Scheduling (RMS)
  - EDF (Earliest Deadline First)
- Schedules can be cooperative, static or dynamic
  - Cooperative scheduling relies on a task yielding
  - The priorities of tasks are set at design-time in a static scheduler
  - The priorities of tasks are calculated at run-time in a dynamic scheduler

## IMPORTANCE OF SCHEDULING

- Task1: -  $C_1$ : 25ms (execution time) -  $T_1$ : 50ms (period) -  
 $U_1 = \frac{25}{50} = 50\%$  (utilization)
- Task2: -  $C_2$ : 40ms (execution time) -  $T_2$ : 100 ms (period)  
-  $U_2 = \frac{40}{100} = 40\%$  (utilization)
- Total utilization is 90%. Is it possible to meet both deadlines?

# CHALLENGES OF SCHEDULING

- Only two possibilities:
  - Case 1:  $\text{priority}(\text{Task1}) > \text{priority}(\text{Task2})$
  - Case 2:  $\text{priority}(\text{Task1}) < \text{priority}(\text{Task2})$
- Which case actually can be achieved in terms of meeting deadlines?
- Setting priorities is clearly critical

## CYCLIC EXECUTIVE SCHEDULE

- Simple approach
- Pack tasks into a cycle
- Cycle repeats infinitely
- Easy to guarantee predictability
- Hard to modify tasks and order

## RATE MONOTONIC SCHEDULING

- Assigns priorities based on how often a task needs to execute
- Optimal static scheduler
- Assumptions
  - All tasks are periodic (occur at regular intervals)
  - Tasks do not synchronize with another, share resources or exchange data
  - CPU always executes highest priority task

## RATE MONOTONIC SCHEDULING THEOREM

A schedule exists if  $\sum_i \frac{E_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$

- Note that critical functions can be run in the RMS framework, with non-critical functions in the background
- In the limit, this tends to  $\ln(2) = 69\%$
- This says that if we have utilization below the limit, then we are guaranteed to be able to derive a schedule

# PRACTICAL RATE MONOTIC SCHEDULING

- If tasks are harmonic lengths of one another, we can get 100% CPU usage safely<sup>2</sup>
  - E.g. {20,60,120} is harmonic as each task is a divisor of the longer task
  - E.g. {20,67,120} is not harmonic and thus more difficult to schedule

---

<sup>2</sup>Better Embedded Software: Koopman

## DYNAMIC SCHEDULING: EDF

- Earliest Deadline First: choose task that is closest to missing its deadline as the next one to run
- Dynamic scheduling algorithm
  - $\mathcal{O}(n^2)$  complexity
- In reality, EDF is very expensive to run online, so approximations of EDF are used in real systems

# COMPARISON OF SCHEDULING APPROACHES

- Table driven approaches are deterministic, but inflexible
- RMS is often used in safety-critical systems, as it is simpler to analyse than dynamic schedules
- Dynamic scheduling is often used in best-effort systems

# DEADLOCK

- Two processes out of sync, waiting for the other to send a message
- Example:
  - Task A holds resource X, waiting for Y
  - Task B holds resource Y, waiting for X
- This is a simple example, but it could be very intricate
- **Solution:** each task to only be allowed one mutex at a time

## PRIORITY INVERSION

- A high-priority task is waiting for an event from a low priority task
- It now runs as fast as the low-priority task
- **Solution:** Temporarily elevate the low-priority task

# STARVATION

- High priority tasks fire too often
- Low priority tasks do not get enough CPU time
- Leads to degradation of service
- **Solution:** Reassign priorities

## \*COROUTINES: BABY TASKS

- An alternative to fully-fledged tasks is to use coroutines instead
  - This is used in the ContikiOS for IoT end-devices
  - C++ 20 supports coroutines<sup>3</sup>
- Well suited to severely resource constrained devices
- Coroutines share a common stack
- However, no isolation between tasks

---

<sup>3</sup>Belson, Bruce, et al. "C++ 20 Coroutines on Microcontrollers-What We Learned." IEEE Embedded Systems Letters (2020).

## \*COST OF OPERATING SYSTEMS

Operating systems are not free<sup>4</sup>

|                                            | Transition time | Release time  | Average power consumption |
|--------------------------------------------|-----------------|---------------|---------------------------|
| Without RTOS,<br>interrupt driven          | 1.2 $\mu$ s     | 54.8 $\mu$ s  | 76.8 $\mu$ W              |
| FreeRTOS,<br>interrupt driven              | 3.6 $\mu$ s     | 69.6 $\mu$ s  | 579.6 $\mu$ W             |
| FreeRTOS,<br>tickless, interrupt<br>driven | 1.2 $\mu$ s     | 85.6 $\mu$ s  | 132.0 $\mu$ W             |
| FreeRTOS,<br>tickless, using<br>semaphores | 34.4 $\mu$ s    | 133.6 $\mu$ s | 304.6 $\mu$ W             |
| Keil RTX,<br>tickless, interrupt<br>driven | 3.6 $\mu$ s     | 89.6 $\mu$ s  | 156.1 $\mu$ W             |

<sup>4</sup>System Design Impacts on Battery Runtime of Wearable Medical Sensors, Tobola et al. MobileMed 2014

## SUMMARY

- Increasing the level of abstraction carries a performance penalty
- Operating systems are essential in complex systems
  - For speed, however, sometimes code in an ISR will live outside the OS
  - Models like FSMs are used within tasks as well

## FURTHER READING

- “Practical UML Statecharts in C/C++” - Samek
- “Design Methods for Reactive Systems: Yourdon, Statemate, and the UML (The Morgan Kaufmann Series in Software Engineering and Programming)” - R. J. Wieringa

## 9. OPTIMIZATION

---

# OVERVIEW

- How do we make things faster?
- How do we make things run longer?

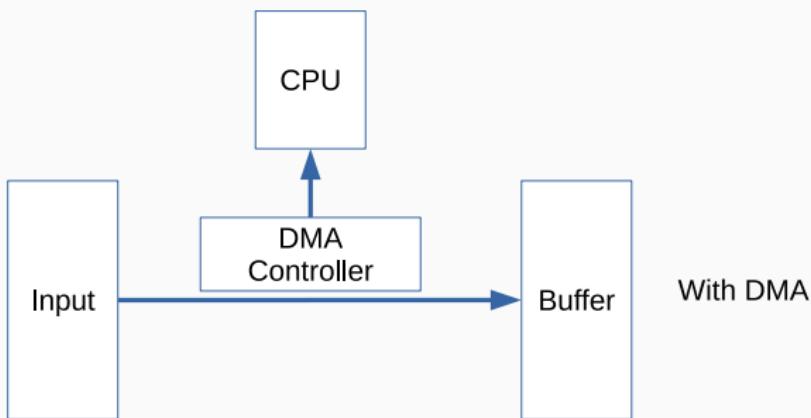
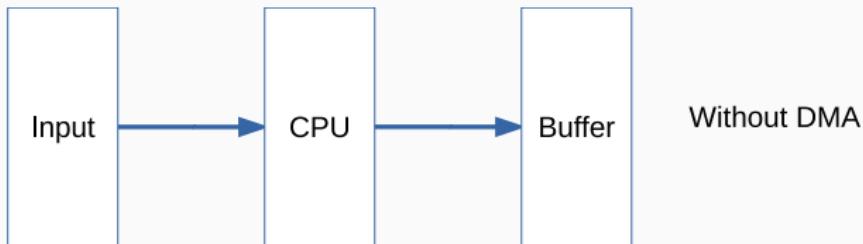
## OPTIMIZING FOR SPEED

- For high-speed signal processing, we can write better algorithms
- However, sometimes the hardware is simply not fast enough
- Alternative approaches to microcontrollers are sometimes needed
- Look at some HW and SW techniques
- Sometimes (but not always) making something faster will make it use less power

## HW: DMA

- DMA: Direct Memory Access
- A logic block that bypasses the CPU
  - Setup the DMA
  - It will run and copy data across
  - Trigger interrupt to indicate process is done
- Fewer interrupts: less context switching, less overhead
- Many embedded devices have DMA controllers
- Caveat: one more thing running in the background...

## HW: DMA

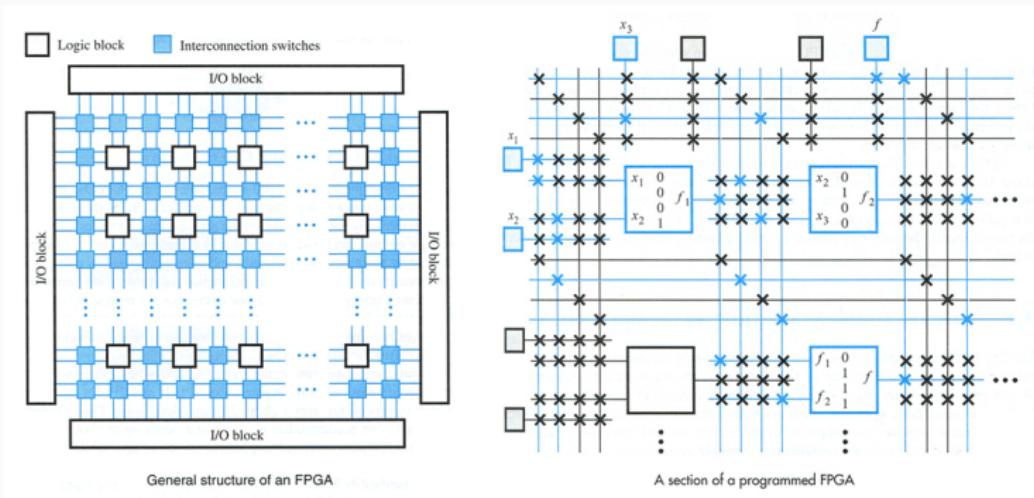


- For high speed, numerically intense algorithms, an FPGA device can yield significant speed gains
- FPGA = field programmable gate array
- Fabric of configurable logic blocks which can be connected in various ways
- Can be viewed as multiple FSMs running in parallel
- Also have components like DSP slices
- Written in Verilog or VHDL

## HW: FPGA

- High-end FPGA\footnote{Xilinx Virtex ~ \\$8k}
  - 2 M CLBs
  - 400 Mbyte RAM
  - 12 M DSP slices
  - 21 GFLOPs
  - 8000 Gb/s transceiver bandwidth
- IP blocks exist for CPU cores
  - FPGAs are sometimes used to prototype new CPUs

# HW: MATRIX AND LOGIC CELLS



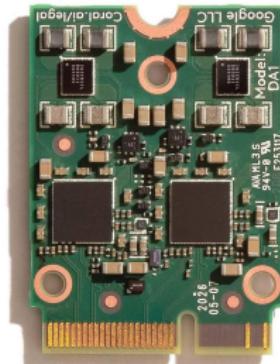
# SIMPLE FLASHING LED IN VERILOG

```
module blinker(
 input clk,
 input rst,
 output blink
);
reg [24:0] counter_d, counter_q;
assign blink = counter_q[24];
always @ (counter_q) begin
 counter_d = counter_q + 1'b1;
end
always @ (posedge clk) begin
 if (rst) begin
 counter_q <= 25'b0;
 end else begin
 counter_q <= counter_d;
 end
end
endmodule
```

- FPGAs are typically debugged in simulation
- Closer to designing logic circuits than writing code
- FPGAs are often used in safety critical applications as their behaviour is more straightforward to analyse
- They are also used in specific applications where speed is critical
  - CERN Hadron Collider
  - SKA Radio Astronomy

## HW: NPU

- As Edge AI becomes more important, microcontrollers are including Neural Processing Units (NPU)
- Google Coral AI
  - 4 TOPS @ 2W
  - Typical neural speedup of 10x-25x



## HW: SOC

- System-on-chips integrate blocks into a single device
  - e.g.
  - Microcontrollers
  - DSP
  - FPGA cells
  - Wireless
- SoCs give a lot of flexibility in a very small package
- The line between microcontroller, FPGA and SoC is becoming blurred

## TUNING SOFTWARE FOR SPEED

- Better algorithms
- Or simply exploiting what you know about how a microcontroller works...

# SW: OPTIMIZATION

- How to make this faster?

```
float mean(float a, float b)
{
 return ((a+b)/2.0);
}
float ave;
ave = mean(3.0,4.0);
```

## SW: INLINING

- The compiler will put the code **inline**
- This eliminates the call and return
- Trade code space for speed

```
inline float mean(float a, float b)
{
 return ((a+b)/2.0);
}
float ave;
ave = mean(3.0,4.0);
```

## SW: PASS BY POINTER REFERENCE FOR LARGE TYPES

- Again, think about the stack space to pass an arguments

```
struct big_struct {
 uint8_t status;
 uint32_t values[10];
};

// This copies the whole struct onto the stack
uint32_t slow_max(struct big_struct buffer)
{
 // code
}
// This just copies the pointer to the struct
uint32_t fast_max(struct big_struct * buffer_ptr)
{
 // code
}
```

# SW: LOOP UNROLLING

- This is sometimes done by the compiler

```
uint8_t i;
uint8_t sum = 0;
// Original loop
for (i = 0; i < 100; i++)
{
 sum++;
}
// Loop unrolled by factor 2
for (i = 0; i < 50; i++)
{
 sum++;
 sum++;
}
```

## SW: FIXED POINT

- Most microcontrollers do not have a floating point unit (FPU)
- Floating point operations are significantly more expensive than native integer operations
- Approximate floating point arithmetic using fixed point integers

# FIXED POINT: WHAT IS A FIXED POINT NUMBER?

- An n bit fixed point number consists of:
  - An **integer** (whole number) part
  - A **fractional** part
- The **radix** (decimal point) specifies the boundary between the integer and fractional part



## FIXED POINT: A SIMPLE EXAMPLE

- Consider a 3 bit number. It has values from 0-7 under a straightforward binary encoding
- We can use a number of bits ( $f$ ) for the fractional part
- Altering  $f$  changes the scaling/mapping
- To get the fixed point representation:
  - Take the decimal equivalent of the binary
  - Divide by  $2^f$

# FIXED POINT: A SIMPLE EXAMPLE

- Example with different fractional bits

| Binary | f=0 | f=1 | f=2  | f=3   |
|--------|-----|-----|------|-------|
| 000    | 0   | 0.0 | 0.0  | 0.0   |
| 001    | 1   | 0.5 | 0.25 | 0.125 |
| 010    | 2   | 1.0 | 0.50 | 0.250 |
| 011    | 3   | 1.5 | 0.75 | 0.375 |
| 100    | 4   | 2.0 | 1.00 | 0.500 |
| 101    | 5   | 2.5 | 1.25 | 0.625 |
| 110    | 6   | 3.0 | 1.50 | 0.750 |
| 111    | 7   | 3.5 | 1.75 | 0.875 |

## FIXED POINT: A SIMPLE EXAMPLE

- Note that  $f$  does not need to match the bit width and can even be negative!

| Binary | f=-1 | f=0 | f=1 | f=2  | f=3   | f=4    |
|--------|------|-----|-----|------|-------|--------|
| 000    | 0    | 0   | 0.0 | 0.0  | 0.0   | 0.0    |
| 001    | 2    | 1   | 0.5 | 0.25 | 0.125 | 0.0625 |
| 010    | 4    | 2   | 1.0 | 0.50 | 0.250 | 0.1250 |
| 011    | 6    | 3   | 1.5 | 0.75 | 0.375 | 0.1875 |
| 100    | 8    | 4   | 2.0 | 1.00 | 0.500 | 0.2500 |
| 101    | 10   | 5   | 2.5 | 1.25 | 0.625 | 0.3125 |
| 110    | 12   | 6   | 3.0 | 1.50 | 0.750 | 0.375  |
| 111    | 14   | 7   | 3.5 | 1.75 | 0.875 | 0.4375 |

## FIXED POINT: REPRESENTATIONS

- Some common representations
  - 8.8 - Unsigned, 16 bit word, range from 0.00 to 256 -  $2^{16}$  (255.996094)
- Fractional fixed point are very commonly used, with a maximum value of 1.00
  - Q15 - Signed, 16 bit word, range from -1.00 to +1.00- $2^{16}$
  - Q31 - Signed, 32 bit word, range from -1.00 to +1.00- $2^{32}$
- Fractional fixed point will not overflow under multiplication

## FIXED POINT: OPERATIONS

- Addition and subtraction work as expected
- Multiplication does not work quite as expected:
  - Floating point:  $a \times b = ab$
  - Fixed point:  $\frac{a}{2^f} \times \frac{b}{2^f} = \frac{ab}{2^{f+f}}$
  - When multiplying, we need to scale by  $2^f$  to get the correct result of  $\frac{ab}{2^f}$
  - Luckily, this can be done using left shift, which is a fast operation on a microcontroller
- Division is a little more complex still...
  - If dividing by a known number, simply multiply by the reciprocal

## FIXED POINT: CARE AND ATTENTION

- Fixed point is fast, but it can introduce some arithmetic issues
- **Truncation**: when multiplying two small values together, the lower bits get lost
- **Windup**: Due to the fixed precision, if integrating over a long time-period, the rounding errors can compound and grow exponentially
- **Saturation**: Accumulating partial results can lead to overflow - if not handled, results can wrap
- Tools (Matlab) exist that convert floating point algorithms to fixed point

## FIXED POINT: SUMMARY

- Fixed point is useful for devices without a floating point unit
- Fixed point arithmetic can be an order of magnitude faster than software based floating point
- Fixed point with reduced precision can greatly increase neural throughput (e.g. INT8 vs FLOAT32)
- However, use with caution, as it can lead to numerical peculiarities

## SW: AVOID EXPENSIVE ARITHMETIC INSTRUCTIONS

- Many algorithms have a large number of floating point instructions, especially transcendent ones (sine, cosine etc)
  - Cyclical Redundancy Checks (CRC)
  - Sensor Linearization
  - Motor control (sine-wave generation)
  - High order polynomial functions
  - Counting number of set bits in a word
- Remember our “transformative” view of computing?
  - A function maps the input to an output

## SW: LOOKUP TABLES

- An approach to speeding these algorithms up is to precompute some or all of the mapping values
- Lookup tables store the results of the computation
- These can be populated at:
  - Compile time (best)
  - Run time (if necessary)

## LOOKUP TABLE EXAMPLE

- Taylor series approximation for  $\sin(x)$
- $\sin(x) \approx x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040}$
- Populate a table depending on the level of domain precision required
  - A 100 element table will give an angular precision of  $3.6^\circ$
- Further tricks
  - Interpolate between two adjacent values in the table to boost precision
  - Note that  $\sin(x)$  is a periodic, odd function, so we only store the first quadrant ( $90^\circ$ )

## OPTIMIZING FOR ENERGY CONSUMPTION

- Some techniques discussed above (e.g. fixed point) can be used to reduce overall energy usage
- Again, there are hardware and software techniques to reduce power consumption

## HW: TURN IT OFF

- The simplest way to reduce power consumption is to turn the power off
  - peripherals
  - the CPU itself
- However, this means they can't do work while off
  - e.g. receive data
- They also might take time to restart or require re-initialization
- In the case of the CPU, it needs some background peripheral that can wake it up through an interrupt

## HW: TURN IT OFF

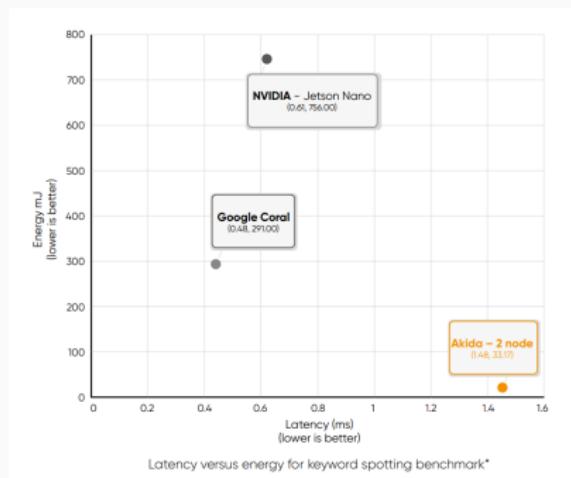
- External devices also draw power when idle
  - wireless transceiver
  - memory
  - sensors
- Use GPIO pins to disable power to any energy suckers
- Again, make sure that the peripheral is reinitialized correctly

## HW: MOVE COMPUTATION INTO MORE EFFICIENT DEVICES

- Many systems are bursty i.e. most of the time is spent doing very little
- However, to meet peak demand, we need a high-speed CPU
- Unfortunately, this consumes a large amount of energy when idling
- One solution is to use a low end, energy efficient microcontroller that wakes up the high-speed CPU when needed
- Discrete solutions abound, but integrated solutions (ARM BIG.little) are emerging

# HW: NOVEL COMPUTATION

- For Edge AI, neuromorphic (spiking) computation provides an alternative to MCUs or NPUs
- High performance, at low power<sup>1</sup>



<sup>1</sup>Akida Brainchip: [brainchip.com](http://brainchip.com)

- DMA controllers consume less power than the CPU
  - E.g. acquire 512 samples from ADC
- If done using DMA, the CPU can be put to sleep while the DMA is running
- When the buffer is full, an interrupt will be triggered which will wake the device

## HW: FREQUENCY SCALING

- Altering the clock frequency is a common strategy to drop overall power consumption
  - Microcontrollers typically have power figures measured in A/MHz e.g. 0.15mA/MHz
- When not doing intensive computations, drop core speed to minimum e.g 32kHz
- It is often more energy efficient to do a computation at the maximum clock rate and then put the processor entirely to sleep...

## HW: VOLTAGE SCALING

- Energy in a CMOS circuit scales quadratically with voltage ( $E \propto V^2$ )
  - Scaling voltage is less common in embedded systems, but can make further gains
- Research is ongoing into schedulers which automatically scale frequency/voltage to meet real-time constraints<sup>2</sup>

---

<sup>2</sup>A framework of concurrent task scheduling and dynamic voltage and frequency scaling in real-time embedded systems with energy harvesting, Lin et al. 2012

## HW: RUN FROM RAM

- Powering up the FLASH memory consumes a relatively high idle current
- However, the program code is in Flash
- To drop power consumption, portions of code can be copied and executed from RAM
- Flash is turned off
- This is quite a challenging technique to get right!

## SW: RATE-ADAPTIVE SAMPLING

- The standard approach to acquiring data from a sensor is to sample at twice the maximum bandwidth to satisfy Nyquist Criterion
- If however, there is some prior knowledge on the dynamics of the signal, then this can be exploited
  - If a user hasn't pressed a key in the last minute, drop key scanning rate to 1 Hz from 10 Hz
  - If an accelerometer reading hasn't changed in the last minute, halve the sampling rate
- A more sophisticated approach is to choose a sampling rate that gives ideal reconstruction, based on the signal properties

## SW: COMPRESSION

- If an acquired waveform needs to be stored or transmitted wirelessly, compressing it will save energy
  - Iff the gains from compression outweigh the cost of running the compression algorithm
- Compression algorithms exploit redundancy in signals
  - Run Length Encoding (RLE) is a simple example that replaces repeated occurrences of a symbol
  - Huffman coding builds a dictionary of symbol frequency
- Compressed Sensing is a ‘new’ (2006) technique to compress a signal at source

## SW: EFFICIENT ALGORITHMS

- Good algorithm design can greatly reduce computation time
- Consider the common case of detecting whether or not a signal (tone) is present in a waveform
- One way would be to take an FFT of a window and extracting the peak
- However, if we already know what frequency we are looking for (e.g. carrier tone), we can use a very efficient filter (Goertzel Algorithm)

## WHAT TO OPTIMIZE?

- When reducing power consumption, it is challenging to know where to start
- The first step is to build a power budget:
  - how much energy is used in each mode (e.g. sending data)
  - and what the contributors to the consumption are
- Start with the biggest energy drainer and optimize that
- Iterate until you have
  - reached your goals
  - reached the limits of physics
  - run out of time/money

## SUMMARY

- Optimization is one of the most critical areas in embedded systems
- From a commercial perspective, it is what differentiates similar products
  - e.g. higher bandwidth for same price
  - e.g. equivalent spec. with cheaper BOM = more profit
  - e.g. longer lifetime for same sized battery
- Combination of hardware and software (algorithms) working together

## FURTHER READING

- “Making embedded systems” - White
- “The Art of Embedded Systems” - Ganssle