

**ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA ĐIỆN-ĐIỆN TỬ
BỘ MÔN KỸ THUẬT ĐIỆN TỬ**



Master Course

Advanced Embedded System Design

Chapter 0: Course Introduction



Course Information

- Instructor
 - Truong Quang Vinh, Ph.D.
 - Department of Electronics
<http://www.dee.hcmut.edu.vn/vn/bomon/bmdientu>
 - Email: tqvinh@hcmut.edu.vn
 - Homepage: <http://www4.hcmut.edu.vn/~tqvinh>
 - Office: 118B1, IC Design Lab, Monday 9-11am
- Related undergraduate courses:
 - Micro-processor (Vi xử lý)
 - Embedded system design (Thiết kế hệ thống nhúng)
 - Embedded programming (Lập trình nhúng)



Textbooks

- [1] Frank Vahid and Tony Givargis , **Embedded System Design: A Unified Hardware/Software Approach**, John Wiley & Sons, Inc. 2002
- [2] Joseph Yiu, “**The Definitive Guide to the ARM Cortex-M3**”, Elsevier Newnes, 2007
- [3] Jonathan W Valvano, **Embedded Systems: Introduction to Arm® Cortex(TM)-M Microcontrollers** (Volume 1), 2012
- [4] Jonathan W Valvano, **Embedded Systems: Real-Time Interfacing to Arm® Cortex™-M Microcontroller**, 2012



Course Description

- This course provides students with advanced knowledge of embedded system design process.
- Students will have ability to
 - **design** hardware part of an embedded system using ARM microcontroller with peripherals including GPIO, ADC, UART, SPI, USB, and Ethernet.
 - **program** software part of an embedded system with and without operating system using C programming language.
 - **develop** an embedded system project using Proteus, IAR, and KeilC development tools.

Undergraduate course: Embedded system design

Contents	Embedded system design	Advanced embedded system design
Embedded system design methodology	<ul style="list-style-type: none">-Apply design process-Design hardware by block diagram and schematic-Develop software by flow chart and C program	<ul style="list-style-type: none">-Apply design process-Design hardware by block diagram and schematic-Develop software by flow chart and C program-Analyze features and issues: hardware & software-Optimize design metrics-Quality function deployment
Microcontroller	PIC16F84 and PIC16F877	ARM Cortex M3 and M4
Hardware & software tools	PIC kit Proteus CCS C	LM4F120 launch pad STM32F4 Discovery Code Composer Studio Keil ARM



Syllabus

Week	Content	Note
1	Chapter 0: Course introduction 0.1. Course information 0.2. Syllabus and schedule 0.3. Course preparation Require students to prepare textbooks, tools, and course materials	Students select class project's topics
2	Chapter 1: Embedded System Design Process 1.1. Embedded system features and issues 1.2. Embedded system design process 1.3. Embedded system analysis Require self-studying for 3 hours	Quiz
3,4	Chapter 2: Microcontroller Series 2.1. ARM Cortex-M3 2.2. ARM Cortex-M4 Require self-studying for 6 hours	Assignment 1



Syllabus

5,6	Chapter 3: C Programming for Embedded Systems 3.1. C Program Basics 3.2. ARM Cortex-M C Compiler 3.3. ARM software library 3.4. FreeRTOS Require self-studying for 6 hours	Assignment 2
7	Chapter 4: Development tools 4.1. Advanced simulation with Proteus 4.2. Programming tools: IAR and Keil Require self-studying for 3 hours	Assignment 3
8,9	Chapter 5: Using Peripherals and Interrupts 5.1. Parallel IO ports 5.2. Timers 5.3. Interrupts 5.4. Analog IO 5.5 Serial communication Require self-studying for 6 hours	Assignment 4



Syllabus

10	Chapter 6: Designing an embedded system project 6.1. Project description 6.2. Hardware design 6.3. Software design 6.4. Design simulation 6.5. Design verification Require self-studying for 3 hours	
11-15	Experiment 1. ARM Cortex M3 with Stellaris LM3S9B96 kit 2. ARM Cortex M4 with Stellaris EK-LM4F120XL kit Require self-studying for 10 hours	Students do experiments at lab
16-19	Class project Each group of students do class project at lab	Students do class projects
20	Present class project Each group of students presents and reports the class project	Students report class projects



Grading

- Final exam: 50%
- Lab: 10%
- Assignment: 10%
- Project: 30%
 - 2-3 students for one group
 - Select project's topic at **week 3**
 - Submit project at **week 16**

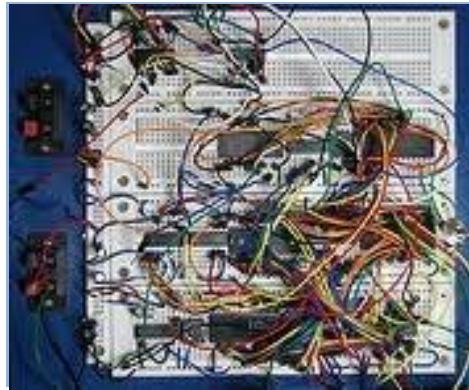


Course Preparation

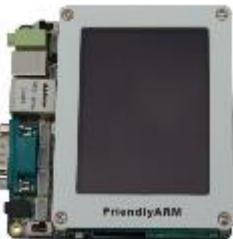
- Textbooks:
 - download 3 required textbooks
- Software tools:
 - Code Composer Studio (CCS)
 - KeilC
- Programming knowledge:
 - C/C++ programming

Project's requirements

- Report in MS Word (follow embedded system design process)
- Simulate the design
- Make prototype by bread board or PCB board.
- Present the design in class



Development Boards

**FriendlyARM Mini2440 Board**ARM9 board, 400 MHz, 64MB RAM,
256MB Nand, 3.5" Touch screen

LCD



Giá : 2,399,000 VND / Cái

**LM4F120 LaunchPad**

Evaluation

Stellaris® LM4F120 LaunchPad

Evaluation



Giá : 550,000 VND / Cái

**BeagleBoard-xM**ARM Cortex -A8 MHz Board, 512MB
RAM, 1 GHz CPU, BeagleBoard

Giá : 4,299,000 VND / Cái

**STM32F4-Discovery
Cortex-M4 Kit**STM32F4 DISCOVERY (ARM
Cortex M4 + DSP Core)

Giá : 449,000 VND / Cái

**STM32F3 Discovery**

KIT EVAL DISCOVERY STM32F3



Giá : 460,000 VND / Cái

**STM32F4Discovery
EXTBOARD**

STM32F4 mother board, RS232 ,

LCD touch, CAN, Network



Giá : 1,199,000 VND / Cái

**Arduino Due R3**SAM3X8E ARM Cortex-M3 CPU,
84Mhz, 96 KBytes SRAM

Giá : 899,000 VND / Cái

**Raspberry Pi Model B**(Made in UK) BCM2835 700MHz
ARM1176JZF-S processor with FPU
and Videocore 4 GPU

Giá : 1,235,000 VND / Cái

Note: Friendly ARM, LM4F120 LaunchPad, BeagleBoard-xM are **available** at the Lab 116B1



Recommended class project topics

Using **STM32F3-Discovery Kit / STM32F4-Discovery Kit/ LM4F120 LaunchPad**

1. Hand motion detection
2. Remote Control through Ethernet/Zigbee/Bluetooth
3. Temperature & humidity measurement
4. Solar control system
5. Motor control system
6. RFID reader

Using **Friendly ARM kit / Beagle Board / Raspberry Pi**

1. Image capturing system
2. Data acquisition system
3. Object detection & recognition
4. Remote Control through Ethernet
5. MP3 system

Simple project's Topics

1. 20-Chasing LEDs (at least 10 modes)
2. LED Message Board (8x32)
3. 3D-LED cube (3x3x3)
4. LED fan display
5. Two-LED Dice
6. Two-digit 7-Segment LED counter up/down
7. Digital clock with LCD display
8. Voltmeter with LCD display
9. Calculator with keypad and LCD
10. Serial communication-based calculator
11. Step motor controller
12. DC motor controller using PWM
13. I2C data communication
14. Battery charger (1A)
15. Temperature controller
16. Alarm controller using IR LED
17. Automatic light controller
18. Simple music keyboard
19. Digital door lock
20. SD card project



Course Overview

1. What is an **embedded system**?
2. What are **differences** between embedded system and general computer system?
3. What are **applications** for embedded systems?
4. What is the **most important part** in an embedded system?
5. Which kind of embedded system **development boards** have you practiced on?
6. Which kind of **micro-processors** do you have experience on?



Advanced Embedded System Design

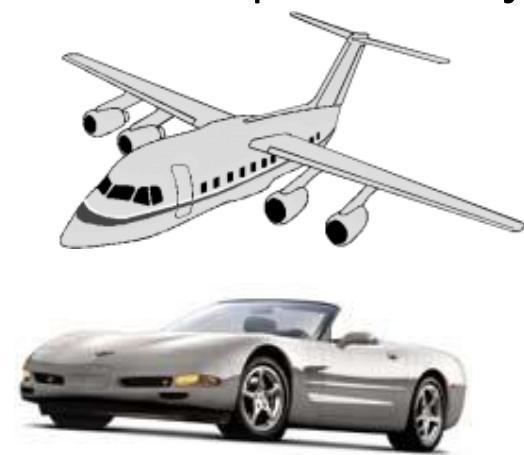
Chapter 1: Embedded System Design Process

1. Embedded system features and issues
2. Embedded system design process
3. Embedded system partitioning

1. Embedded system features and issues

- **Embedded system definition**

- any device that includes a programmable computer but is not itself a general-purpose computer - **Wayne Wolf**
- An embedded computer system includes a micro-computer with mechanical, chemical and electrical devices attached to it, programmed for a specific dedicated purpose, and packaged as a complete system - **Jonathan W. Valvano**
- An embedded system is one that has computer-hardware with software embedded in it as one of its most important component - **Raj Kamal**



1.1. Embedded System Features

Common Features (Non-technical specification)



Cost



Applications



Speed



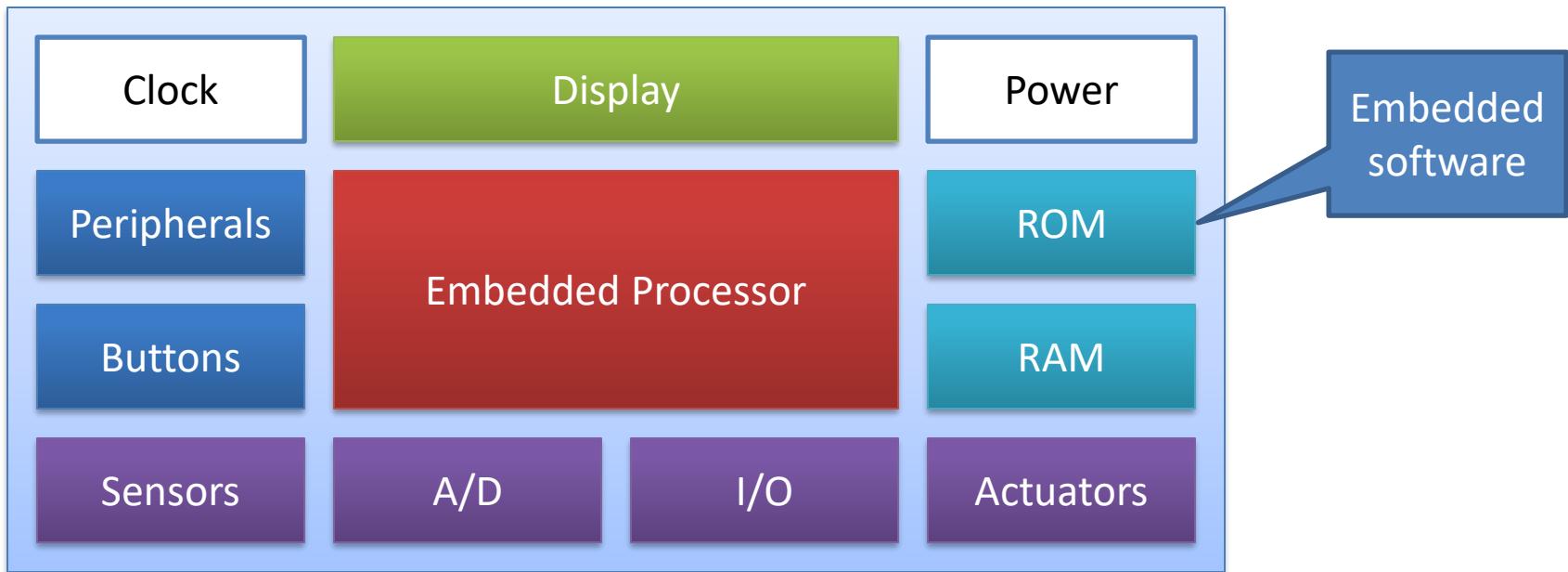
Size/Weight



Power/Energy

1.1. Embedded System Features

□ HARDWARE FEATURES



- **Embedded processor** is a heart of the embedded system:
 - micro-processor: 8086, ARM7/9/11, PowerPC, Nios
 - micro-controller: 8051, ARM Cortex-M, PIC16F, 68HC11, LM4F120, STM32F4

1.1. Embedded System Features

- **Memory**
 - ROM: embedded program
 - RAM: processing data
- **Peripheral interface**
 - GPIO, UART, I2C, SPI
 - ADC, PWM
 - USB, MicroSD
 - Ethernet, wifi
- **User interface**
 - Button / keypad / switch
 - LCD / 7-segment LED / LED indicator
- **Sensors**
 - temperature, humidity, light, ultrasound, pressure
 - Motion sensor, gyroscope
- **Actuators**
 - motor, solenoid, relay, ...
- **Clock**
 - crystal
- **Power**
 - +5V / +3.3V / +2.5V
 - Battery, DC adapter, ...

1.1. Embedded System Features

□ SOFTWARE FEATURES

- **Operating system**
 - Non-OS
 - RTOS, LynxOS, RTLinux
 - Linux, Android, WinCE
- **Embedded programs**
 - Supported functions
 - User applications
 - Software update
- **User Interface**
 - Command line
 - Text display
 - Graphic user interface

1.2. Embedded System Design Issues

- **Design issues** are problems that make it difficult to design an embedded system

1. Constraint issues

- cost may matter more than speed
- long life cycle
- Reliability/safety
- Low-power
- Size / weight

Examples: Portable heart-beat monitor

- Long life cycle (10 years)
- Reliability (accuracy 99%)
- Low-power (5 using days)
- Light weight (<1kg)



1.2. Embedded System Design Issues

1. Constraints

Examples for smart home system

No.	Constraints	Note
1	Low price (< 1.000.000 dong)	Correct
2	Ability to detect smoke and fire	Wrong
3	Low power (100mW when idle, 3W when active)	Correct
4	Response time for control < 1ms	?
5	Support remote control by smartphones	?
6	Easy to install	?

Constraints are limitations or restrictions of some parameters of the system

1.2. Embedded System Design Issues

2. Functional issues

- safety-critical applications
- damage to life, health, economy
- affect to environment, society, politics

Example:

- Message LED for a shop
 - Display message for customers
 - Malfunctions result in small damage
- Message LED for a stock market
 - Display stock data
 - Malfunctions could damage to economy
- Battery charger
 - ?



1.2. Embedded System Design Issues

2. Functional issues

Examples for battery charger

No.	Issues	Note
1	Battery can be over-heat, it need to be detected by a sensor	Correct
2	Display a charging current and battery status	Wrong
3	The system must have a fuse for protection of over current	?
4	Support 3 charging modes	?
5	Apply efficient algorithm for fast charging and increase battery life cycle	?

Functional issues are problems which can affect to life, health, economy, environment, society, politics, ethics.

1.2. Embedded System Design Issues

3. Real-time issues

- Determine whether the system is hard/soft/non real-time
- Determine the time constraint (delay)

Example

- Door entry alarm
 - Non/soft real-time system: delay < 1-2s
- Video recorder
 - Soft real-time system: delay < 1ms
- Car airbag system
 - ?
- Weather temperature monitoring
 - ?





1.2. Embedded System Design Issues

4. Concurrent issues

- System and environment run concurrently
- multi-functions
- interface with other systems
- May need a scheduler to manage concurrent tasks

Examples: Weather temperature monitoring

Multi-functions:

- Read temperature values from the sensor
- Write data to memory
- Display data on LCD

1.2. Embedded System Design Issues

5. Reactive issues

- **Continuous / discontinuous interaction**
 - Power on demand
 - Turn ON when using
 - **Ex:** MP3 player, Tivi system
 - Always ON, once started run forever
 - Continuous interaction with their environment
 - Termination is a bad behavior => watchdog timer
 - **Ex:** Camera surveillance system, data acquisition system
- **Response to external periodic/non-periodic events**
 - Events are periodic: the system needs a scheduler to capture the events
 - Events are non-periodic: the system needs to estimate miss event cases

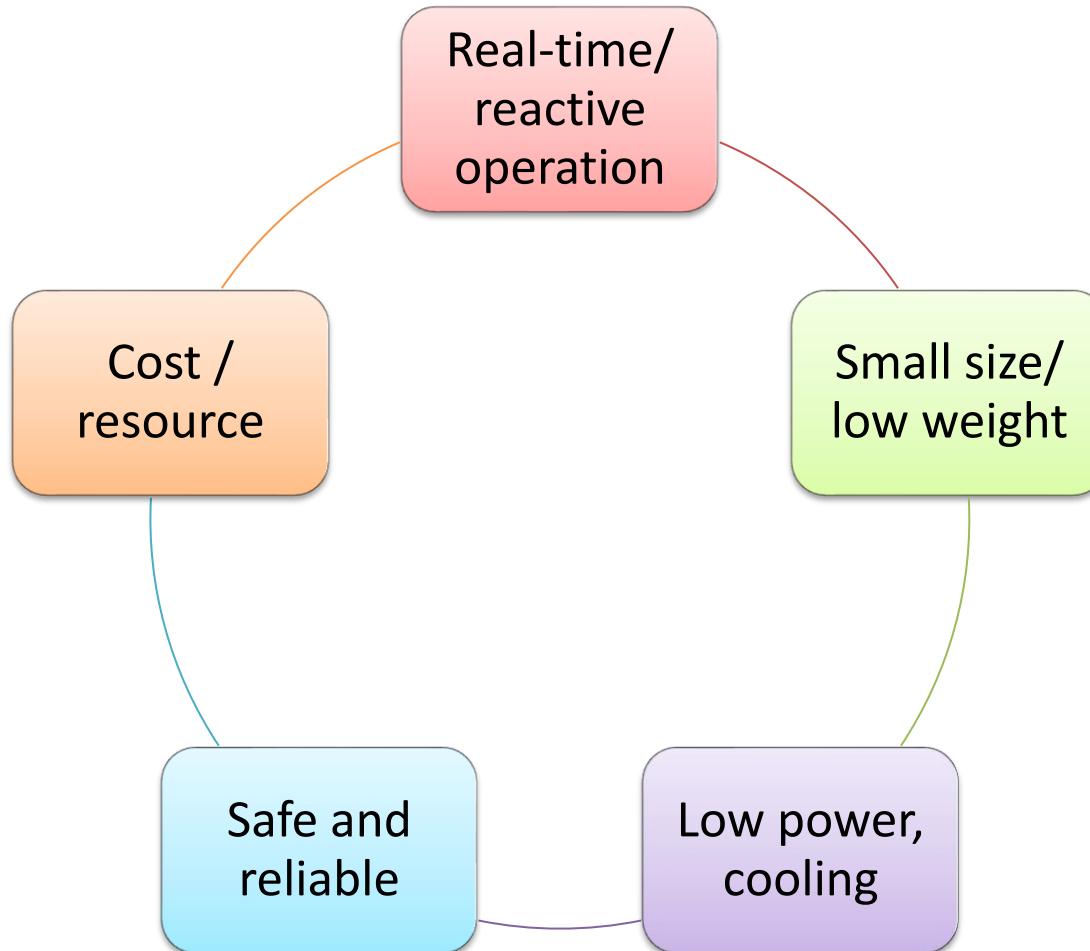
1.2. Embedded System Design Issues

- Group discussion
 - Discuss about design issues of your own class project



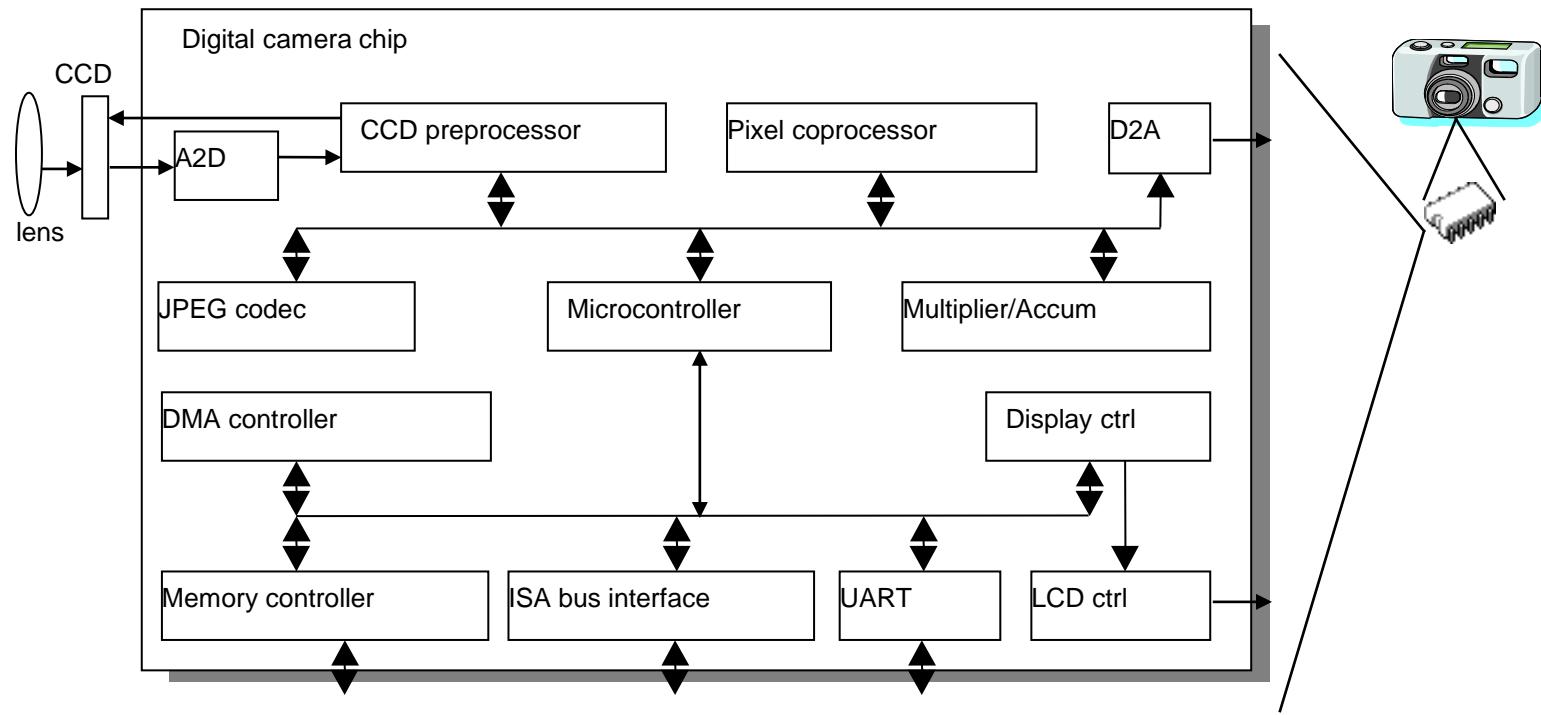
1.2. Embedded System Issues

- Basic requirements for an embedded system



An embedded system example - 1

A digital camera



- **Constraint:** Low cost, low power, small, fast
- **Function:** capture and store pictures / videos
- **Real-time:** only to a small extent
- **Concurrent:** single function
- **Reactive:** power on demand

An embedded system example - 2

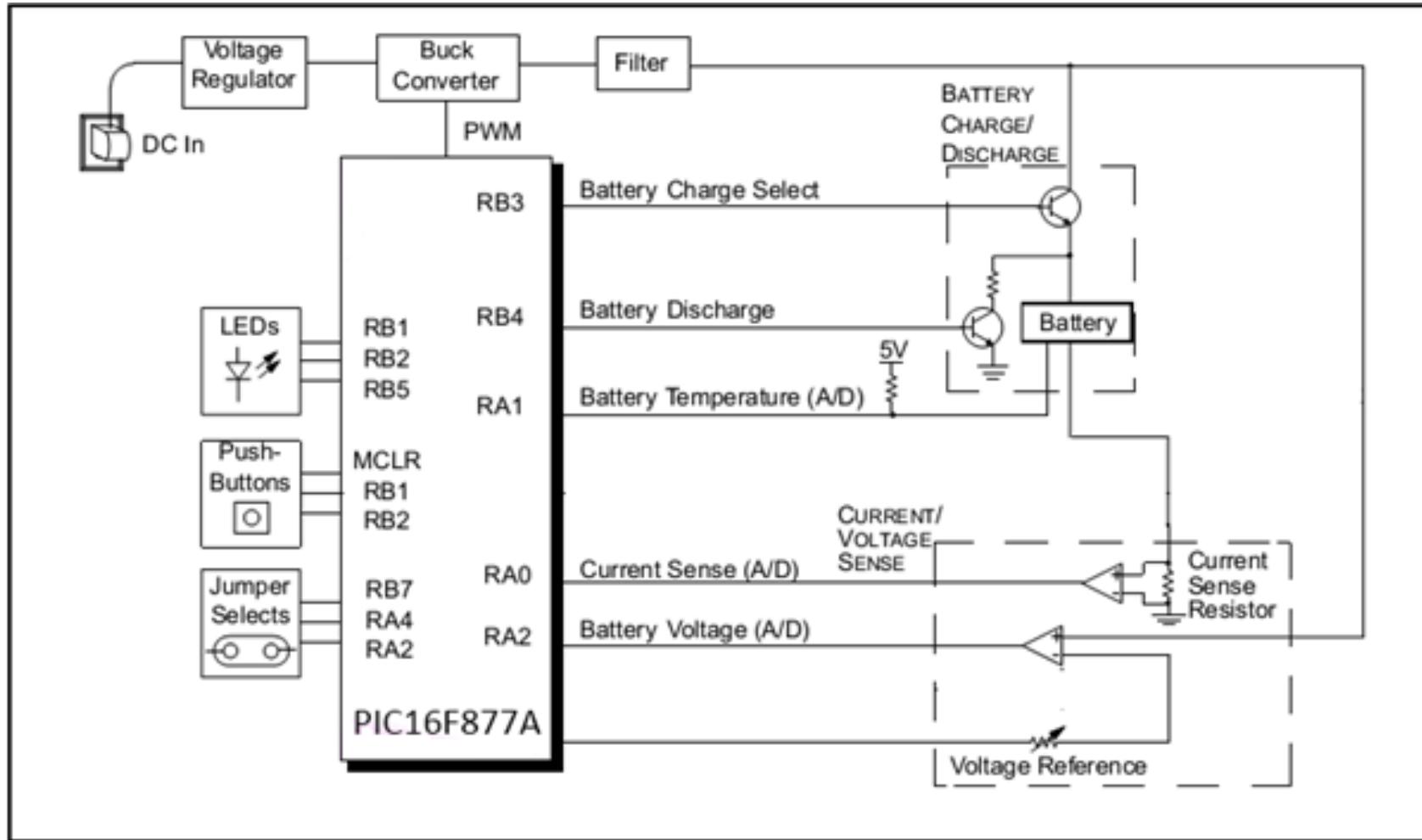
Battery charger

- Common features
 - **Cost:** Low Cost
 - **Applications/Functions**
 - Ability to charge NiCd, NiMH, Li-Ion batteries
 - User Selectable Charging Algorithms
 - High Charge Current Capability
 - Electrical safety
 - **Speed:** Fast Charge Rate
 - **Size/weight:** Small size, portable
 - **Power/energy:** 12V DC adapter



An embedded system example - 2

- Battery Charger Design



An embedded system example - 2

Battery Charger

- Hardware features
 - **Processor:** PIC16F877A
 - **Memory:** 8k Flash ROM
 - **Peripheral interface:** GPIO, ADC, RS232
 - **User interface:** text LCD, LED, button, Jumper
 - **Sensor:** current sensor, voltage sensor
 - **Actuator:** 5A BJT
 - **Clock:** 16MHz
 - **Power:** 5VDC

An embedded system example - 2

Battery Charger

- Software features
 - **Operating system:** Non-OS
 - **Embedded programs**
 - Charging controlling
 - Charging current, voltage displaying
 - **User Interface**
 - Text LCD: displaying mode, current, voltage
 - LED: indicating status
 - Button: selecting charging mode
 - Jumper: selecting battery type

An embedded system example - 2

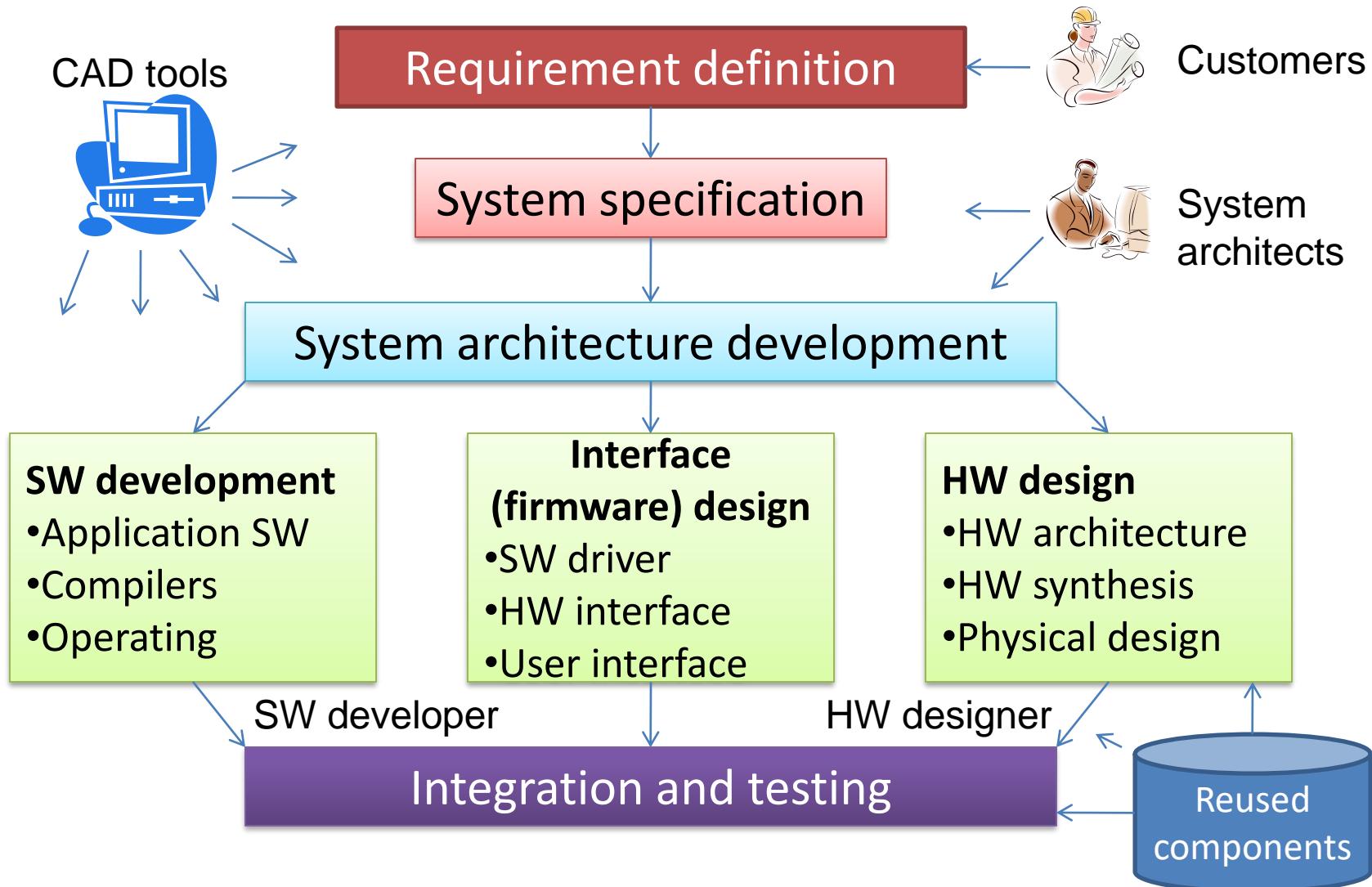
- Design Issues
 - **Constraints**
 - High charging current: 3A
 - **Functions**
 - Charging process: bulk charge, absorption charge, float charge
 - Safety: fuse protection, insulating box
 - **Real-time system**
 - Non-critical real-time system
 - Charging process based on predefined timer
 - **Concurrent system**
 - Charging control and current/voltage display
 - **Reactive system**
 - Non-continuous interaction



Quiz

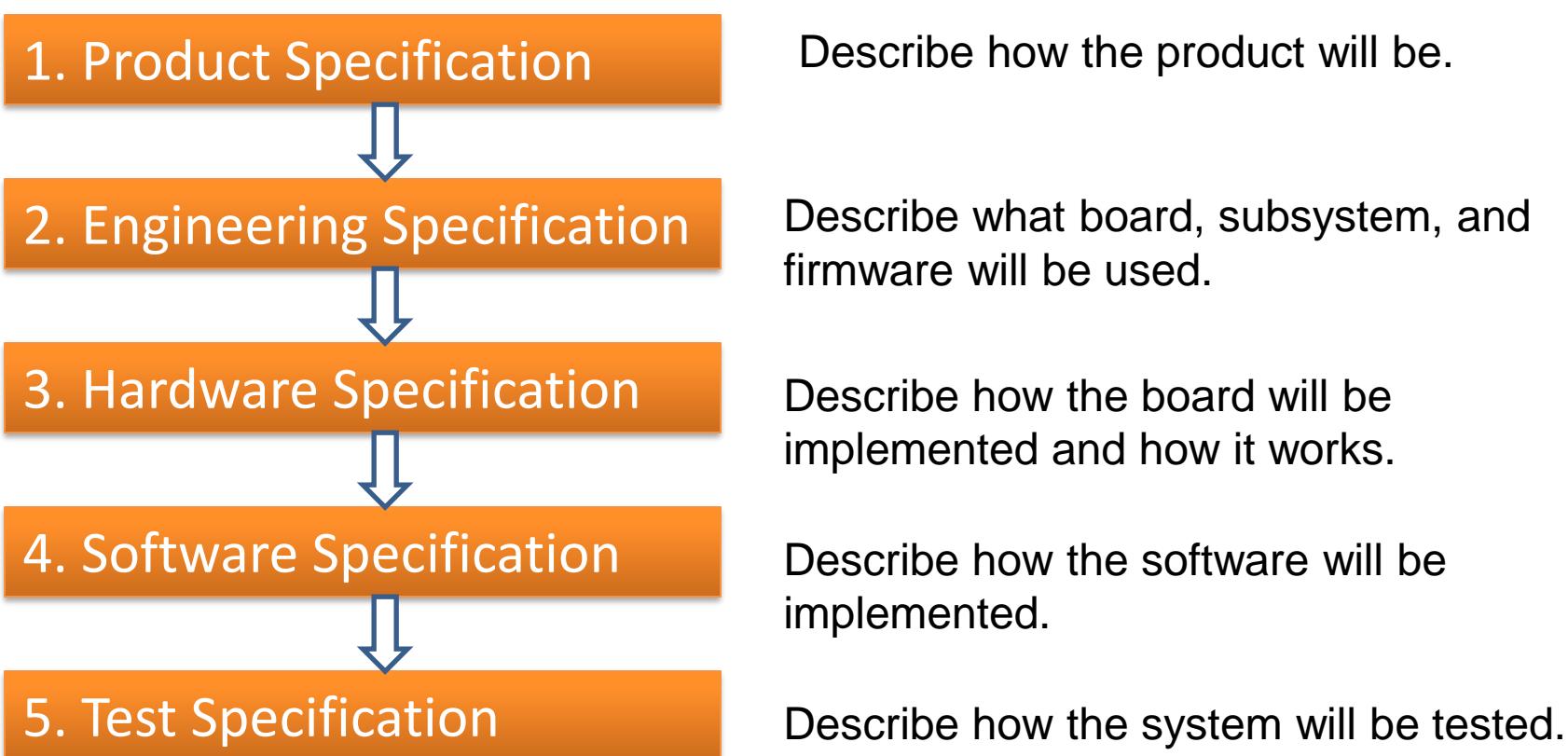
1. What is an embedded system?
2. What are embedded system features?
3. What are issues of embedded system design?
4. What are five basic requirements of embedded system?
5. What are common characteristics of an embedded system?
6. What are optimizing design metrics?
7. What is the most importance step of embedded system design process?
8. What are three key technologies for embedded systems?
9. Design features for the following design:
 1. Digital door lock
 2. Digital clock
 3. 3D LED cube

2. Embedded System Design Process



2.1. System Specification

Documents for System Specification



2.1. System Specification

1. Product Specification/Product Requirement

- What the system is to do ?
- What is the user interface?
- What the real world I/O consists of ?
- What are the functions ?
- What is the performance ?
- What the external interface to other system is (if any) ?
- What are the constraints? (speed, stability, low power, cost) ?



Sample Requirements Form (1/3)

- 1. Name
- 2. Purpose
- 3. Inputs and outputs
- 4. Use cases
- 5. Functions
- 6. Performance
- 7. Manufacturing costs
- 8. Power
- 9. Physical size/weight
- 10. Installation
- 11. Certification



Sample Requirements Form (2/3)

1. Name:

Name of the product

2. Purpose

One- or two-line description of the purpose

3. Inputs and outputs

Types of data: analog? digital? mechanical? . . .

data characteristics: periodic? occasional? how many bits? . . .

. . .

types of I/O devices: buttons? A/D converters? video displays? . . .



Sample Requirements Form

4. Use case

How user will interact with the system

5. Functions

more detailed description of the system

when the system receives an input, what does it do?

how do interface inputs affect these functions?

how do different functions interact?

6. Performance

must be identified earlier to ensure that the system works properly



Sample Requirements Form

7. Manufacturing costs

- cost has substantial influence on architecture
- work with some idea of the cost range

8. Power

- battery powered? plugged into a wall?

9. Physical size/weight

- more or less flexibility in the components to use

10. Installation

- Device is fixed, wall mounted or on desk, etc

11. Certification

- Need to meet standards for safety, compatibility.
- Some Certification is UL, CE, FCC, IP



2.1. System Specification

2. Engineering Specification / Design Specification:

- Written from a designer point of view
- System architecture
- Block diagram if appropriate
- What kind of hardware will be used
- What are the requirements for hardware and software
- Sample form:
 - 1. Principle of operation
 - 2. Environment
 - 3. System block diagram
 - 4. Block description
 - 5. Hardware software partitioning

2.1. System Specification

3. Hardware Specification:

- The requirements from engineering documents
- How the hardware implements the functionality
- The software interfaces to the hardware
- Example: **Oven temperature control system**
 - PIC Microcontroller 12MHz, sensor LM35, LCD 16x2-B, Keypad 16, ADC0809, FET IRF260 for heater/fan control
 - Microcontroller reads temperature value from LM35 through ADC, display this value to LCD, and then control heater and fan based on PID control algorithm

2.1. System Specification

4. Software Specification:

- The requirements from engineering specification
- Interface to other software
- How the software implements the requirements
- Example: **Oven temperature control system**
 - design a function:
 - void LCD_display(String str)
 - int PID_control(int temp, int data[])

2.1. Embedded System Design Process

5. Test Specification:

- The device, equipment, environment for testing
- Prototype
- Testing process
- Example: **Oven temperature control system**
 - Voltage meter, temperature meter
 - Prototype: bread board
 - Testing process:
 - calibrate temperature sensor
 - check LCD, keypad
 - check output port (heater, fan)
 - verify PID control algorithm

2.2. System architecture development

- Partitioning the system into three parts:
 - Interface design
 - Hardware design
 - Software development
- Methodology
 - Block diagram
 - Waveform
 - Function description
 - Coding guideline

2.2. System architecture development

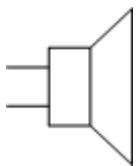
- Hardware block diagram
 - Use a rectangle for a hardware block



- Use an arrow for a connection



- Use a symbol for a special block



Speaker



Lamp



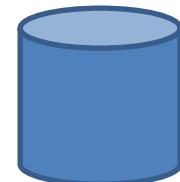
Energy



Network



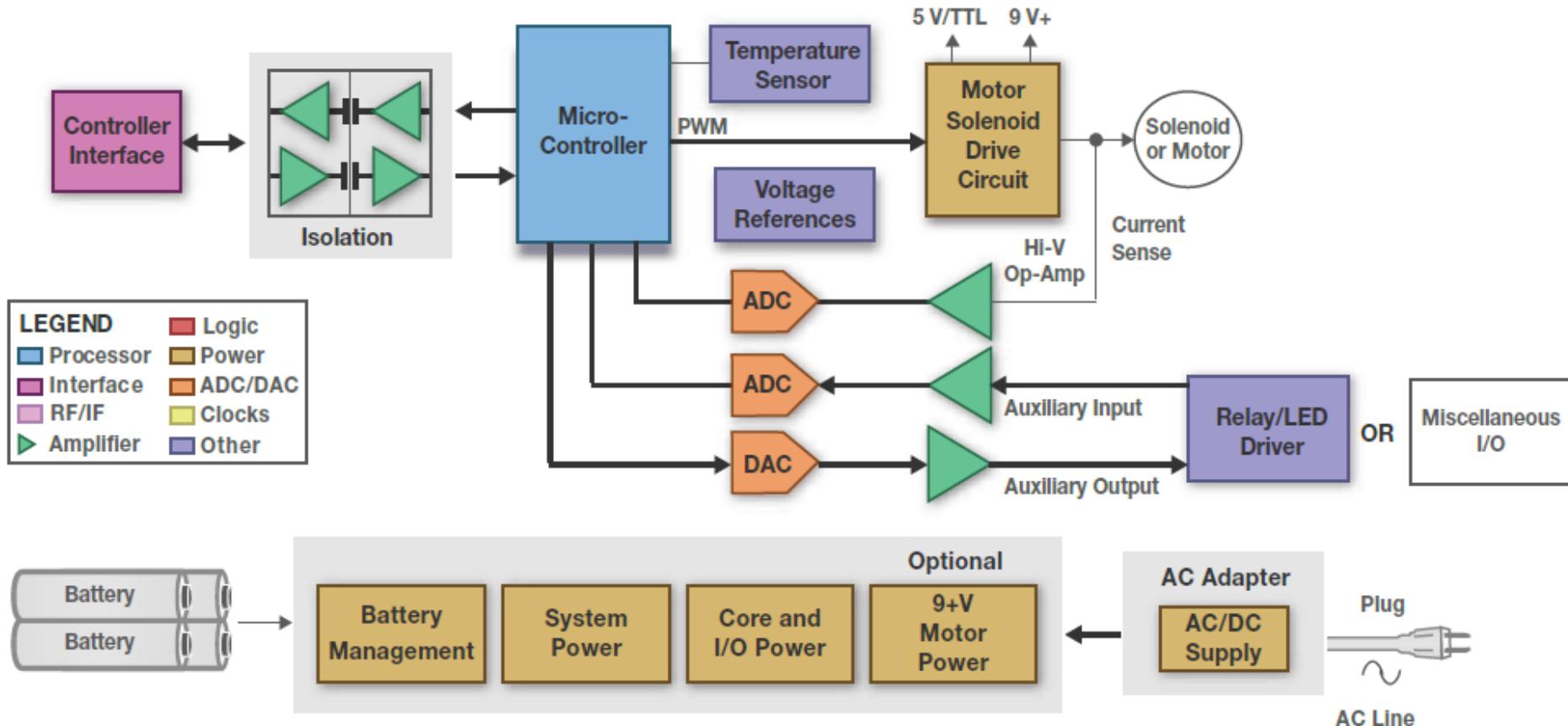
computer



Database

2.2. System architecture development

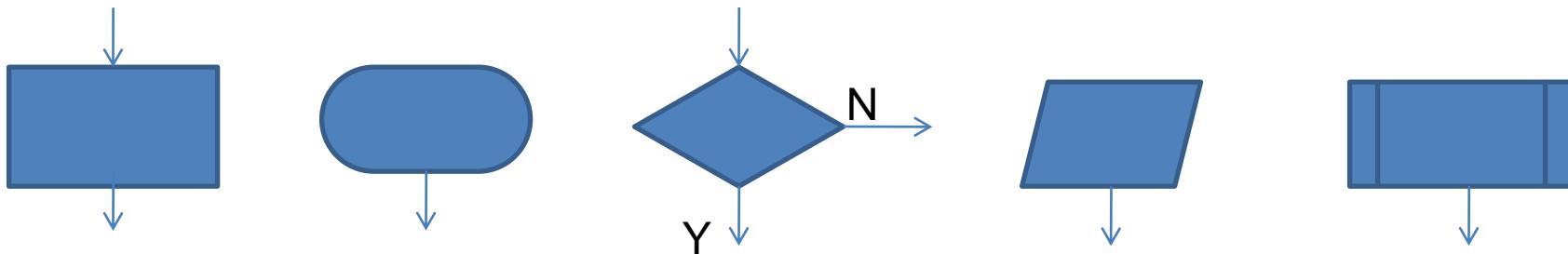
- Hardware block diagram - Example



MOTOR CONTROL

2.2. System architecture development

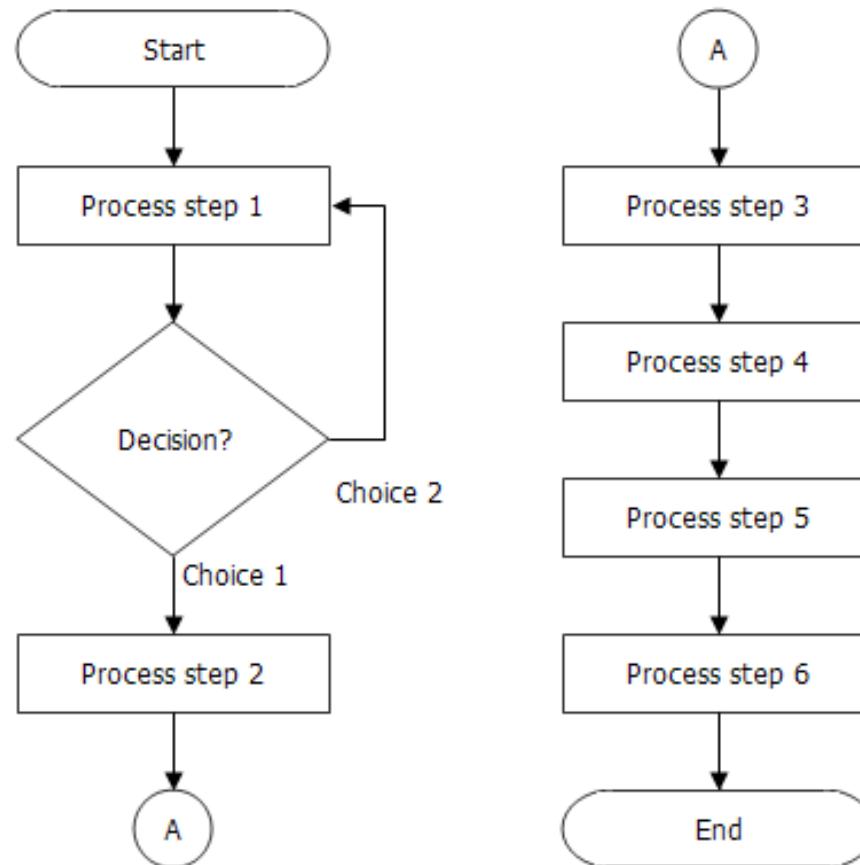
- Software diagram: flowchart, state diagram
 - Use a rectangle for a **process**
 - Use a rounded rectangle for a **terminator**
 - Use a diamond shape for a **decision**
 - Use a parallelogram for **data**
 - Use a rectangle with two vertical lines for **predefine process**
 - Use a circle for a **label**



2.2. System architecture development

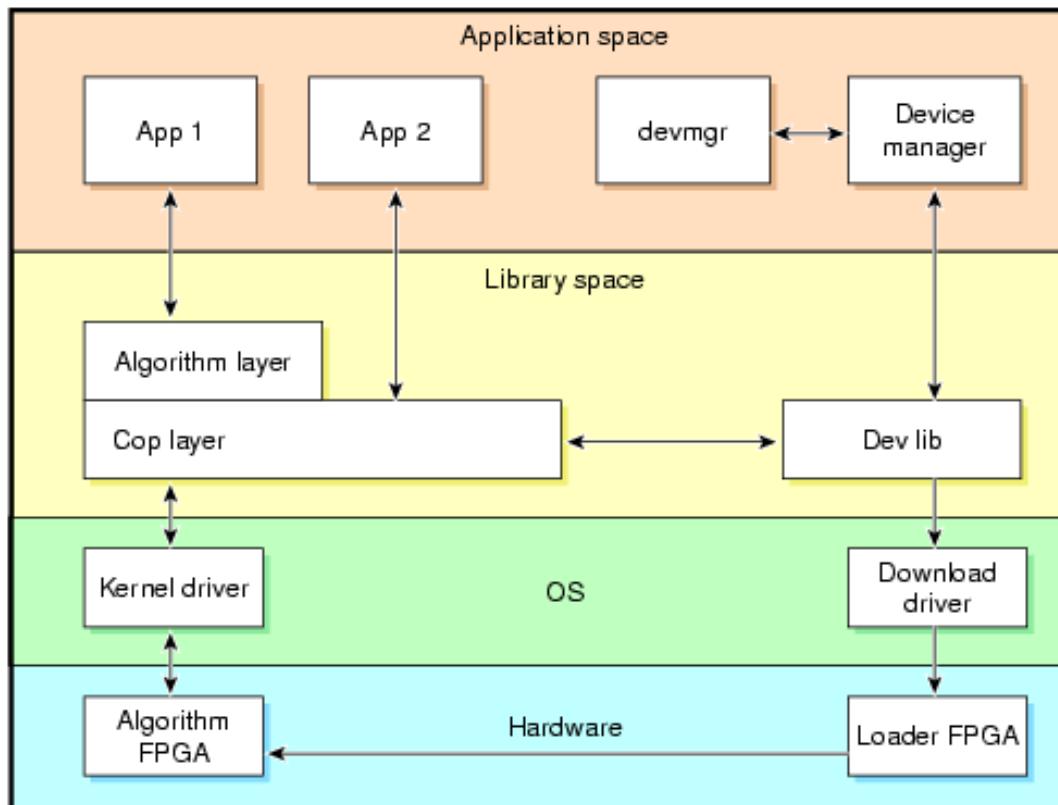
- Software block diagram - Example

Basic Flowchart



2.2. System architecture development

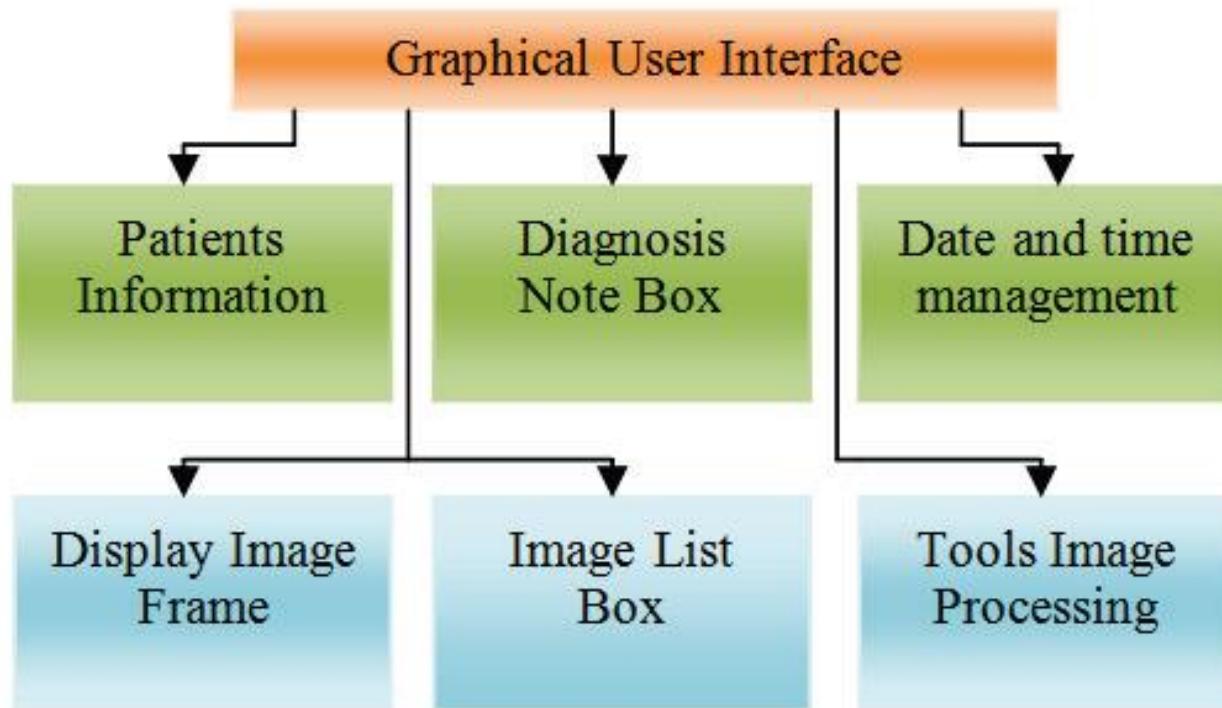
- Interface design
 - Show the connections between hardware and software
 - Show the connections with other systems
 - Show the interface with users



Examples

2.2. System architecture development

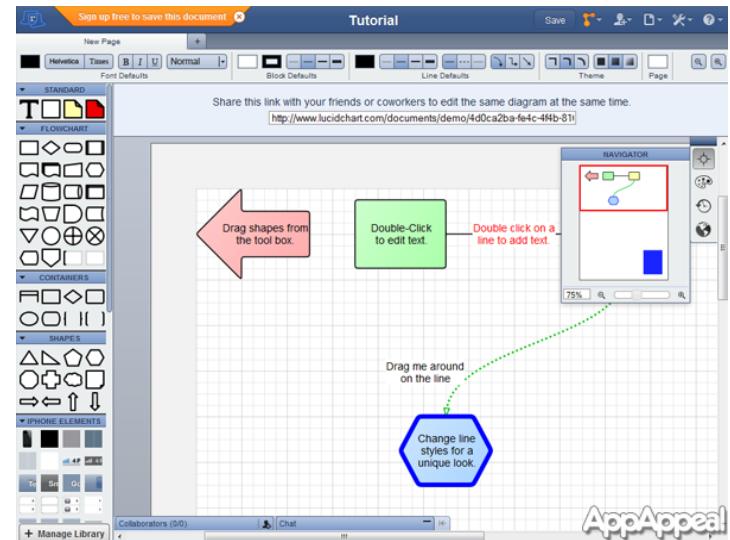
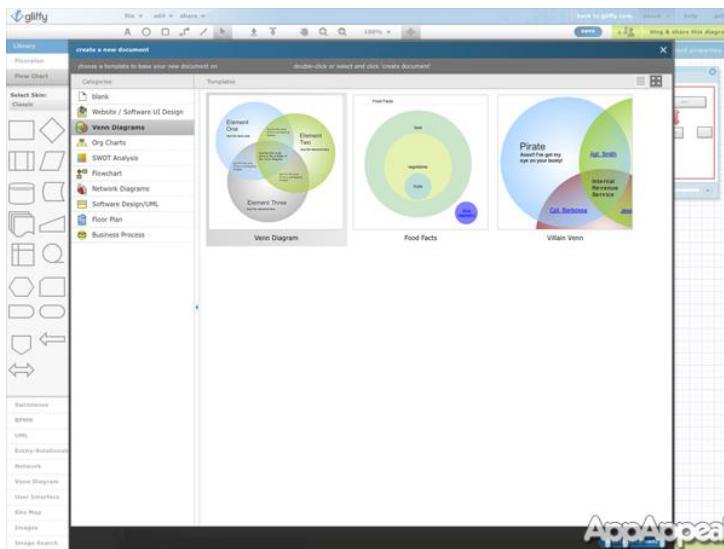
- Interface design – example :



Interface design for a dental camera system

2.2. System architecture development

- Diagram software tools
 - Microsoft Visio
 - Gliffy
 - Lucid chart



2.3. Integration and Testing

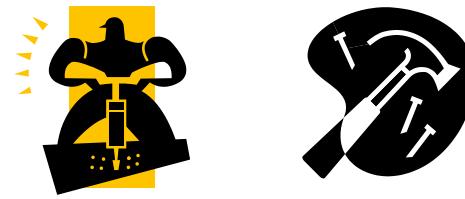
- Integration process
 - Configuration
 - Define configurations, parameters, versions
 - Hardware integration
 - integrate microprocessor, peripherals, sensors, actuators, user interfaces
 - Software integration
 - integrate OS, drivers, functions,
 - System integration

2.3. Integration and Test

- Testing process
 - Hardware testing
 - CPU
 - Peripheral interface
 - Sensor
 - Actuator
 - User interface
 - Software testing
 - Operating system, drivers
 - Applications/functions
 - User interface
 - Hard/soft co-testing
 - Test cases

3. System partitioning

- **System partitioning:** divide the system into three parts:
 - Hardware (HW): microcontrollers, memories, peripherals,
 - Software (SW): OS, program, application software
 - Interface: SW driver, HW interface, user interface



- **Alternative way:** divide the system into HW and SW
 - which functions should be performed in hardware, and which in software?
 - the more functions in software, the lower will be the product cost

3. System partitioning

- System partitioning examples: Oven temperature control system

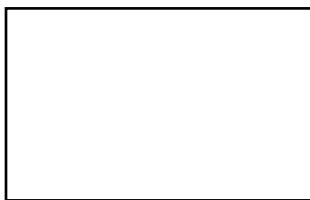
No.	Functions	Hardware	Software
1	Measure temperature	- Sensor LM35 - AD converter	-ADC reading function -Calculate temperature value from the sensor input value
2	Display temperature value	- LCD 1602	- LCD control function - Display function
3	Heating	- Heater	-Heater control function
4	Change the heating level	- Button	- Button reading function - Heating level setup function
5	Change heating time	- Button	- Timer function

3.2. Technology Selection

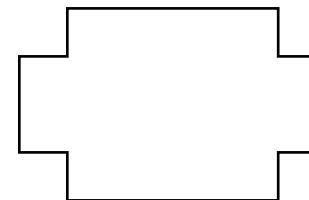
- Technology
 - A manner of accomplishing a task, especially using technical processes, methods, or knowledge
- Three key technologies for embedded systems
 - **Processor technology**: general-purpose, application-specific, single-purpose
 - **IC technology**: Full-custom, semi-custom, PLD
 - **Design technology**: Compilation/synthesis, libraries/IP, test/verification

3.2. Technology Selection

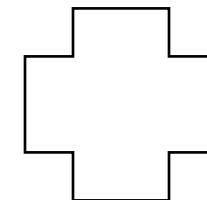
- Processor selection
 - number of IO pins required
 - interface required
 - memory requirements
 - number of interrupts required
 - real-time considerations
 - development environment
 - processing speed required



General-purpose
processor



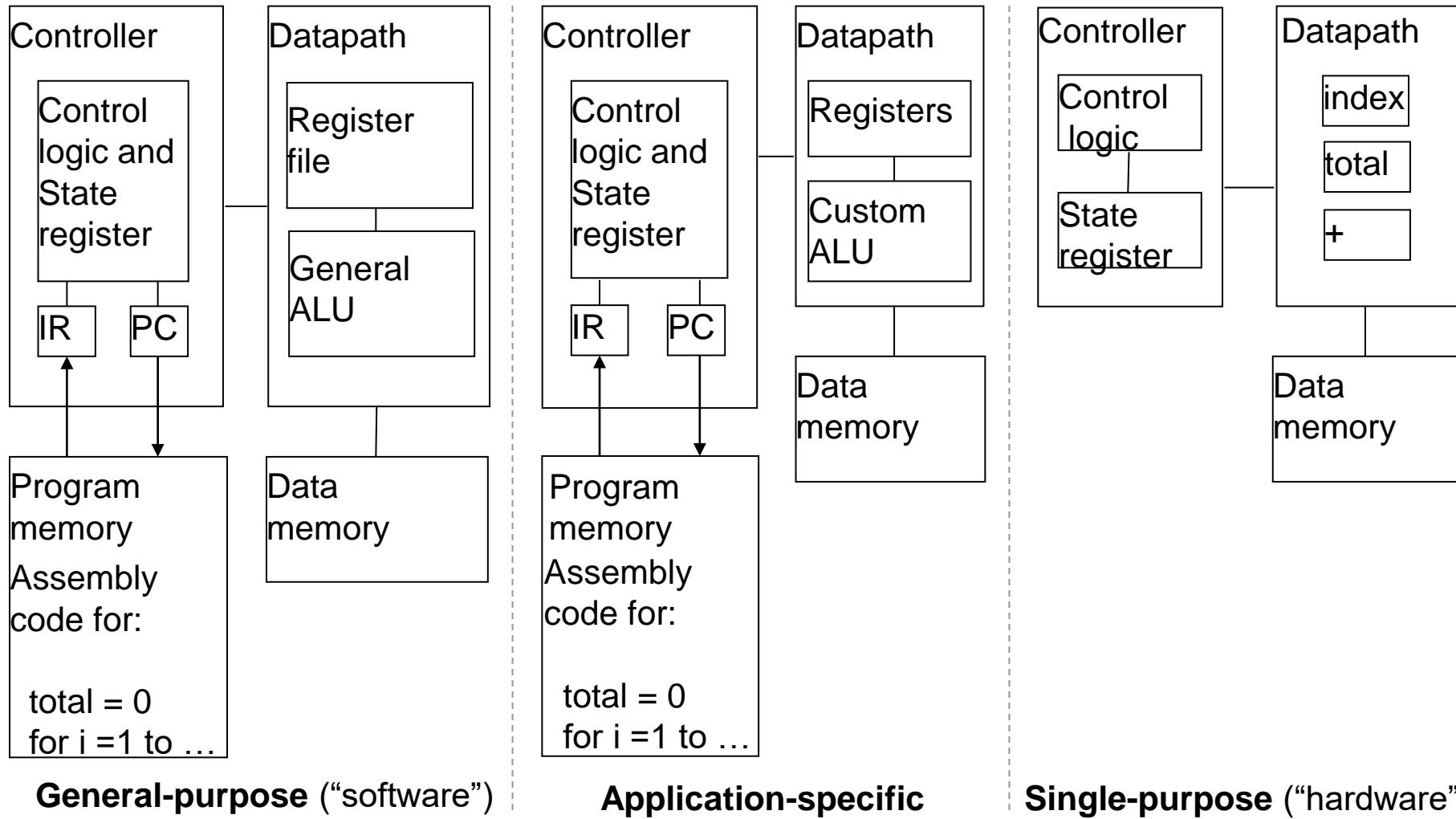
Application-specific
processor



Single-purpose
processor

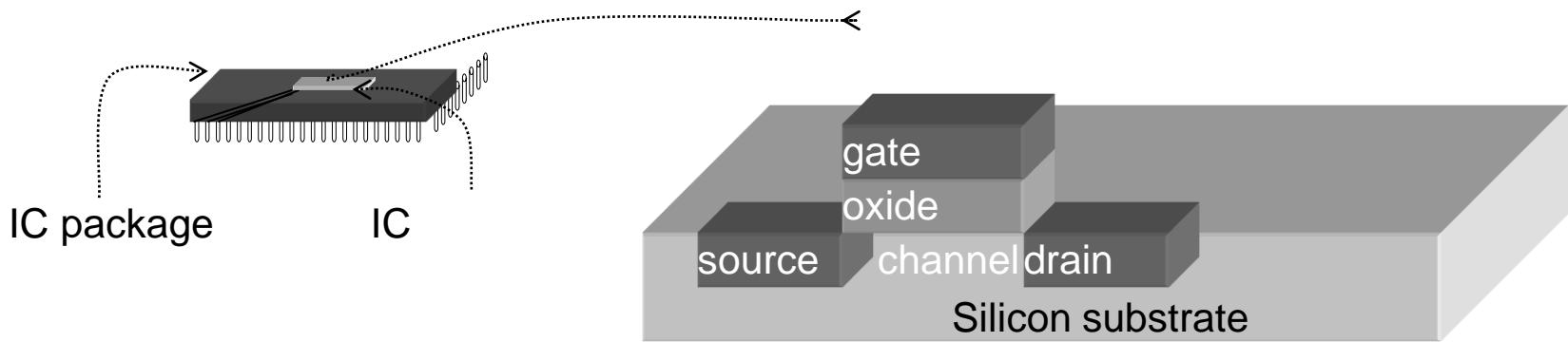
Processor technology

- “Processor” *not* equal to general-purpose processor



IC technology

- The manner in which a digital (gate-level) implementation is mapped onto an IC
 - IC: Integrated circuit, or “chip”
 - IC technologies differ in their customization to a design
 - IC’s consist of numerous layers (perhaps 10 or more)
 - IC technologies differ with respect to who builds each layer and when





Embedded System Design

Chapter 2: Microcontroller Series

1. Introduction to ARM processors
2. ARM Cortex-M3
3. ARM Cortex-M4
4. ARM Cortex-M3 & M4 Microcontroller Series
5. ARM programming

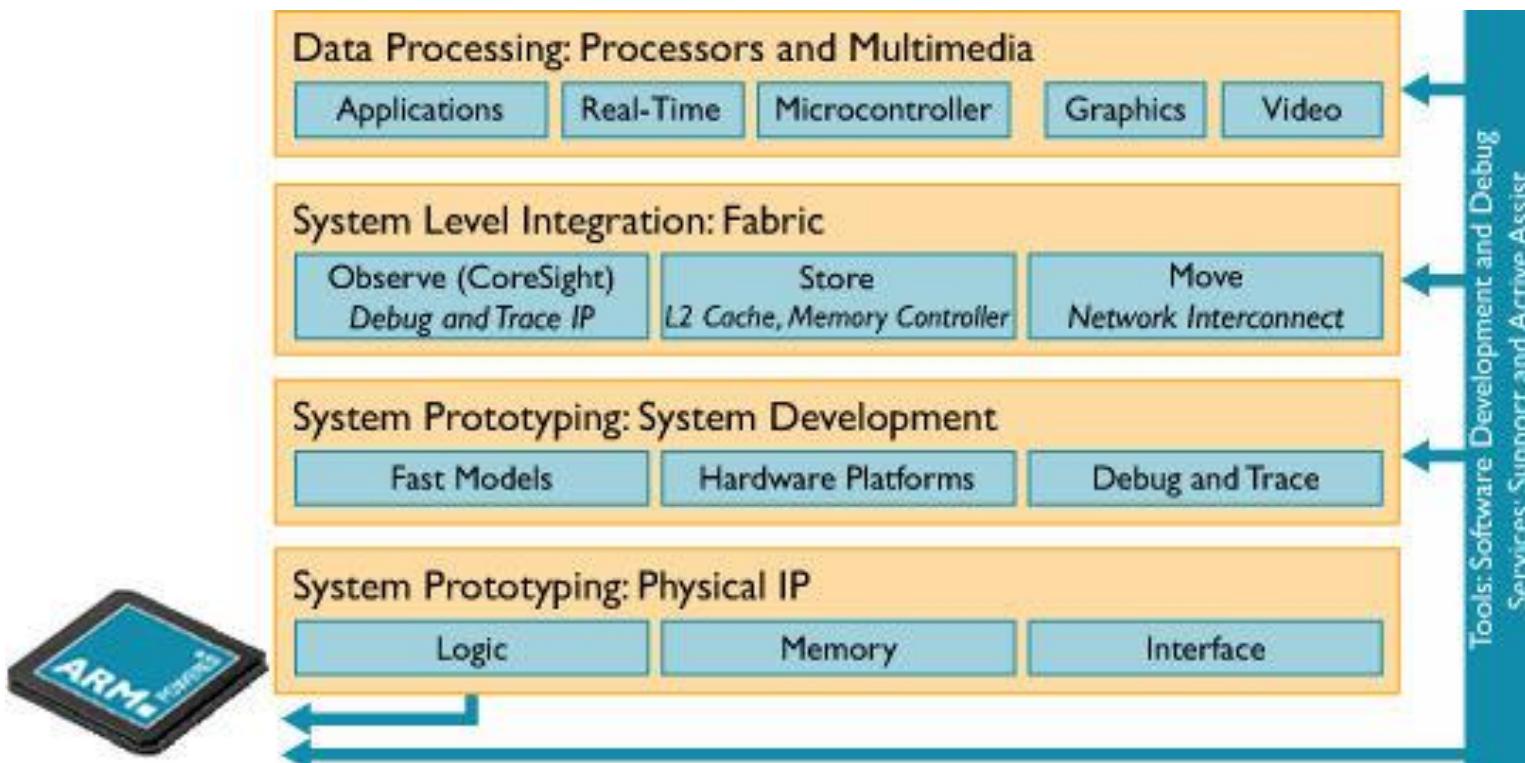
References

- Textbook
 - Joseph Yiu, “The Definitive Guide to the ARM Cortex-M3”, Elsevier Newnes, 2007
- Websites
 - www.arm.com www.thegioiic.com
 - www.ti.com www.arm.vn
 - www.ti.com www.tme.vn
 - www.nxp.com www.proe.vn



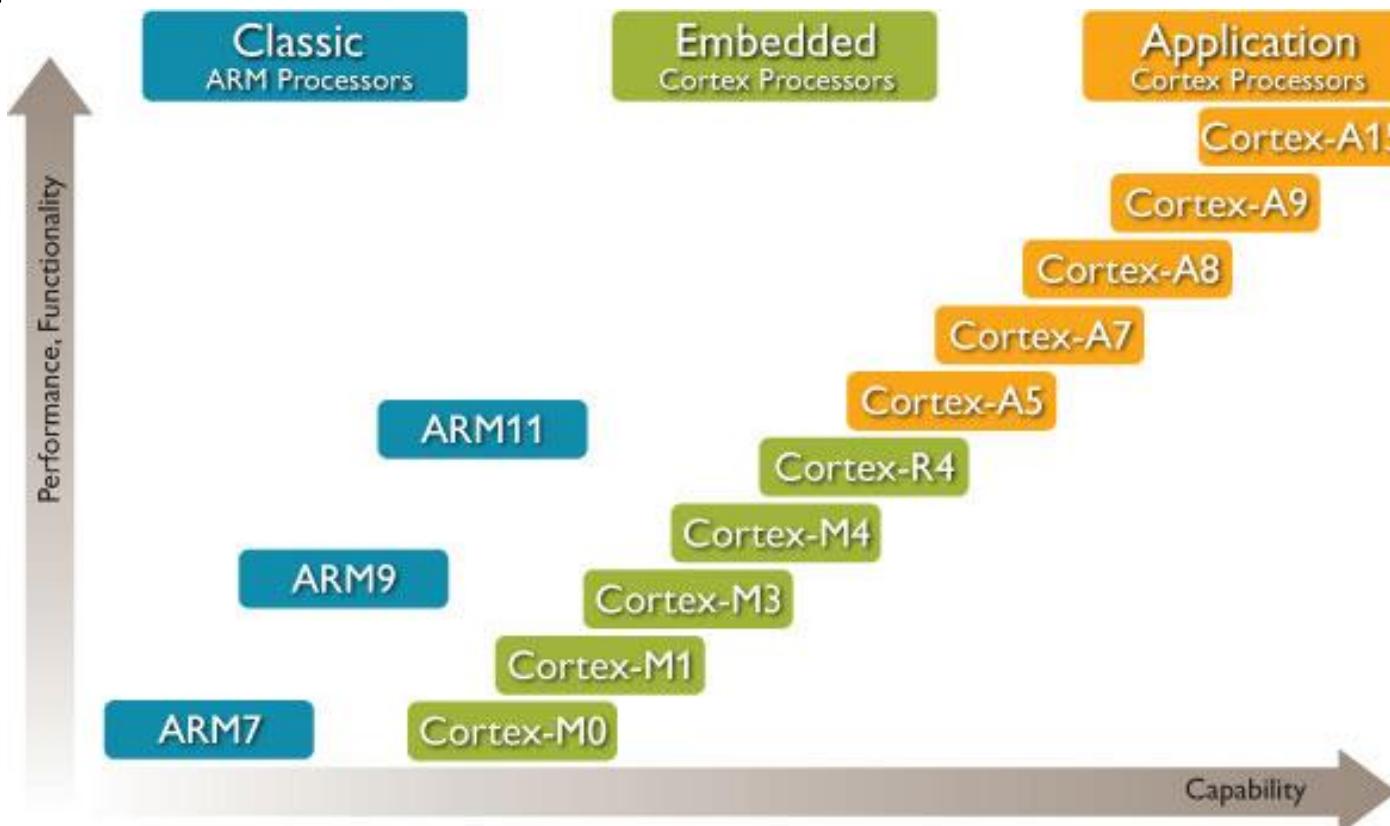
1. Introduction to ARM processors

- ARM (Advanced RISC Machine)
 - is the industry's leading provider of 32-bit embedded microprocessors
 - offering a wide range of processors that deliver high performance, industry leading power efficiency and reduced system cost



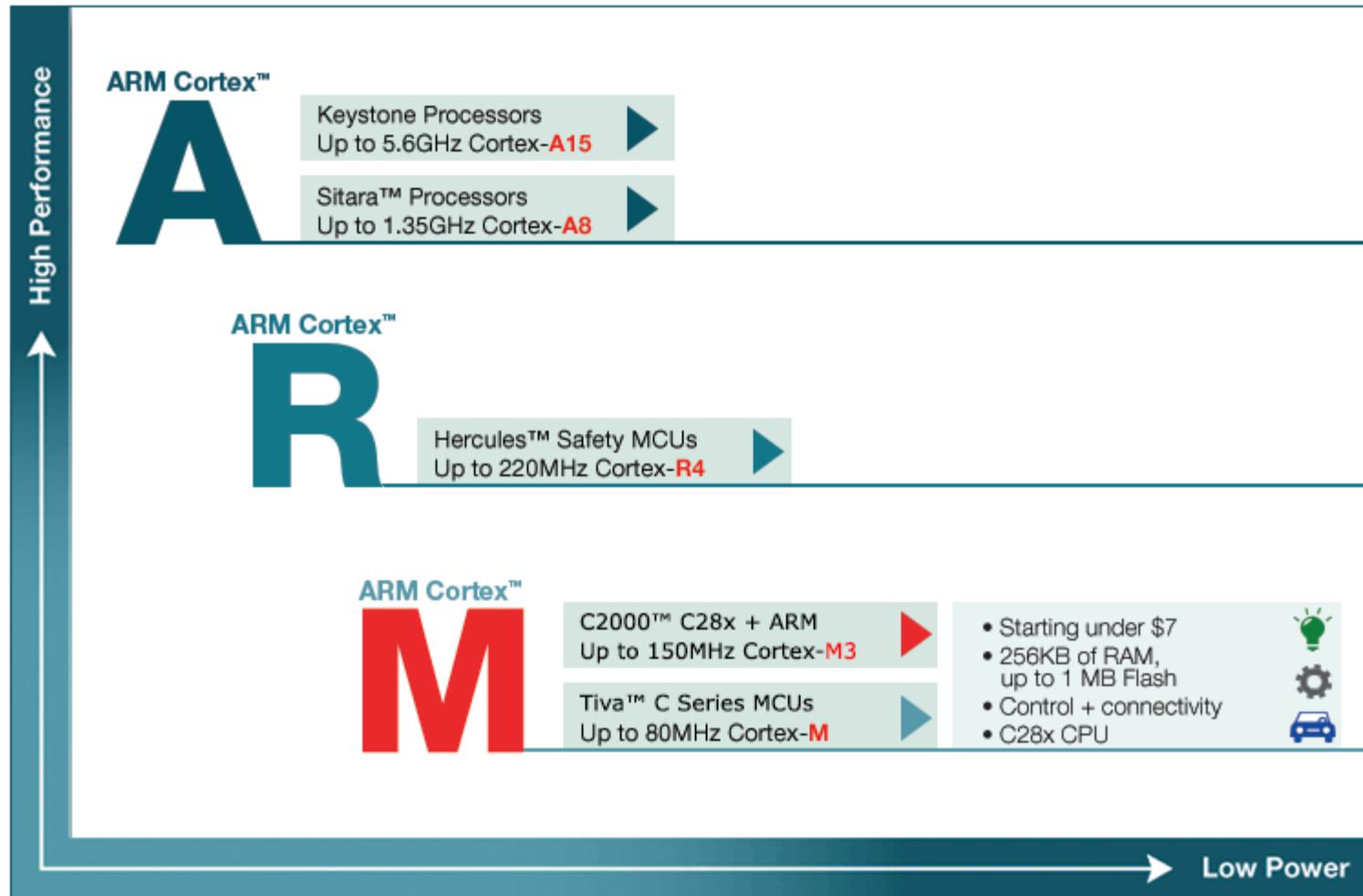
1. Introduction to ARM processors

- **Cortex™-A Series** - High performance processors for open Operating Systems
- **Cortex-R Series** - Exceptional performance for real-time applications
- **Cortex-M Series** - Cost-sensitive solutions for deterministic microcontroller applications

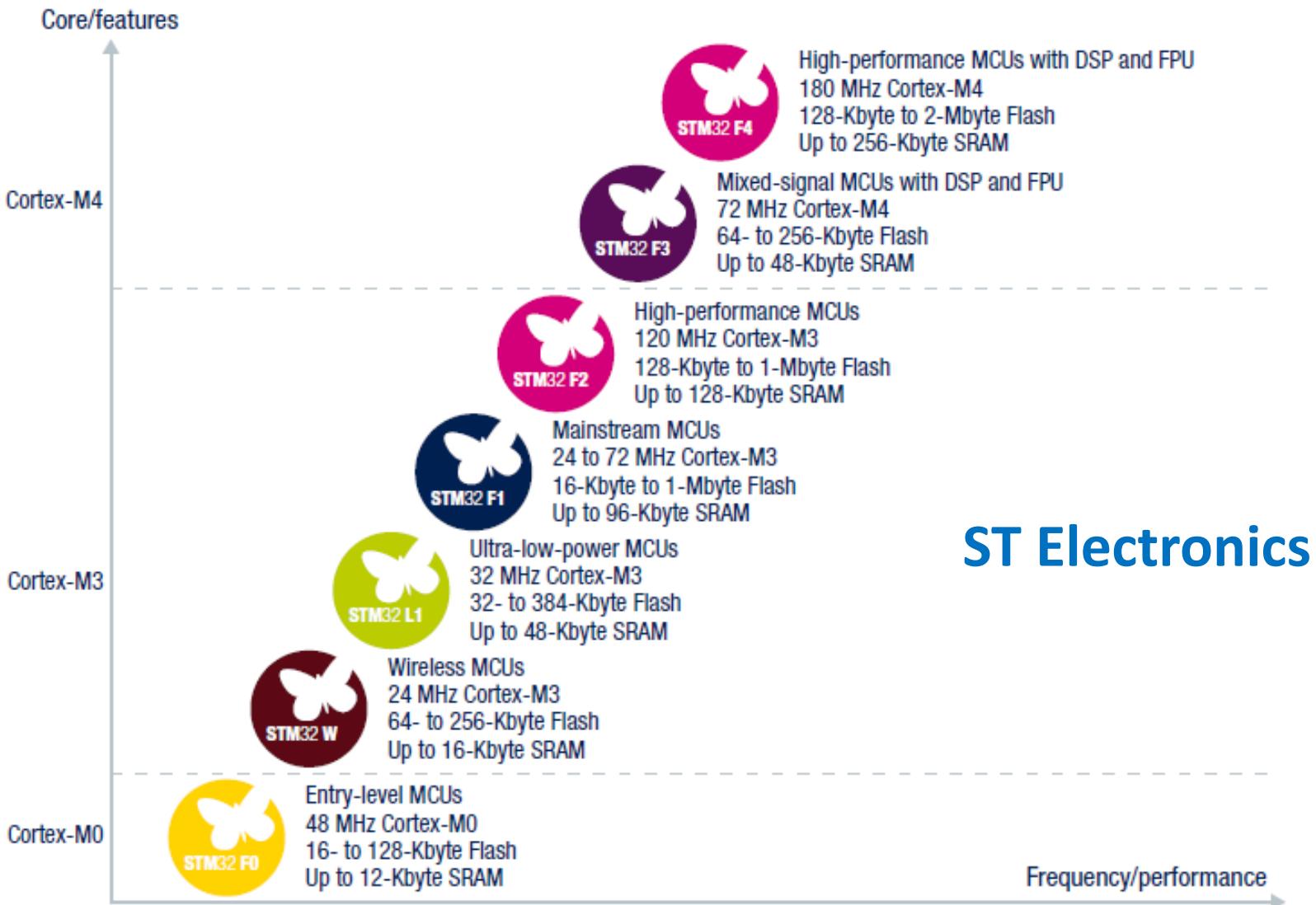


1. Introduction to ARM processors

- TI's ARM microcontroller overview

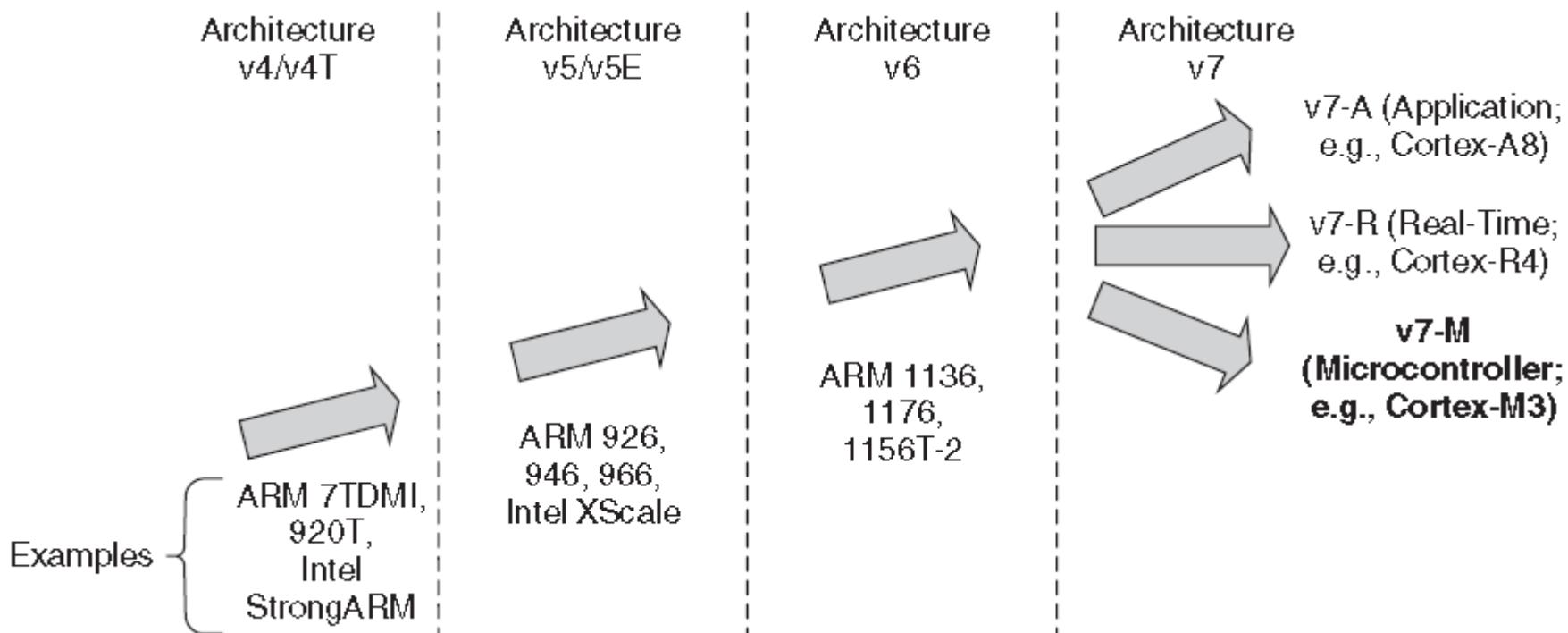


1. Introduction to ARM processors



1. Introduction to ARM processors

- The Cortex processor families are the first products developed on architecture v7
- Cortex-M3 processor is based on one profile of the v7 architecture



1. Introduction to ARM processors

- Cortex-M Series:
 - **Easy to use**: Global standard across multiple vendors, code compatibility, unified tools and OS support
 - **Low cost**: smaller code, high density instruction set, small memory requirement
 - **High performance**: deliver more performance per MHz, enable richer features at lower power
 - **Energy efficiency**: run at lower MHz or with shorter activity periods
- Cortex-M Series applications
 - Microcontrollers
 - Mixed signal devices
 - Smart sensors
 - Automotive body electronics and airbags



2. ARM Cortex-M3

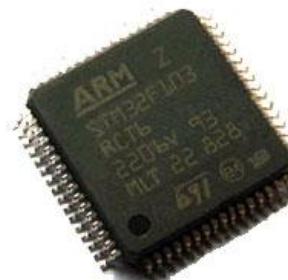
- 32-bit embedded processor
- Improved code density
- Enhanced determinism, quick interrupts
- Low power consumption
- Lower-cost solutions (less than US\$1)
- Wide choice of development tools



LPC1754FBD80



AT91SAM7S64-AU



STM32F103RCT6



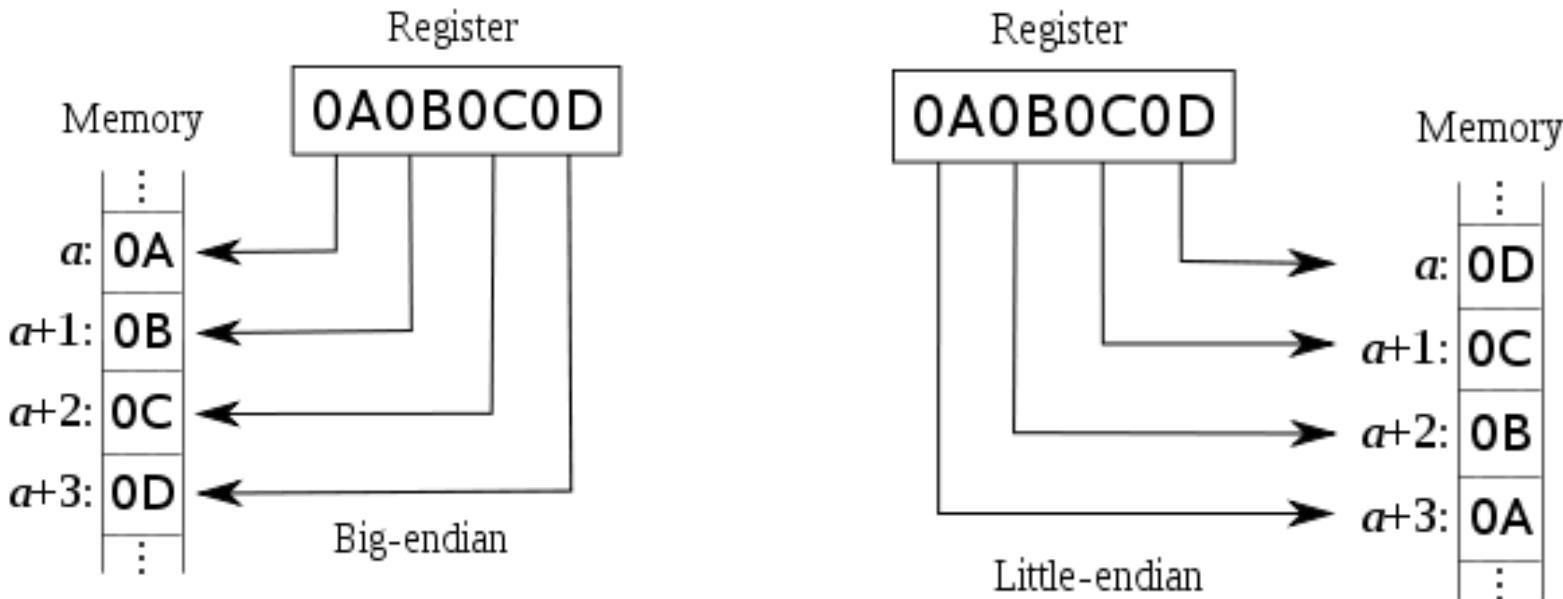
LM3S3749

2.1 ARM Cortex-M3 - Architecture

- 32-bit microprocessor
 - 32-bit data path
 - 32-bit register bank
 - 32-bit memory interface
- Harvard architecture
 - 3-stage pipeline
 - separate instruction bus and data bus
 - share the same memory space, difference length of code and data
- Interrupts
 - 1 to 240 physical interrupts, plus NMI
 - 12 cycle interrupt latency
- Instruction Set
 - Thumb (entire)
 - Thumb-2 (entire)

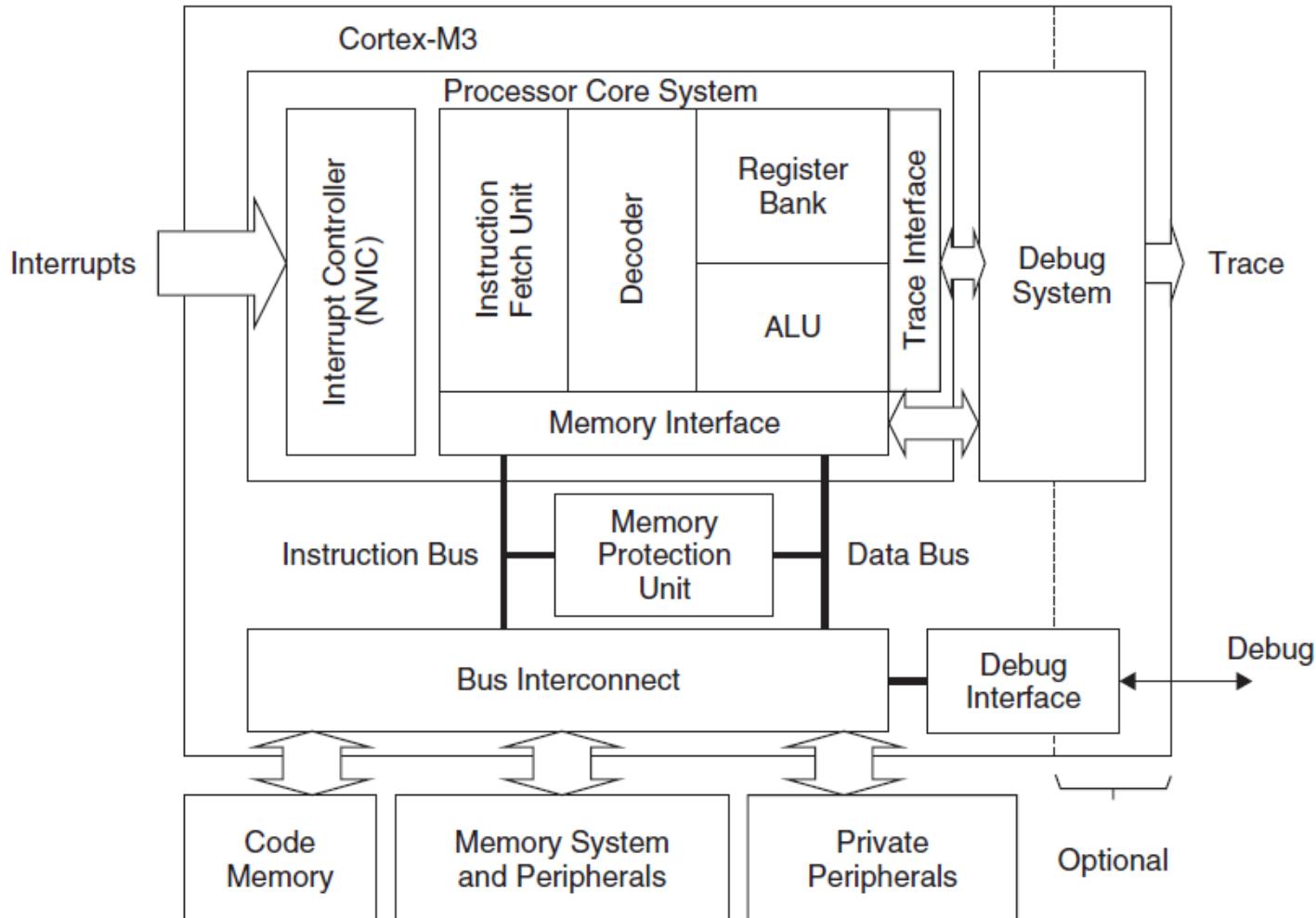
2.1 ARM Cortex-M3 - Architecture

- Endian mode
 - both little Endian and big Endian are supported
 - In most cases, Cortex-M3-based microcontrollers will be little endian.
 - Instruction fetches are always in little endian
 - PPB accesses are always in little endian.

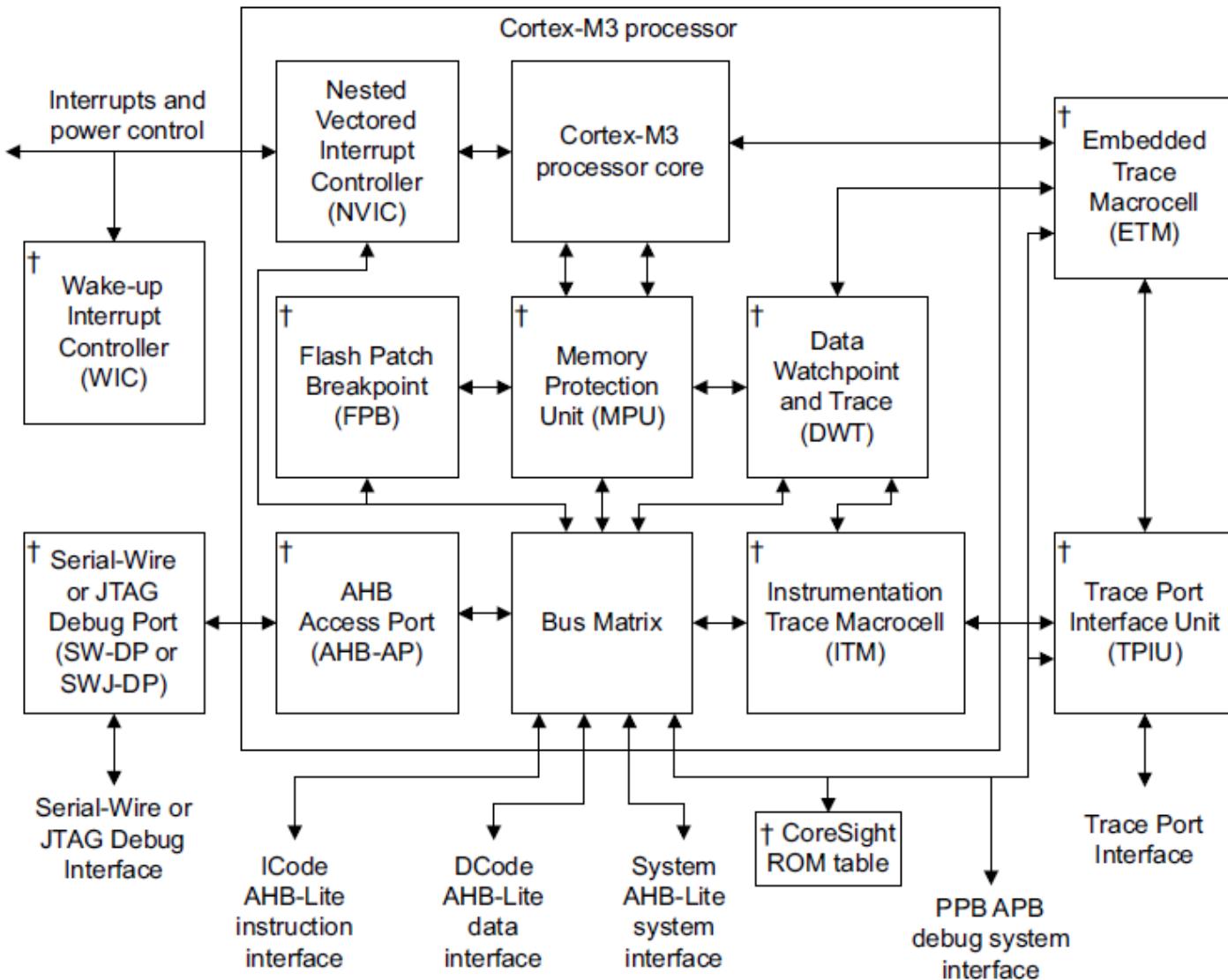


2.1 ARM Cortex-M3 - Architecture

Harvard architecture



2.1 ARM Cortex-M3 – Architecture

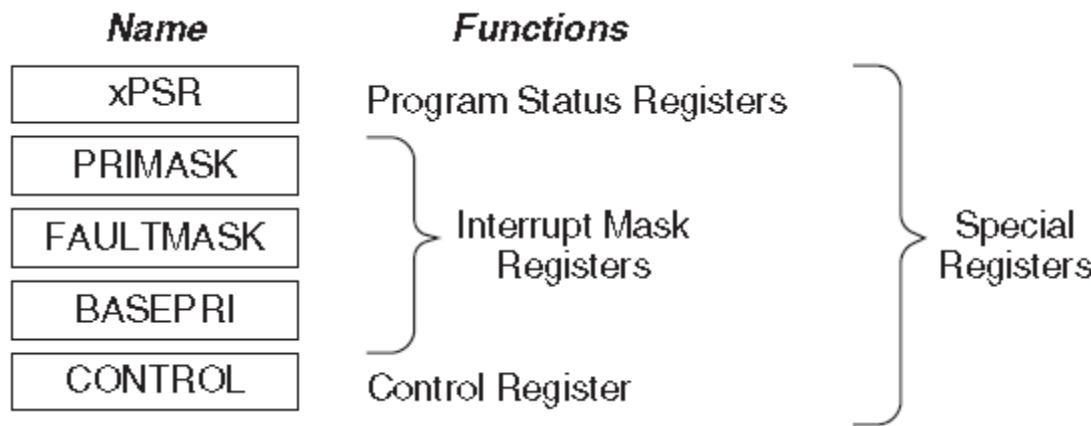


2.2 ARM Cortex-M3 - Registers

- **Registers**
 - R0 – R12: general purpose registers
 - R13: stack pointers
 - R14: link register (store return address)
 - R15: program counter
- **Special Registers**
 - Program Status Registers (PSRs)
 - Interrupt Mask Registers (PRIMASK, FAULTMASK, BASEPRI)
 - Control Register (CONTROL)

2.2 ARM Cortex-M3 - Registers

- Special registers in the Cortex-M3



Register	Function
xPSR	Provide ALU flags (zero flag, carry flag), execution status, and current executing interrupt number
PRIMASK	Disable all interrupts except the nonmaskable interrupt (NMI) and HardFault
FAULTMASK	Disable all interrupts except the NMI
BASEPRI	Disable all interrupts of specific priority level or lower priority level
CONTROL	Define privileged status and stack pointer selection

2.2 ARM Cortex-M3 - Registers

- **Program Status Registers**

- Application PSR (APSR)
- Interrupt PSR (IPSR)
- Execution PSR (EPSR)

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q											
IPSR																Exception Number
EPSR						ICI/IT	T			ICI/IT						

Figure 3.3 Program Status Registers (PSRs) in the Cortex-M3

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT						Exception Number

Figure 3.4 Combined Program Status Registers (xPSR) in the Cortex-M3

2.2 ARM Cortex-M3 - Registers

- **The Control register**

- has two bits
- used to define the privilege level and the stack pointer selection.

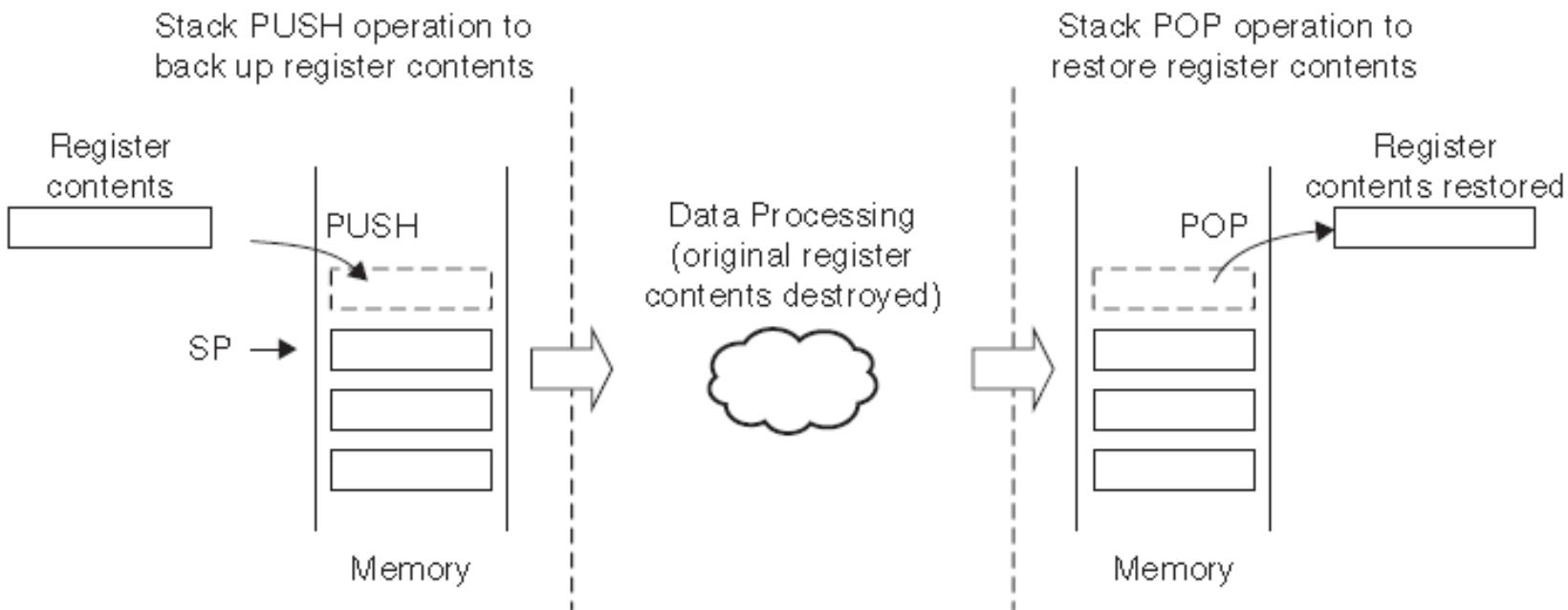
Bit	Function
CONTROL[1]	Stack status: 1 = Alternate stack is used 0 = Default stack (MSP) is used If it is in the Thread or base level, the alternate stack is the PSP. There is no alternate stack for handler mode, so this bit must be zero when the processor is in handler mode.
CONTROL[0]	0 = Privileged in Thread mode 1 = User state in Thread mode If in handler mode (not Thread mode), the processor operates in privileged mode.

2.2 ARM Cortex-M3 - Registers

Name	Type ^a	Required privilege ^b	Reset value	Description
R0-R12	RW	Either	Unknown	<i>General-purpose registers on page 2-4</i>
MSP	RW	Privileged	See description	<i>Stack Pointer on page 2-4</i>
PSP	RW	Either	Unknown	<i>Stack Pointer on page 2-4</i>
LR	RW	Either	0xFFFFFFFF	<i>Link Register on page 2-4</i>
PC	RW	Either	See description	<i>Program Counter on page 2-4</i>
PSR	RW	Privileged	0x01000000	<i>Program Status Register on page 2-4</i>
ASPR	RW	Either	Unknown	<i>Application Program Status Register on page 2-5</i>
IPSR	RO	Privileged	0x00000000	<i>Interrupt Program Status Register on page 2-6</i>
EPSR	RO	Privileged	0x01000000	<i>Execution Program Status Register on page 2-6</i>
PRIMASK	RW	Privileged	0x00000000	<i>Priority Mask Register on page 2-8</i>
FAULTMASK	RW	Privileged	0x00000000	<i>Fault Mask Register on page 2-8</i>
BASEPRI	RW	Privileged	0x00000000	<i>Base Priority Mask Register on page 2-9</i>
CONTROL	RW	Privileged	0x00000000	<i>CONTROL register on page 2-9</i>

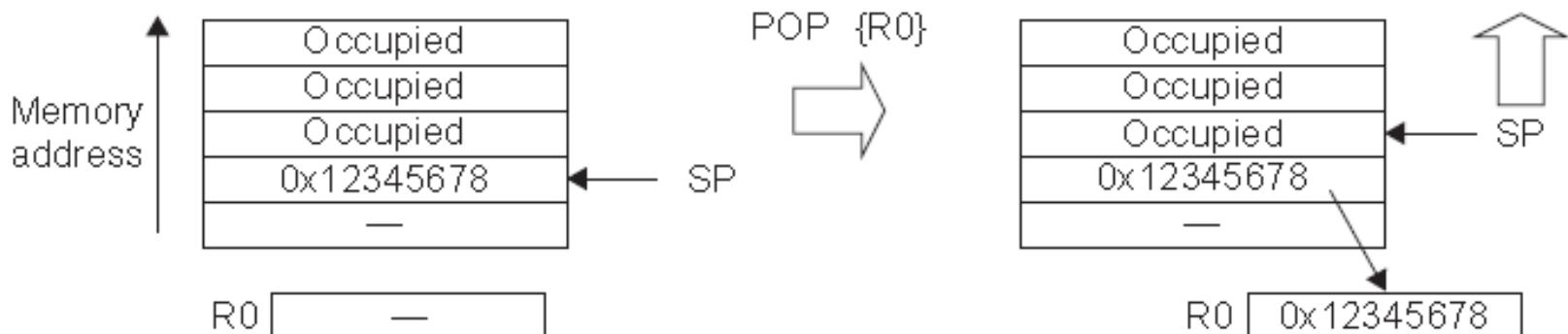
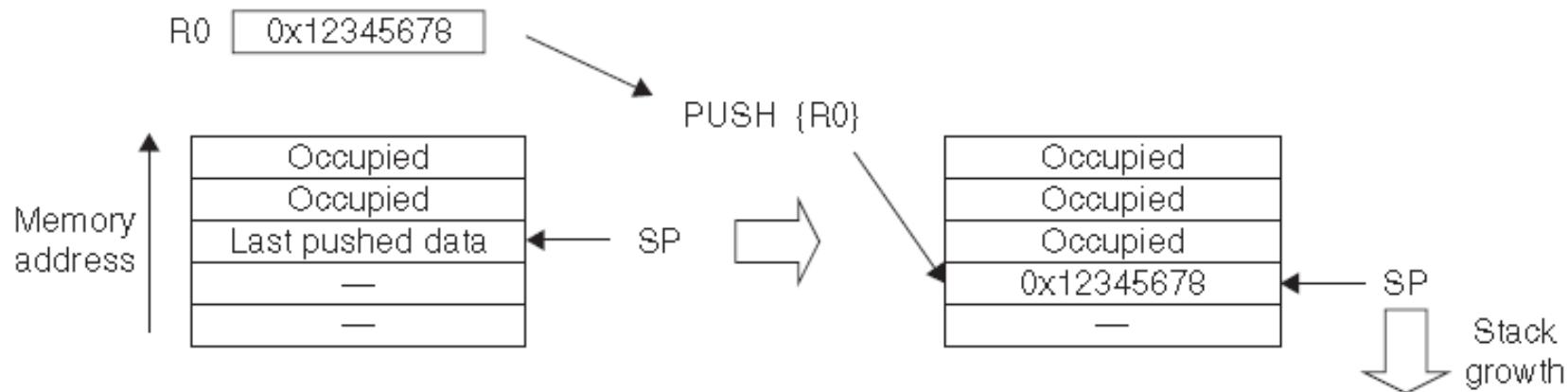
2.3 ARM Cortex-M3 - Stack Pointer

- Cortex-M3 processor has two stack pointers
 - Main Stack Pointer (MSP): used by the OS kernel, exception handlers
 - Process Stack Pointer (PSP): Used by the base-level application code
- When using the register name R13, you can only access the current stack pointer



2.3 ARM Cortex-M3 – PUSH & POP

- The stack pointer (SP) points to the last data pushed to the stack memory,
- The SP decrements before a new PUSH operation



2.4 ARM Cortex-M3 – Operation Modes

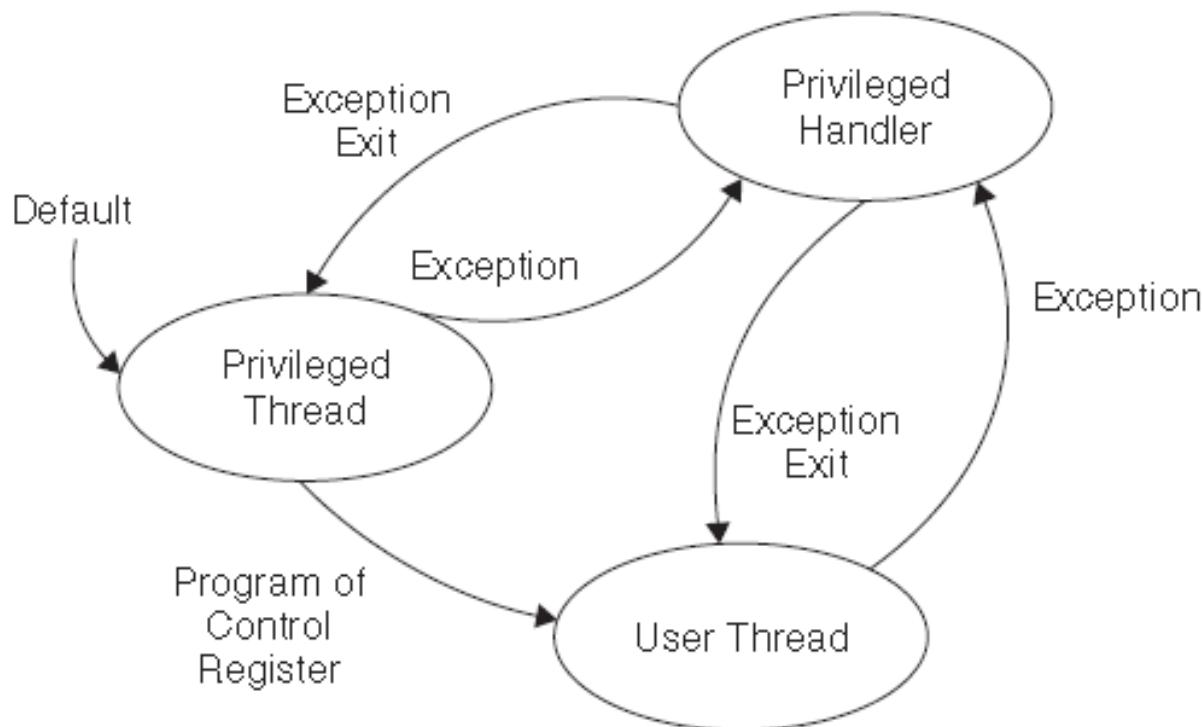
Cortex-M3 has 2 modes and 2 privilege levels

- Thread mode: Used to execute application software
- Handler mode: Used to handle exceptions
- Unprivileged:
 - has limited access to the MSR and MRS instructions, and cannot use the CPS instruction
 - cannot access the system timer, NVIC, or system control block
 - might have restricted access to memory or peripherals
- Privileged: can use all the instructions and has access to all resources

	<i>Privileged</i>	<i>User</i>
<i>When running an exception</i>	Handle Mode	
<i>When running main program</i>	Thread Mode	Thread Mode

2.4 ARM Cortex-M3 – Operation Modes

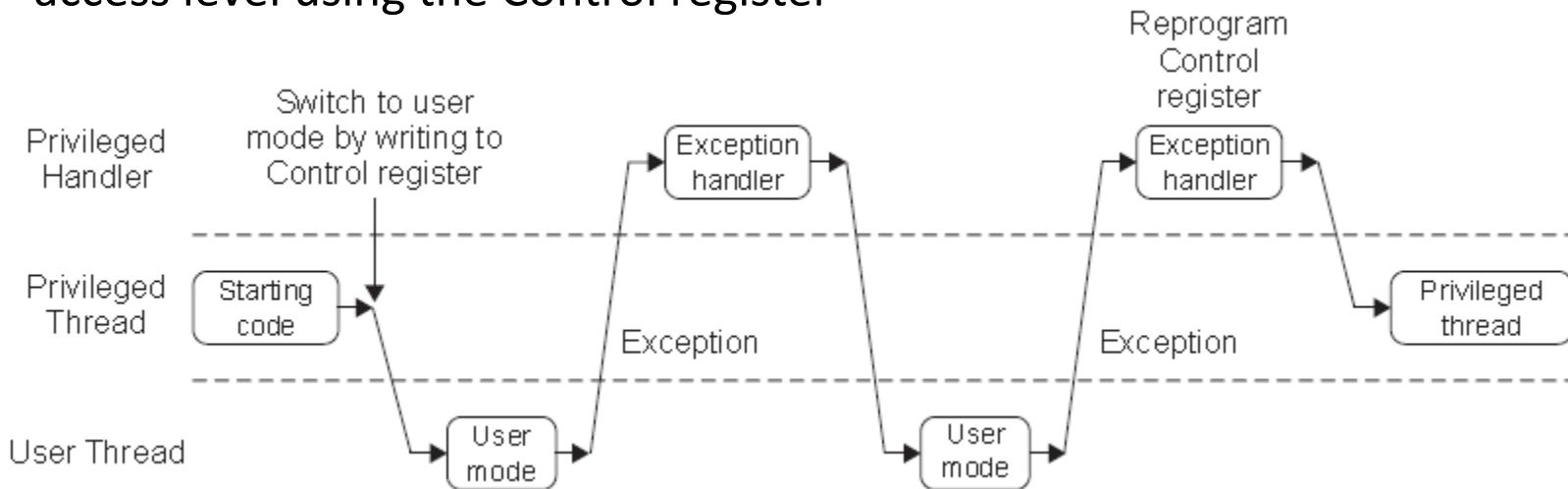
- Mode transitions



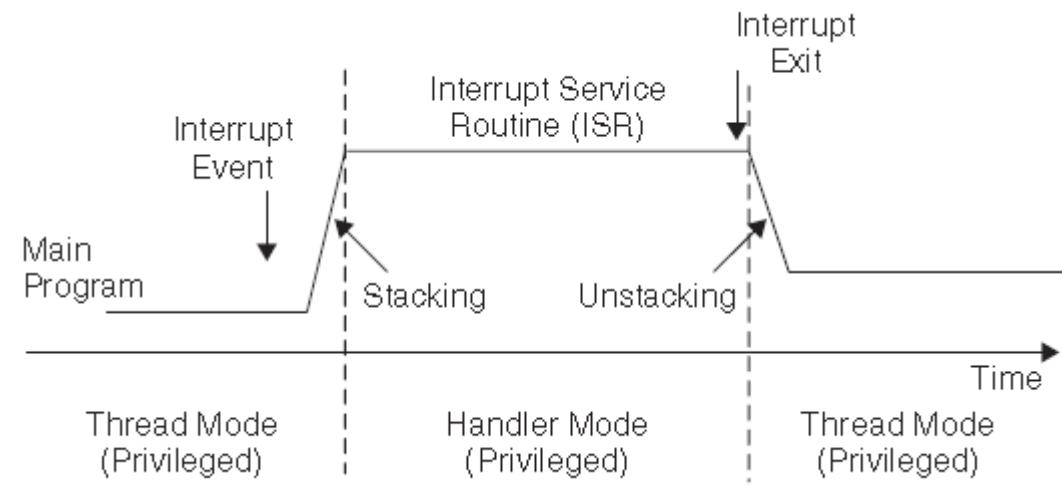
Allowed Operation Mode Transitions

2.2 ARM Cortex-M3 – Operation Modes

- Software in a privileged access level can switch the program into the user access level using the Control register



- Switching Processor Mode at Interrupt

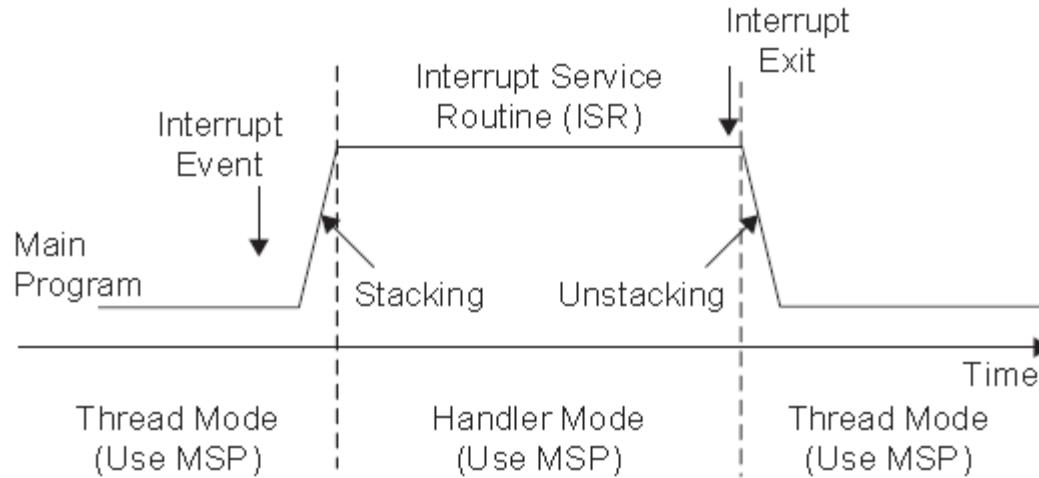


2.5 ARM Cortex-M3 - Interrupt

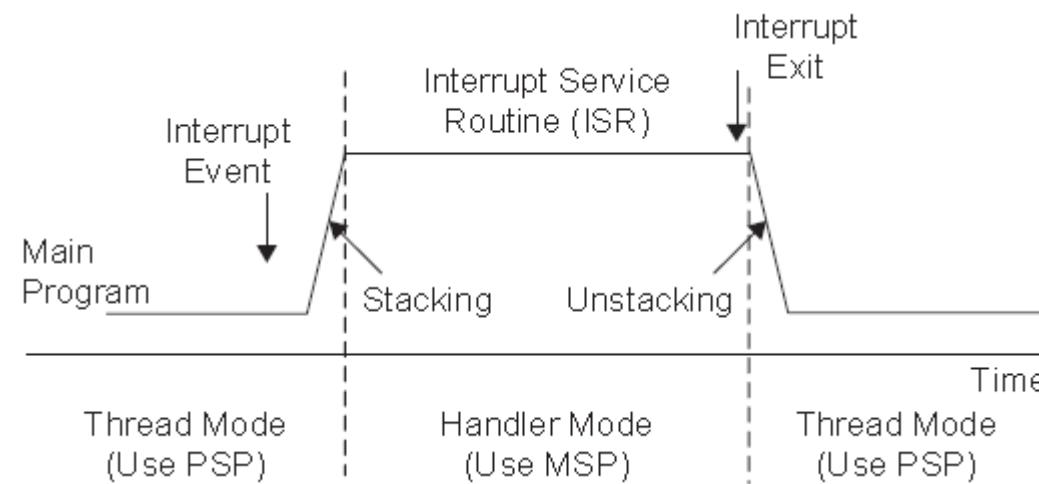
- The Built-In Nested Vectored Interrupt Controller (NVIC) provides a number of features:
 - **Nested interrupt support:** different priority levels
 - **Vectored interrupt support:** interrupt vector table in memory
 - **Dynamic priority changes support:** Priority levels of interrupts can be changed during run time.
 - **Reduction of interrupt latency:** automatic saving and restoring some register contents, reducing delay in switching
 - **Interrupt masking:** Interrupts and system exceptions can be masked

2.5 ARM Cortex-M3 - Interrupt

- Control[1] = 0: Both Thread Level and Handler Use Main Stack

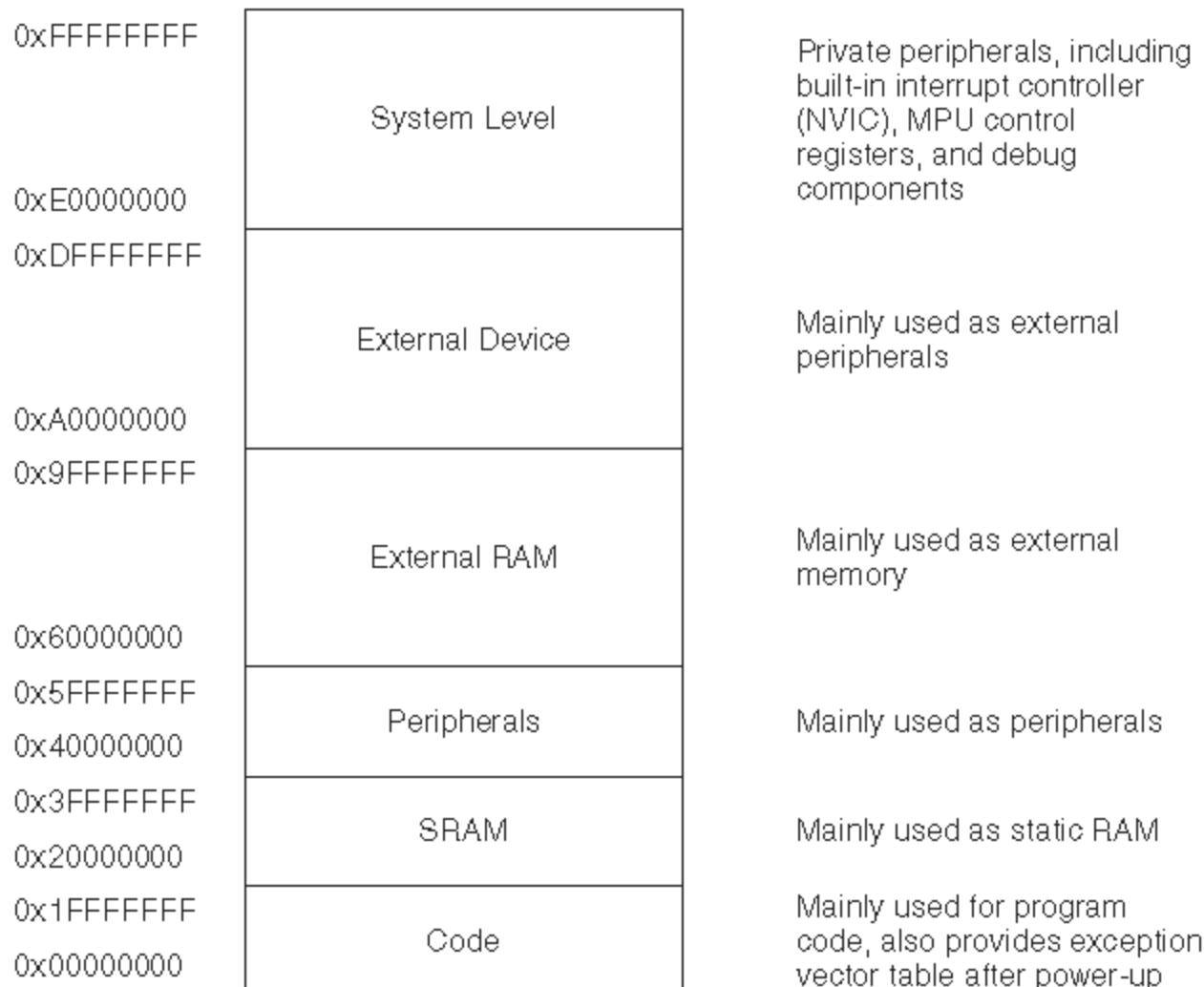


- Control[1] = 1: Thread Level Uses Process Stack and Handler Uses Main Stack



2.6 ARM Cortex-M3 – Memory Map

- The 4 GB memory space can be divided into the ranges shown in Figure:



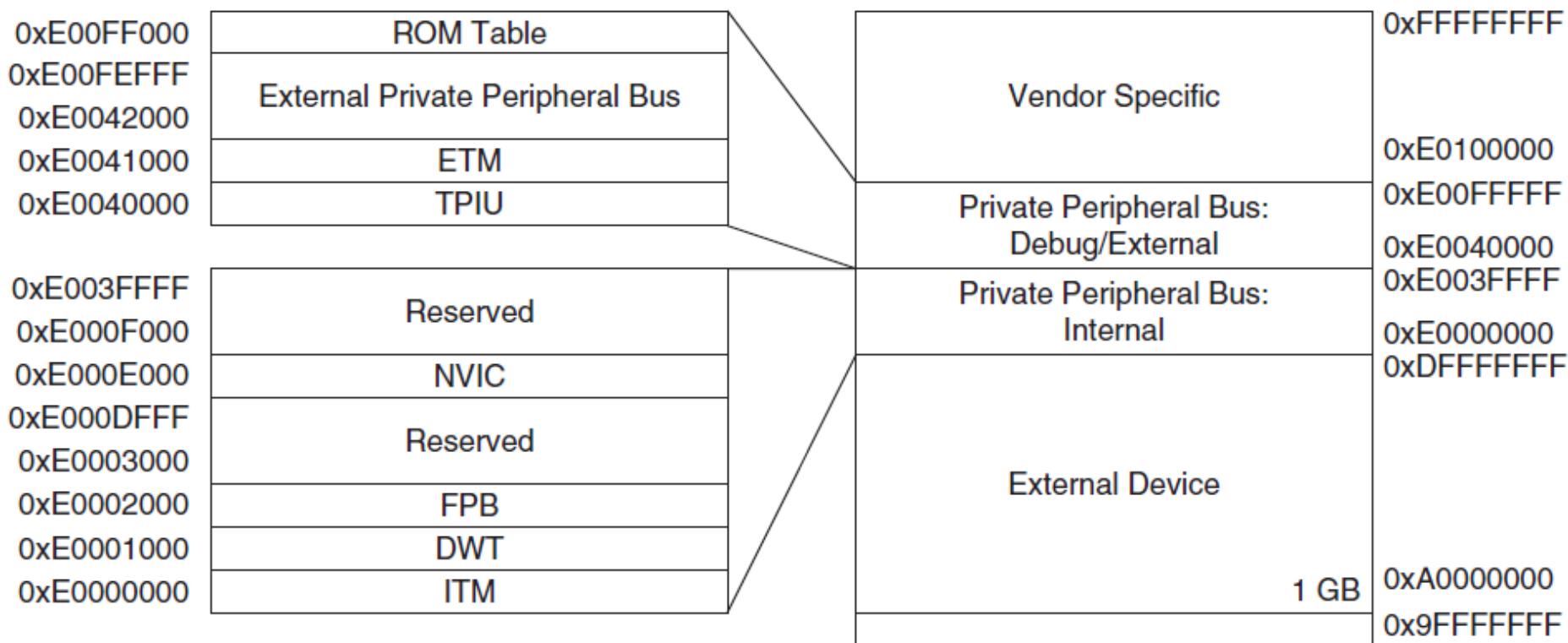
2.6 ARM Cortex-M3 – Memory Access

Address range	Memory region	Memory type ^a	XN ^a	Description
0x00000000- 0x1FFFFFFF	Code	Normal	-	Executable region for program code. You can also put data here.
0x20000000- 0x3FFFFFFF	SRAM	Normal	-	Executable region for data. You can also put code here. This region includes bit band and bit band alias areas, see Table 2-13 on page 2-16.
0x40000000- 0x5FFFFFFF	Peripheral	Device	XN	This region includes bit band and bit band alias areas, see Table 2-14 on page 2-16.
0x60000000- 0x9FFFFFFF	External RAM	Normal	-	Executable region for data.
0xA0000000- 0xDFFFFFFF	External device	Device	XN	External Device memory.
0xE0000000- 0xE00FFFFF	Private Peripheral Bus	Strongly-ordered	XN	This region includes the NVIC, System timer, and system control block.
0xE0100000- 0xFFFFFFFF	Device	Device	XN	Implementation-specific.

Execute Never (XN) Means the processor prevents instruction accesses.

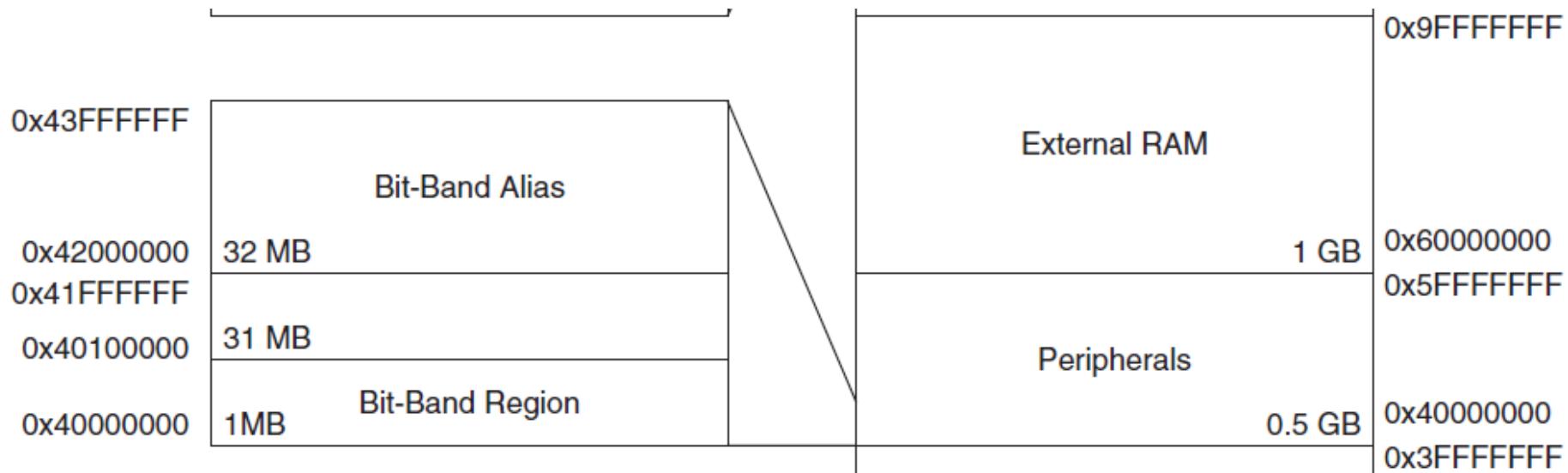
2.6 ARM Cortex-M3 – Memory Access

- System level memory
- External device memory



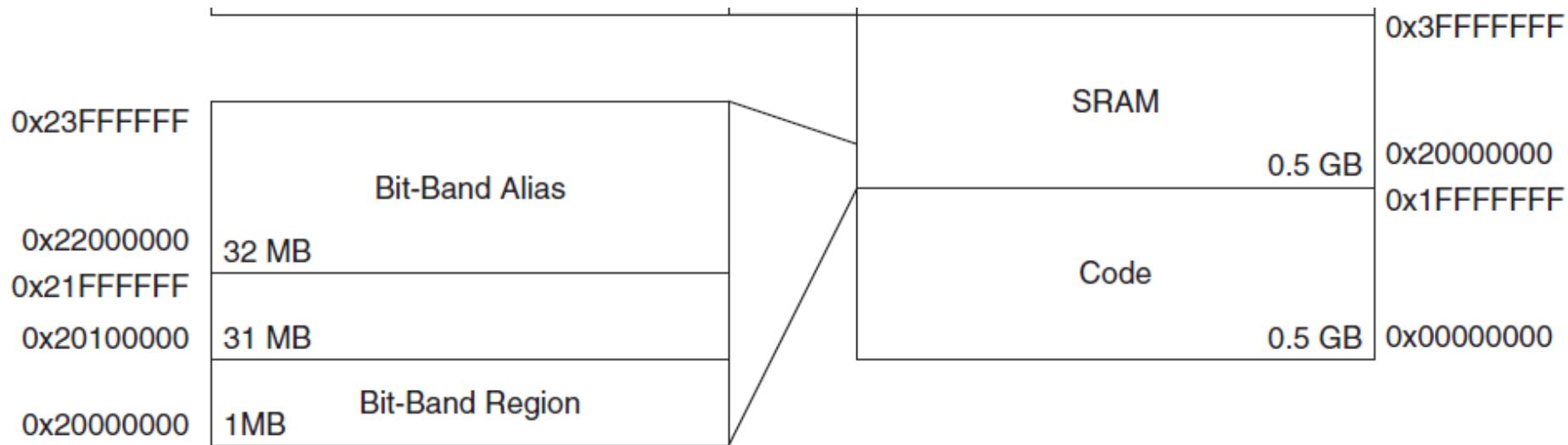
2.6 ARM Cortex-M3 – Memory Access

- Bit-Band region



2.6 ARM Cortex-M3 – Memory Access

- Bit-Band region



2.6 ARM Cortex-M3 – Memory Access

- Default memory access permissions

Memory Region	Address	Access in User Program
Vendor specific	0xE0100000-0xFFFFFFFF	Full access
ROM Table	0xE00FF000-0xE00FFFFF	Blocked; user access results in bus fault
External PPB	0xE0042000-0xE00FEFFF	Blocked; user access results in bus fault
ETM	0xE0041000-0xE0041FFF	Blocked; user access results in bus fault
TPIU	0xE0040000-0xE0040FFF	Blocked; user access results in bus fault
Internal PPB	0xE000F000-0xE003FFFF	Blocked; user access results in bus fault
NVIC	0xE000E000-0xE000EFFF	Blocked; user access results in bus fault, except Software Trigger Interrupt Register that can be programmed to allow user accesses
FPB	0xE0002000-0xE0003FFF	Blocked; user access results in bus fault
DWT	0xE0001000-0xE0001FFF	Blocked; user access results in bus fault
ITM	0xE0000000-0xE0000FFF	Read allowed; write ignored except for stimulus ports with user access enabled
External Device	0xA0000000-0xDFFFFFFF	Full access
External RAM	0x60000000-0x9FFFFFFF	Full access
Peripheral	0x40000000-0x5FFFFFFF	Full access
SRAM	0x20000000-0x3FFFFFFF	Full access
Code	0x00000000-0x1FFFFFFF	Full access

2.7 ARM Cortex-M3 – Bus Interface & MPU

- Bus Interface
 - Code memory bus for code memory, consist of two buses: I-Code and D-code
 - System bus: for memory and peripherals
 - Private peripheral bus: for private peripherals such as debugging components
- Memory Protection Unit (MPU)
 - allow access rules for privilege access and user program access
 - When an access rule is violated, a **fault exception** is generated
 - MPU is setup by an OS
 - allowing data used by privileged code to be protected from untrusted user programs

2.8 ARM Cortex-M3 – Instruction set

- Instruction Set:
 - support thumb-2 instruction set.
 - allow 32-bit and 16-bit instructions
- Traditional ARM processor has two operation states:
 - 32-bit ARM state
 - 16-bit Thumb state
- To get the best of both worlds, many applications have **mixed ARM and Thumb codes**
- Advantages over traditional ARM processor of ARM Cortex M3
 - No state switching overhead, saving both execution time and instruction space
 - No need to separate ARM code and Thumb code source files
 - easier to write software, because there is no need to worry about switching code between ARM and Thumb

2.9 ARM Cortex-M3 - Exceptions

- The interrupt features in the Cortex-M3 are implemented in the NVIC

Exception Number	Exception Type	Priority (Default to 0 if Programmable)	Description
0	NA	NA	No exception running
1	Reset	-3 (Highest)	Reset
2	NMI	-2	Nonmaskable interrupt (external NMI input)
3	Hard fault	-1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus fault	Programmable	Bus error (Prefetch Abort or Data Abort)
6	Usage fault	Programmable	Exceptions due to program error
7-10	Reserved	NA	Reserved

2.2 ARM Cortex-M3 - Exceptions

11	SVCall	Programmable	System service call
12	Debug monitor	Programmable	Debug monitor (break points, watchpoints, or external debug request)
13	Reserved	NA	Reserved
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System tick timer
16	IRQ #0	Programmable	External interrupt #0
17	IRQ #1	Programmable	External interrupt #1
...
255	IRQ #239	Programmable	External interrupt #239

- The number of external interrupt inputs is defined by chip manufacturers.
- A maximum of 240 external interrupt inputs can be supported

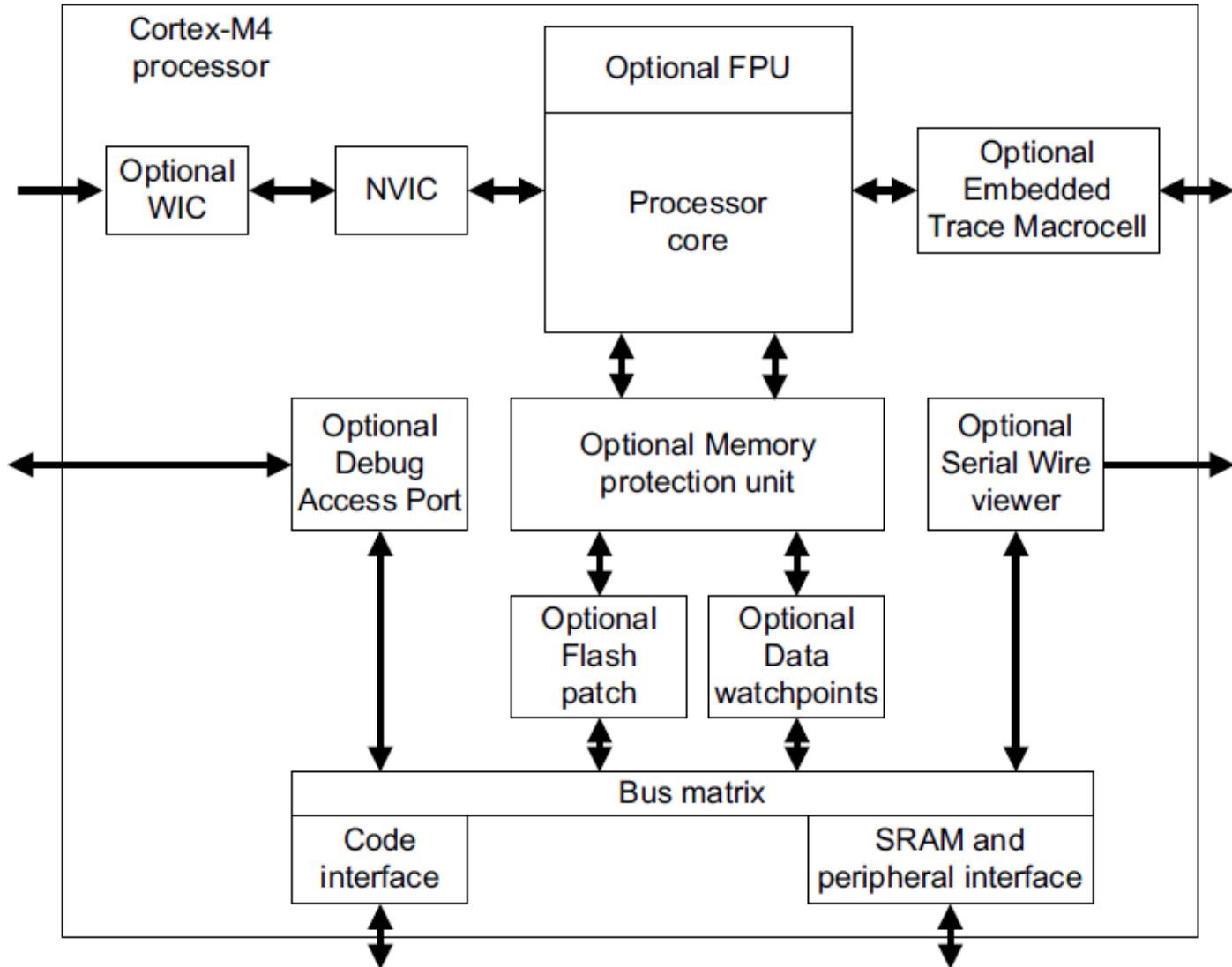
Questions

1. What are differences among Cortex- A, -R, and -M series?
2. What is Harvard architecture?
3. How many general purpose registers in Cortex-M3 are there?
4. How many special registers in Cortex-M3 are there?
5. What are difference between little endian and big endian?
6. How many allowed operation modes in Cortex-M3 are there?
7. What is functions of control register in Cortex-M3?
8. What are features of NVIC in Cortex-M3?
9. What is the maximum memory space of Cortex-M3?
10. What is the size of memory space for external RAM?
11. Can program code can be executed from an external RAM region?
12. Does Cortex-M3 support ARM code?
13. Why is Thumb-2 instruction set more advanced than previous?
14. In which areas of memory are instructions not permitted to be executed?
15. Can a non-maskable interrupt be preempted by an exception?

3. ARM Cortex-M4

- Features
 - tight integration of system peripherals reduces area and development costs
 - Thumb instruction set combines high code density with 32-bit performance
 - **optional IEEE754-compliant single-precision FPU**
 - code-patch ability for ROM system updates
 - power control optimization of system components
 - integrated sleep modes for low power consumption
 - **DSP extension: Single cycle 16/32-bit MAC, single cycle dual 16-bit MAC, 8/16-bit SIMD arithmetic**

3. ARM Cortex-M4 - Architecture



3. ARM Cortex-M4 – Peripherals

- **Nested Vectored Interrupt Controller**
 - An interrupt controller that supports low latency interrupt processing.
- **System Control Block**
 - Provides system implementation information and system control, including configuration, control, and reporting of system exceptions.
- **System timer**
 - The system timer, SysTick, is a 24-bit count-down timer. Use this as a Real Time Operating System (RTOS) tick timer or as a simple counter.
- **Memory Protection Unit**
 - *defining the* memory attributes for different memory regions. It provides up to eight different regions, and an optional predefined background region.
- **Floating-point Unit**
 - The *Floating-Point Unit (FPU)* provides IEEE754-compliant operations on single-precision, 32-bit, floating-point values.

3. ARM Cortex-M4 – Instruction Set

- Extra instructions for floating point operations

VABS.F32	Sd, Sm	Floating-point Absolute
VADD.F32	{Sd,} Sn, Sm	Floating-point Add
VCMP.F32	Sd, <Sm #0.0>	Compare two floating-point registers, or one floating-point register and zero
VCMPE.F32	Sd, <Sm #0.0>	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check
VCVT.S32.F32	Sd, Sm	Convert between floating-point and integer
VCVT.S16.F32	Sd, Sd, #fbits	Convert between floating-point and fixed point
VCVTR.S32.F32	Sd, Sm	Convert between floating-point and integer with rounding

3. ARM Cortex-M4 – Instruction Set

VCVT<B H>.F32.F16	Sd, Sm	Converts half-precision value to single-precision
VCVTT<B T>.F32.F16	Sd, Sm	Converts single-precision register to half-precision
VDIV.F32	{Sd,} Sn, Sm	Floating-point Divide
VFMA.F32	{Sd,} Sn, Sm	Floating-point Fused Multiply Accumulate
VFNMA.F32	{Sd,} Sn, Sm	Floating-point Fused Negate Multiply Accumulate
VFMS.F32	{Sd,} Sn, Sm	Floating-point Fused Multiply Subtract
VFNMS.F32	{Sd,} Sn, Sm	Floating-point Fused Negate Multiply Subtract
VLDM.F<32 64>	Rn{!}, list	Load Multiple extension registers
VLDR.F<32 64>	<Dd Sd>, [Rn]	Load an extension register from memory
VLMA.F32	{Sd,} Sn, Sm	Floating-point Multiply Accumulate
VLMS.F32	{Sd,} Sn, Sm	Floating-point Multiply Subtract
VMOV.F32	Sd, #imm	Floating-point Move immediate

3. ARM Cortex-M4 – Instruction Set

VMOV	Sd, Sm	Floating-point Move register
VMOV	Sn, Rt	Copy ARM core register to single precision
VMOV	Sm, Sm1, Rt, Rt2	Copy 2 ARM core registers to 2 single precision
VMOV	Dd[x], Rt	Copy ARM core register to scalar
VMOV	Rt, Dn[x]	Copy scalar to ARM core register
VMRS	Rt, FPSCR	Move FPSCR to ARM core register or APSR
VMSR	FPSCR, Rt	Move to FPSCR from ARM Core register
VMUL.F32	{Sd,} Sn, Sm	Floating-point Multiply
VNEG.F32	Sd, Sm	Floating-point Negate
VNMLA.F32	Sd, Sn, Sm	Floating-point Multiply and Add
VNMLS.F32	Sd, Sn, Sm	Floating-point Multiply and Subtract
VNMUL	{Sd,} Sn, Sm	Floating-point Multiply

3. ARM Cortex-M4 – Instruction Set

VPOP	list	Pop extension registers
VPUSH	list	Push extension registers
VSQRT.F32	Sd, Sm	Calculates floating-point Square Root
VSTM	Rn{!}, list	Floating-point register Store Multiple
VSTR.F<32 64>	Sd, [Rn]	Stores an extension register to memory
VSUB.F<32 64>	{Sd,} Sn, Sm	Floating-point Subtract

3. ARM Cortex-M4 – Instruction Set

- Extra instructions for DSP

UADD16	{Rd,} Rn, Rm	Unsigned Add 16
UADD8	{Rd,} Rn, Rm	Unsigned Add 8
USAX	{Rd,} Rn, Rm	Unsigned Subtract and Add with Exchange
UHADD16	{Rd,} Rn, Rm	Unsigned Halving Add 16
UHADD8	{Rd,} Rn, Rm	Unsigned Halving Add 8
UHASX	{Rd,} Rn, Rm	Unsigned Halving Add and Subtract with Exchange
UHSAX	{Rd,} Rn, Rm	Unsigned Halving Subtract and Add with Exchange
UHSUB16	{Rd,} Rn, Rm	Unsigned Halving Subtract 16
UHSUB8	{Rd,} Rn, Rm	Unsigned Halving Subtract 8
UBFX	Rd, Rn, #1sb, #width	Unsigned Bit Field Extract
UDIV	{Rd,} Rn, Rm	Unsigned Divide

3. ARM Cortex-M4 – Instruction Set

- Extra instructions for DSP

UMAAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply Accumulate Accumulate Long (32 x 32 + 32 +32), 64-bit result
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply with Accumulate (32 x 32 + 64), 64-bit result
UMULL	RdLo, RdHi, Rn, Rm	Unsigned Multiply (32 x 32), 64-bit result
UQADD16	{Rd,} Rn, Rm	Unsigned Saturating Add 16
UQADD8	{Rd,} Rn, Rm	Unsigned Saturating Add 8

3. ARM Cortex-M4 – Instruction Set

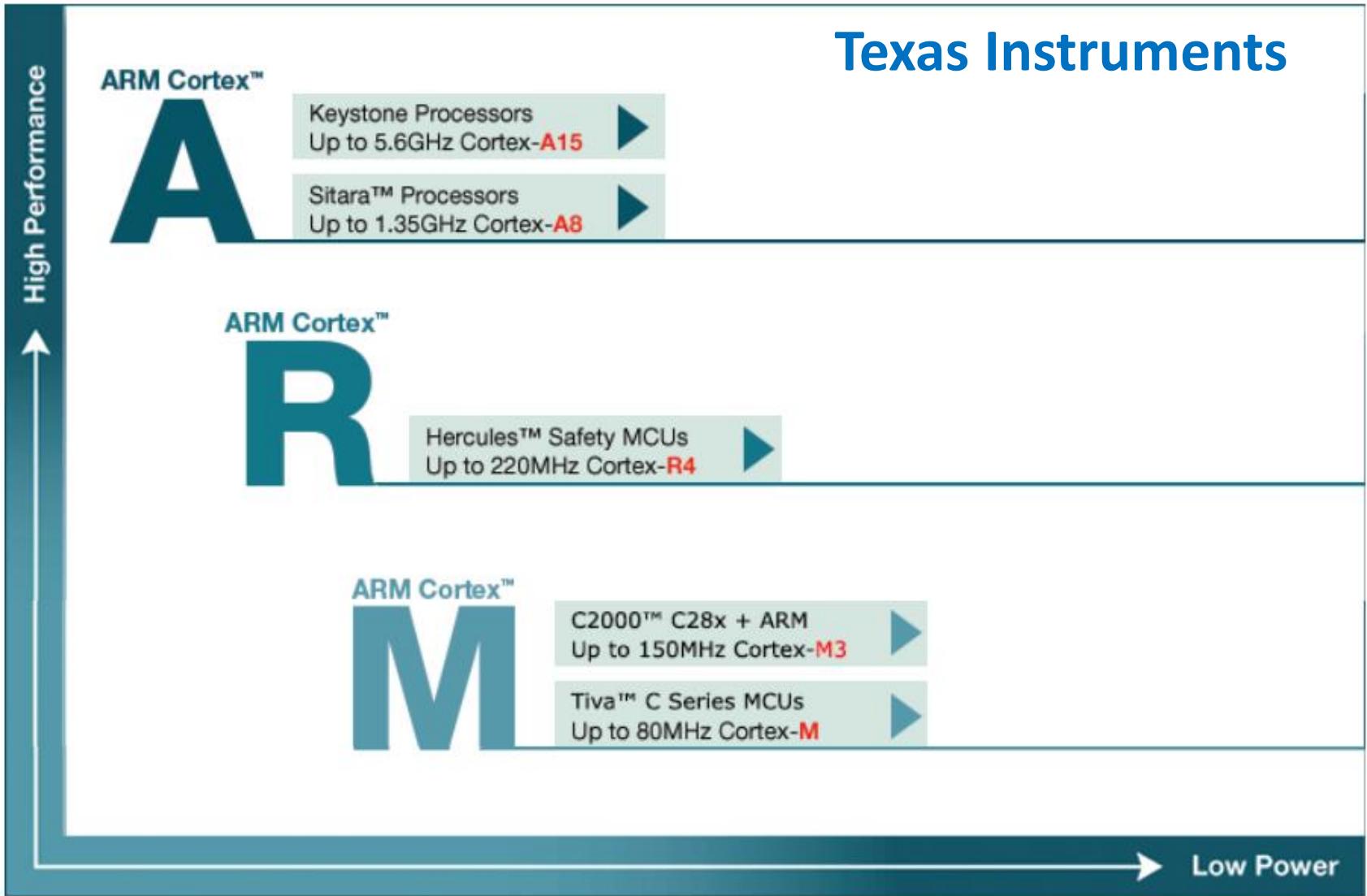
- Extra instructions for DSP

UQASX	{Rd,} Rn, Rm	Unsigned Saturating Add and Subtract with Exchange
UQSAX	{Rd,} Rn, Rm	Unsigned Saturating Subtract and Add with Exchange
UQSUB16	{Rd,} Rn, Rm	Unsigned Saturating Subtract 16
UQSUB8	{Rd,} Rn, Rm	Unsigned Saturating Subtract 8
USAD8	{Rd,} Rn, Rm	Unsigned Sum of Absolute Differences
USADA8	{Rd,} Rn, Rm, Ra	Unsigned Sum of Absolute Differences and Accumulate
USAT	Rd, #n, Rm {,shift #s}	Unsigned Saturate
USAT16	Rd, #n, Rm	Unsigned Saturate 16

4. ARM Cortex Microcontroller Series



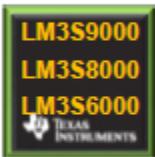
4. ARM Cortex Microcontroller Series



4. ARM Cortex Microcontroller Series

- Stellaris® Roadmap

ARM Cortex-M3



Fixed Point
ENET MAC & PHY
USB & CAN options

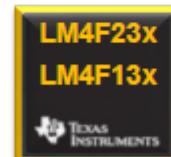


Fixed Point
USB H/D/OTG
CAN options



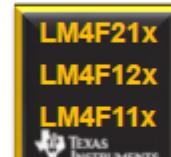
Fixed Point
General Purpose
CAN options

ARM Cortex-M4F Floating-Point



RTP Feb '13 (TMX Now)

- USB H/D/OTG + CAN
- 80 MHz
- 256K Flash / 32K SRAM
- Low-power hibernate
- 2 x 1 Msps 12-bit ADCs
- Motion control options



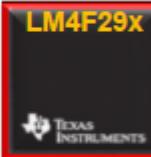
RTP Feb '13 (TMX Now)

- 80 MHz
- 256K Flash / 32K SRAM
- Low-power hibernate
- 2 x 1 Msps 12-bit ADCs
- Up to 2 x CAN
- Motion control options

Production

Sampling

Development



TMS / RTP 2H13
Ethernet + USB + CAN

- 120 MHz
- 1MB Flash, 256KB SRAM
- 10/100 ENET MAC + PHY
- USB H/D/OTG w/FS PHY & HS ULPI
- Up to 2 x CAN
- Parallel Bus Interface (EPI)
- Crypto



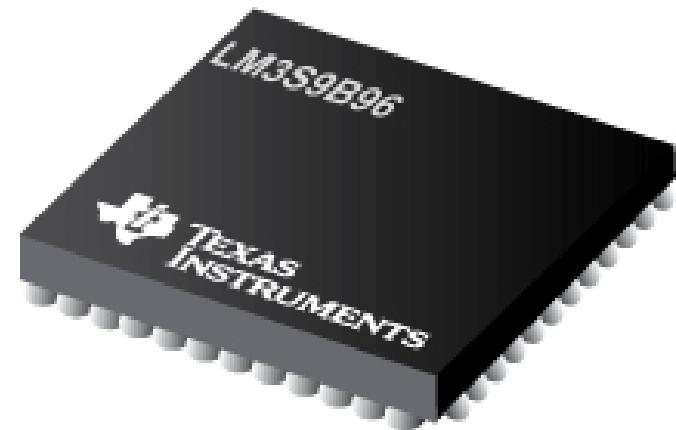
TMS / RTP 2H13

USB + CAN

- 120 MHz
- 1MB Flash, 256KB SRAM
- USB H/D/OTG w/FS PHY & HS ULPI
- Up to 2 x CAN
- Parallel Bus Interface (EPI)
- Crypto

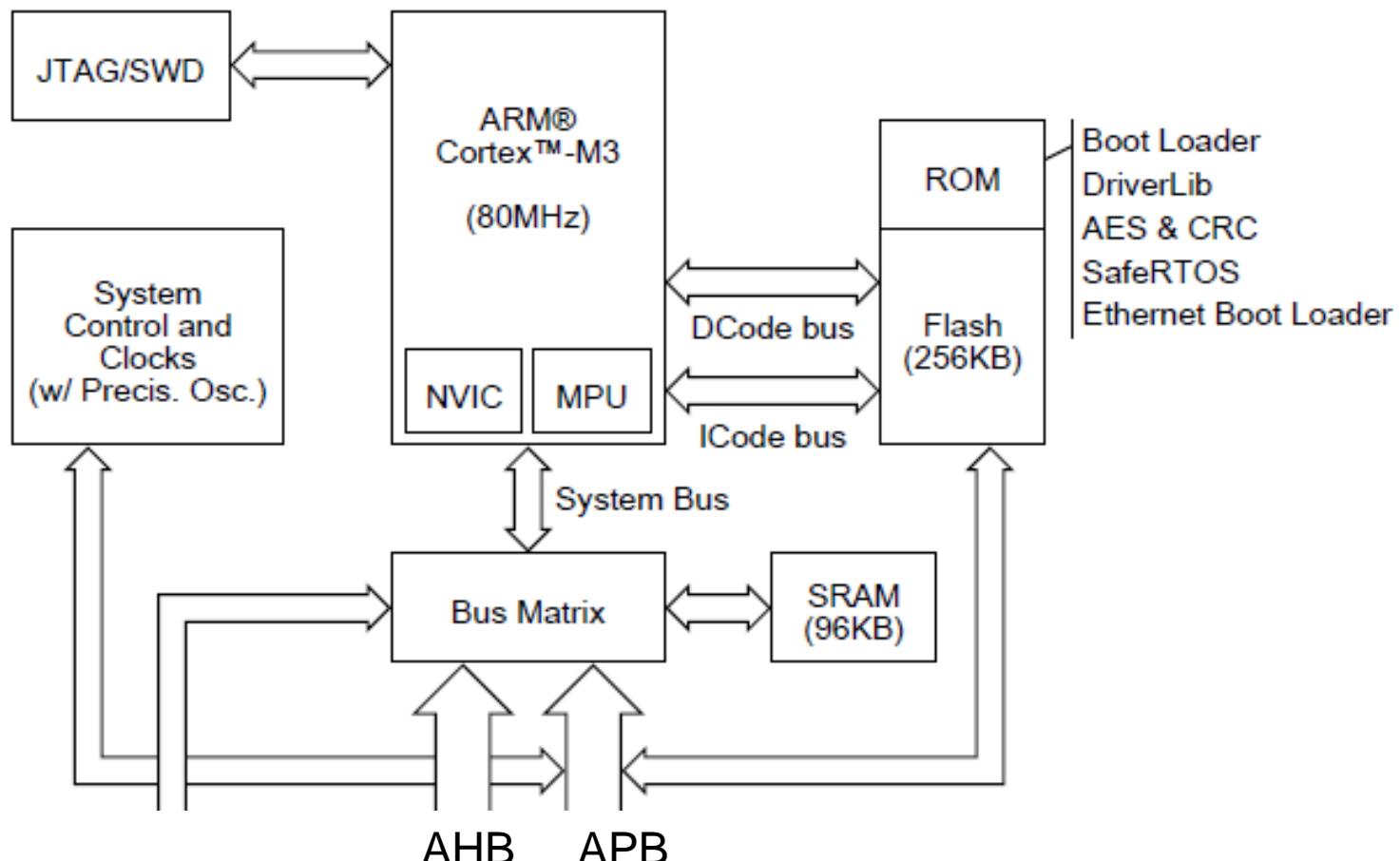
4.1 ARM Cortex-M3 – LM3S9B96

- Stellaris LM3S9B96 microcontroller (Texas Instruments)
 - ARM Cortex-M3
 - 80MHz, 100 DMIPS
 - 256 KB Flash
 - 96KB SRAM
 - UART, SSI, I2C, I2S, CAN, Ethernet, USB
 - Timer, DMA, GPIO
 - PWM
 - ADC
 - JTAG, ARM Serial Wire Debug



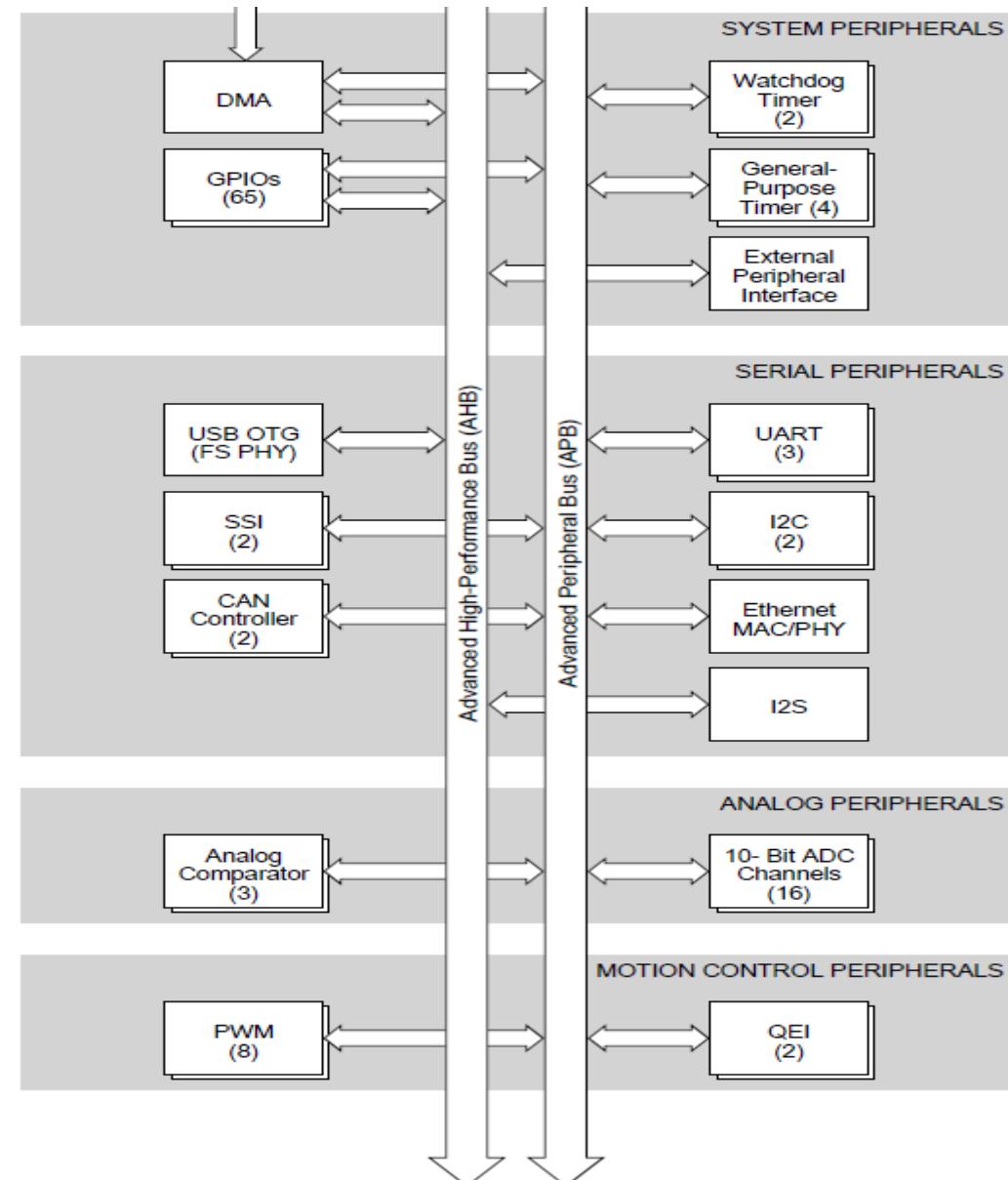
4.1 ARM Cortex-M3 – LM3S9B96

- Block diagram:
 - 2 on-chip buses: AHB, APB



4.1 ARM Cortex-M3 – LM3S9B96

- Target applications
 - Gaming equipment
 - Network appliances
 - Home and commercial site monitoring
 - Motion control
 - Medical instruments
 - Remote monitoring
 - Test equipment
 - Fire and security
 - Lighting control
 - Transportation



4.2 ARM Cortex-M4 – TM4F

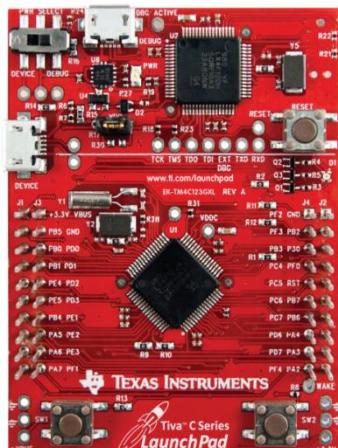
- Features:

- ARM Cortex-M4F core CPU speed up to 80 MHz with floating point
- Up to 256-KB Flash
- Up to 32-KB single-cycle SRAM
- Two high-speed 12-bit ADCs up to 1MSPS
- Up to two CAN 2.0 A/B controllers
- Optional full-speed USB 2.0 OTG/ Host/Device
- Up to 40 PWM outputs
- Serial communication with up to: 8 UARTs, 6 I2Cs, 4 SPI/SSI
- Intelligent low-power design power consumption as low as $1.6 \mu\text{A}$



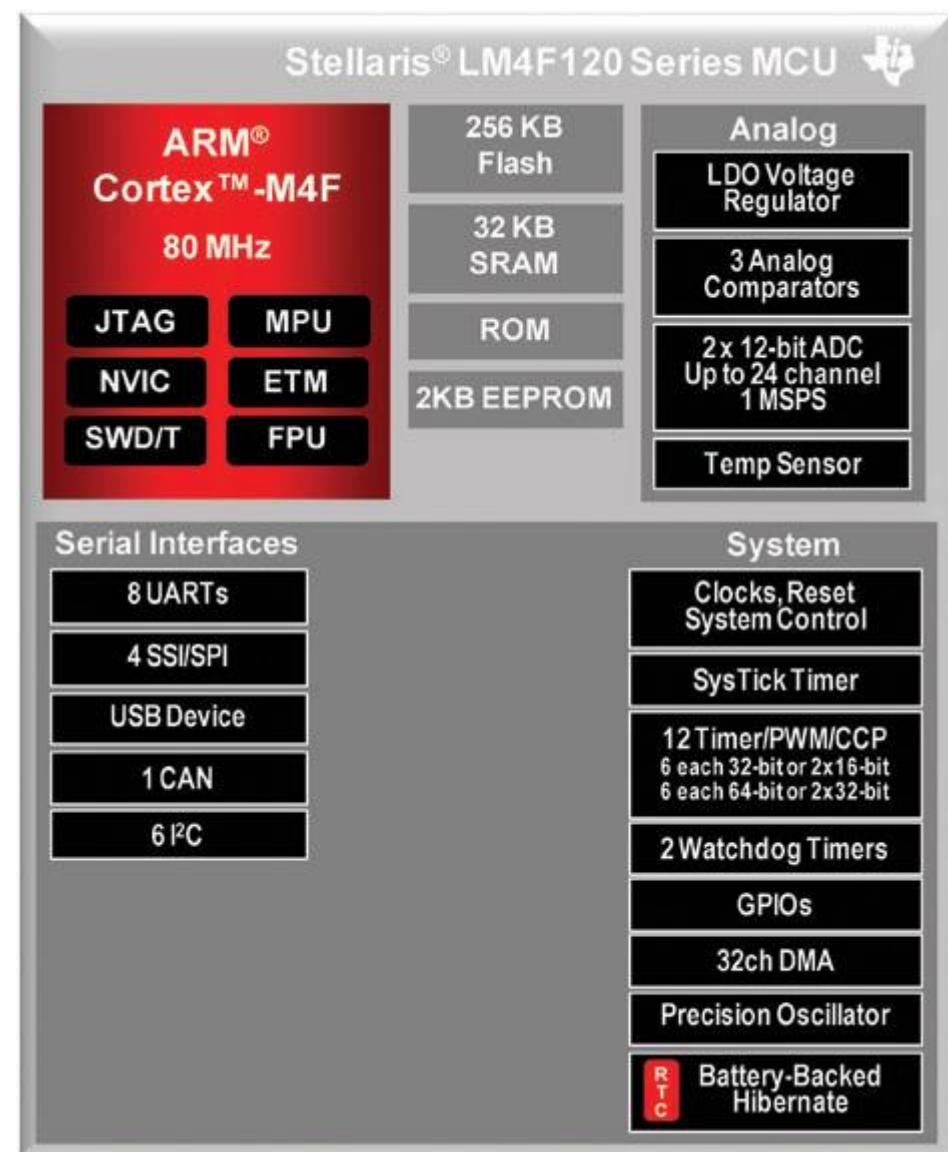
4.2 ARM Cortex-M4 – Development Kit

- EK-TM4F120 LaunchPad Evaluation Kit
 - 80-MHz, 32-bit ARM Cortex-M4 CPU
 - 256 Kbytes of FLASH
 - Many peripherals such as MC PWMs, 1-MSPS ADCs, eight UARTs, four SPIs, four I2Cs, USB Host|Device, and up to 27 timers.
 - EK-LM4F232 Development Kit
 - ARM® Cortex™-LX4F232
 - color OLED display,
 - USB OTG,
 - A micro SD card, a coin cell battery,
 - A temperature sensor,
 - A three axis



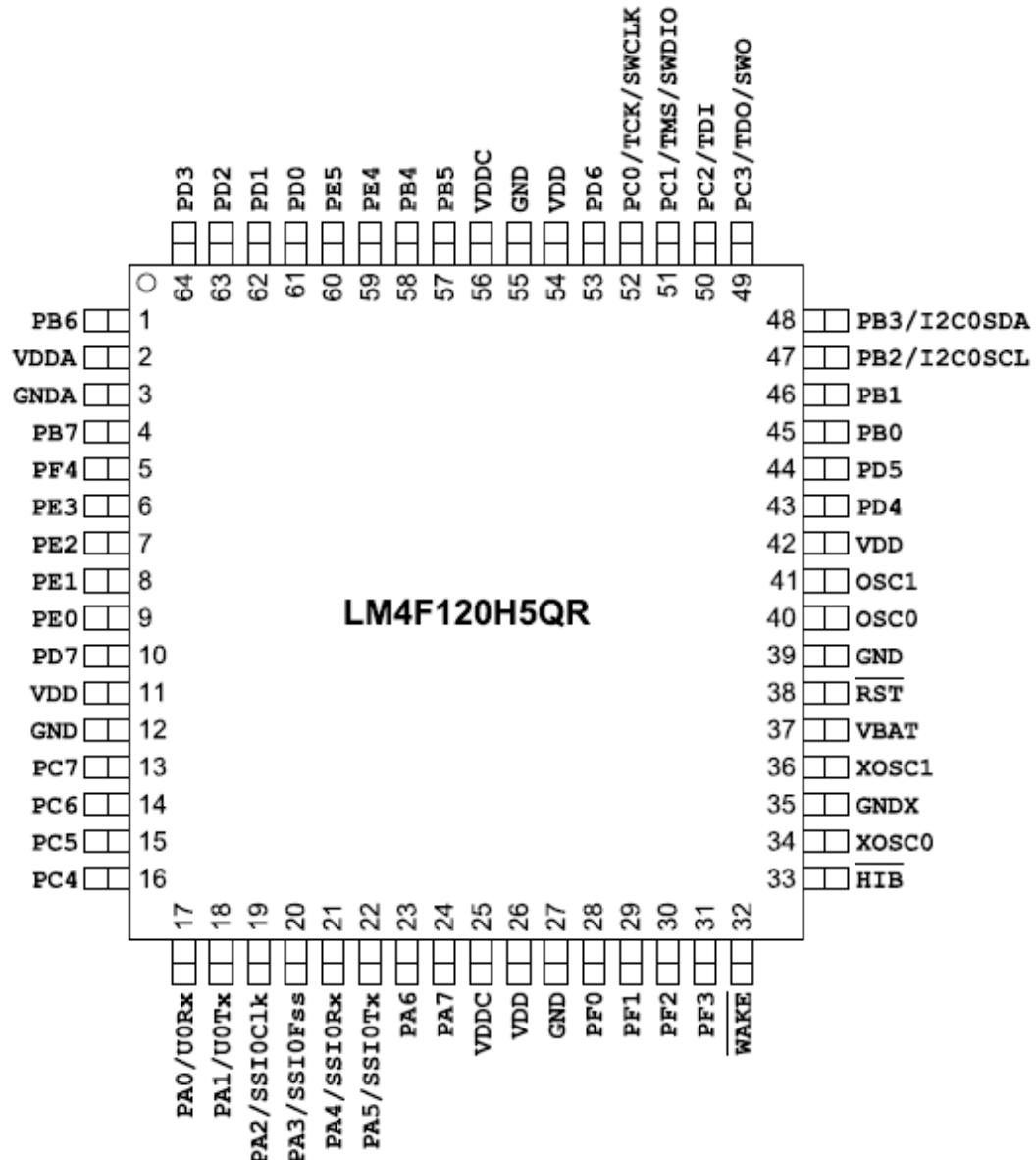
4.2 ARM Cortex-M3 – TM4F

- **Connectivity features:**
 - CAN, USB Device, SPI/SSI, I2C, UARTs
- **High-performance analog integration**
 - Two 1 MSPS 12-bit ADCs
 - Analog and digital comparators
- **Best-in-class power consumption**
 - As low as $370 \mu\text{A}/\text{MHz}$
 - $500\mu\text{s}$ wakeup from low-power modes
 - RTC currents as low as $1.7\mu\text{A}$
- **Solid roadmap**
 - Higher speeds, Ultra-low power
 - Larger memory



4.3. ARM Cortex-M4 - LM4F120H5QR

- LM4F120H5QR
package



4.3. ARM Cortex-M4 - LM4F120H5QR

- LM4F120H5QR has **6 GPIO blocks**, supporting up to **43 IO pins**

- Port A: 8 bits
 - Port B : 8 bits
 - Port C : 8 bits
 - Port D : 8 bits
 - Port E : 6 bits
 - Port F : 5 bits

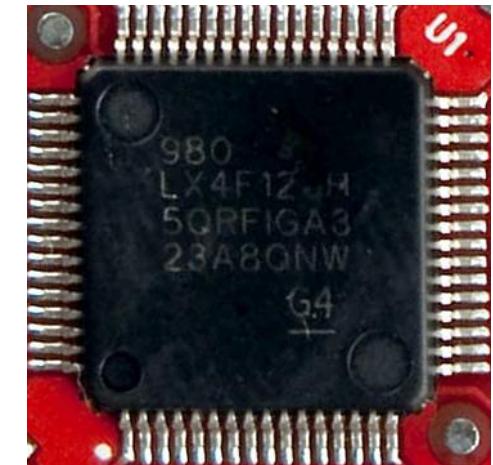
- GPIO pad configuration
 - Weak pull-up or pull-down resistors
 - 2-mA, 4-mA, and 8-mA pad drive
 - Slew rate control for 8-mA pad drive
 - Open drain enables
 - Digital input enables

17	PA0	PB0	45
18	PA1	PB1	46
19	PA2	PB2	47
20	PA3	PB3	48
21	PA4	PB4	58
22	PA5	PB5	57
23	PA6	PB6	1
24	PA7	PB7	4
52	PC0	PD0	61
51	PC1	PD1	62
50	PC2	PD2	63
49	PC3	PD3	64
16	PC4	PD4	43
15	PC5	PD5	44
14	PC6	PD6	53
13	PC7	PD7	10
9	PE0	PF0	28
8	PE1	PF1	29
7	PE2	PF2	30
6	PE3	PF3	31
59	PE4	PF4	5
60	PE5		

LM4F120

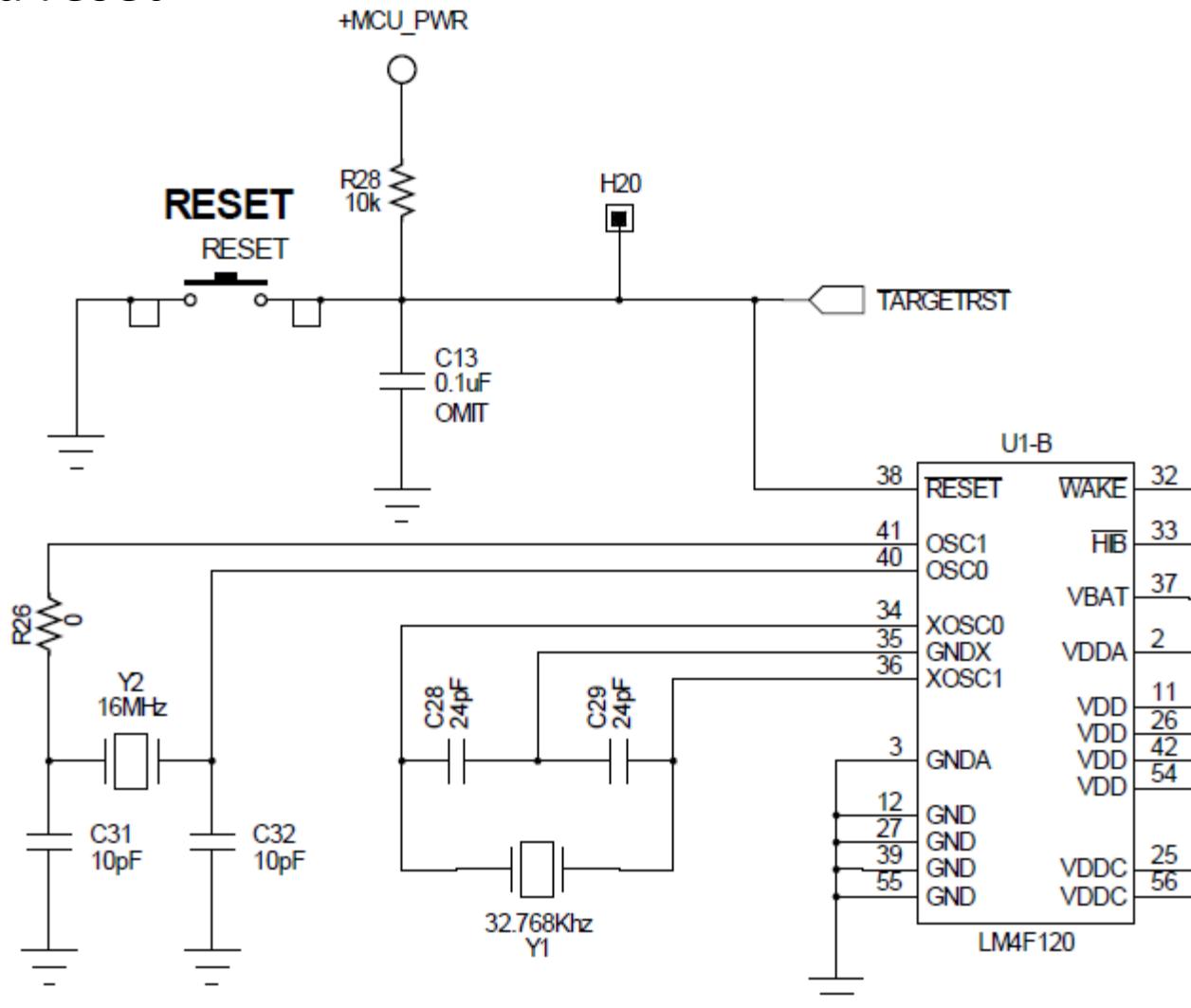
4.3. ARM Cortex-M4 - LM4F120H5QR

- 256KB Flash memory
 - Single-cycle to **40MHz**
 - Pre-fetch buffer and speculative branch improves performance above 40 MHz
- 32KB single-cycle SRAM with bit-banding
 - Internal ROM loaded with StellarisWare software
 - Stellaris Peripheral Driver Library
 - Stellaris Boot Loader
 - Advanced Encryption Standard (AES) cryptography tables
 - Cyclic Redundancy Check (CRC) error detection functionality
- 2KB EEPROM (fast, saves board space)
 - Wear-leveled 500K program/erase cycles
 - 10 year data retention
 - 4 clock cycle read time

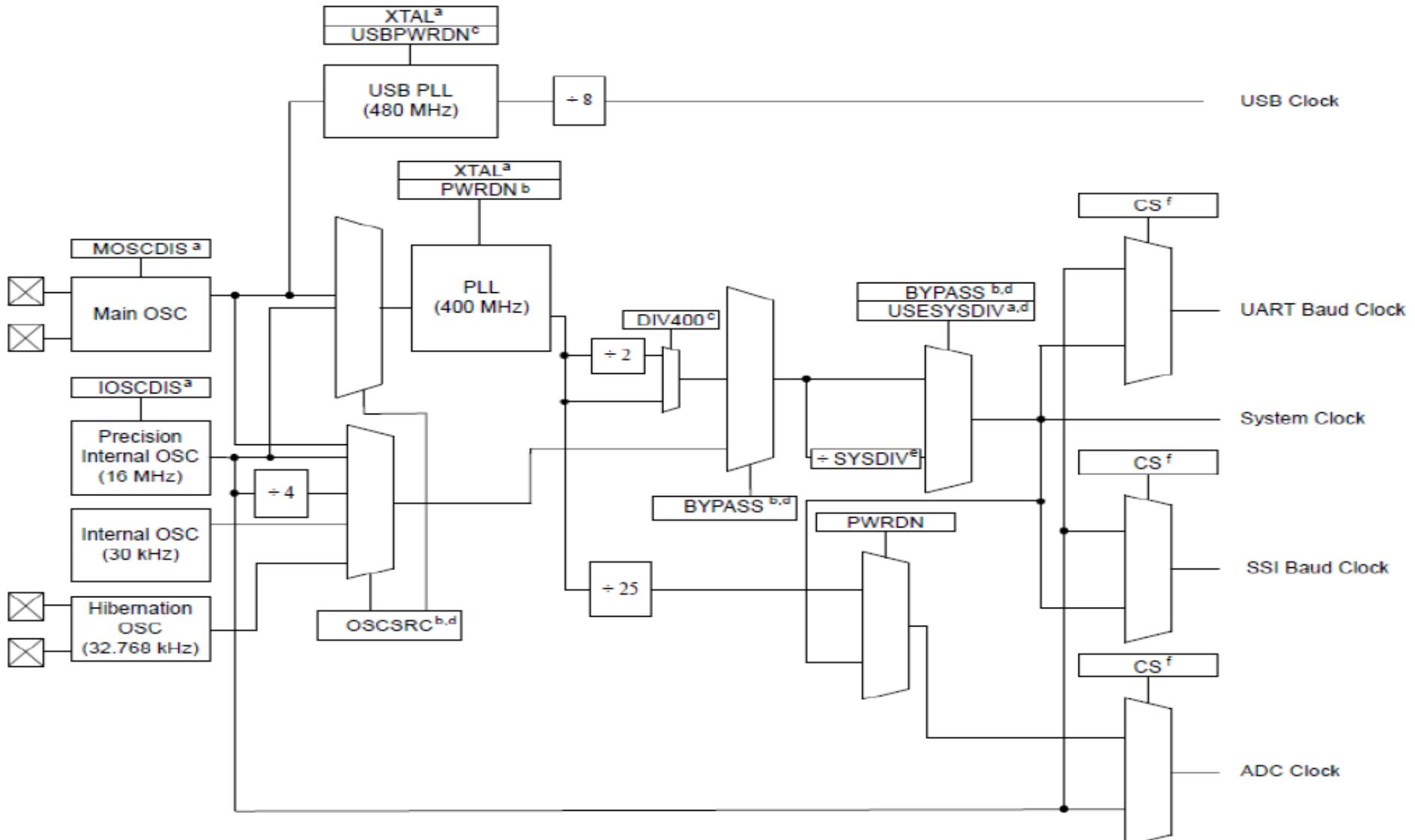


4.3. ARM Cortex-M4 - LM4F120H5QR

- Clock and reset



4.3. ARM Cortex-M4 - LM4F120H5QR



4.3. ARM Cortex-M4 - LM4F120H5QR

USB Device Signals

GPIO Pin	Pin Function	USB Device
PD4	USB0DM	D-
PD5	USB0DP	D+

Stellaris® In-Circuit Debug Interface (ICDI) Signals

GPIO	Pin Function
PC0	TCK/SWCLK
PC1	PC1 TMS/SWDIO
PC2	TDI
PC3	PC3 TDO/SWO

Virtual COM Port Signals

GPIO Pin	Pin Function
PA0	U0RX
PA1	U0TX

4.3. ARM Cortex-M4 - LM4F120H5QR

Virtual COM Port Signals

GPIO Pin	Pin Function
PA0	U0RX
PA1	U0TX

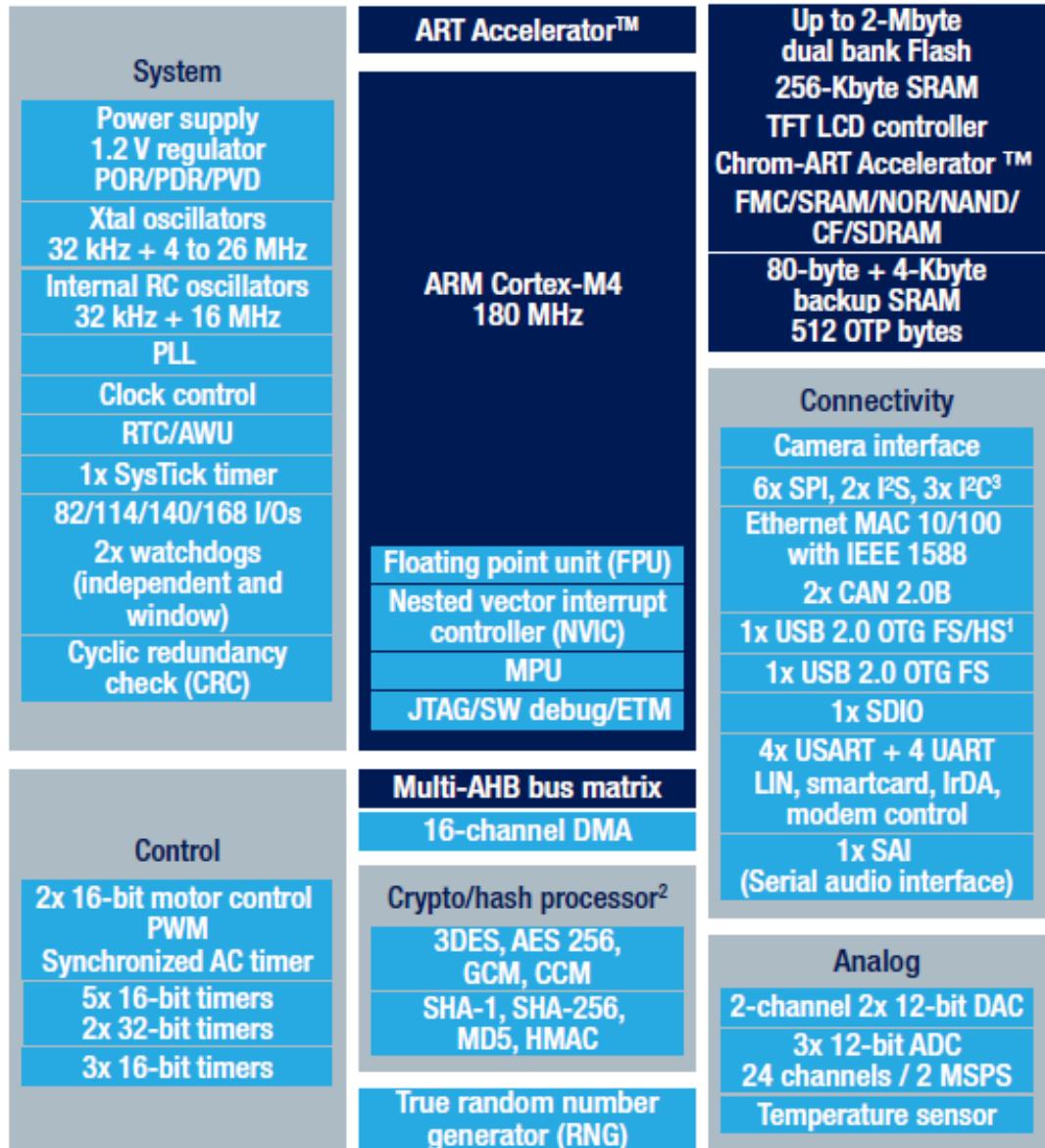
4.3. ARM Cortex-M4 – STM32F4

- Features:
 - 180 MHz/225 DMIPS Cortex-M4
 - Single cycle DSP MAC and floating point unit
 - Memory accelerator
 - Graphic accelerator
 - Multi DMA controllers
 - SDRAM interface support
 - Ultra-low dynamic power in Run mode: $260 \mu\text{A}/\text{MHz}$ at 180 MHz



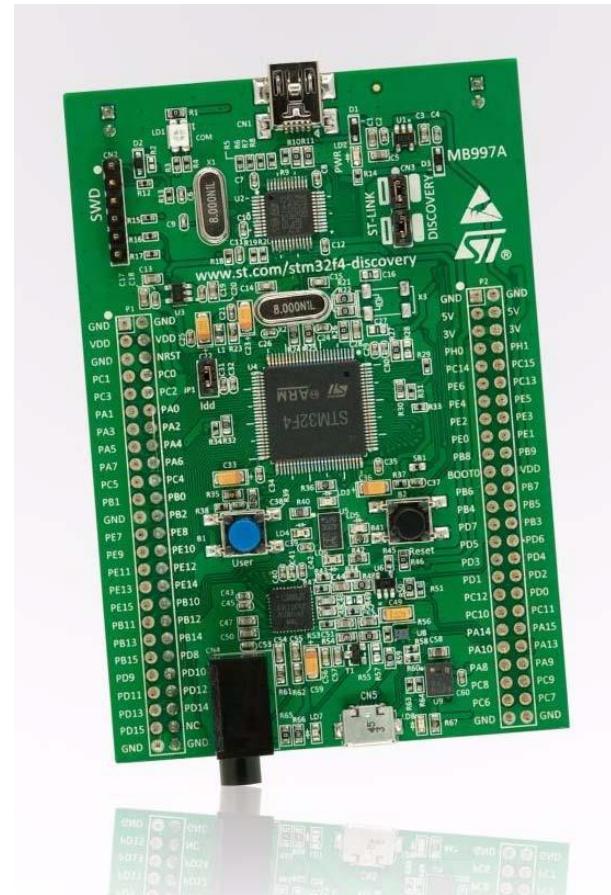
4.3. ARM Cortex-M4 – STM32F4

- Block diagram



4.3. ARM Cortex-M4 – STM32F4

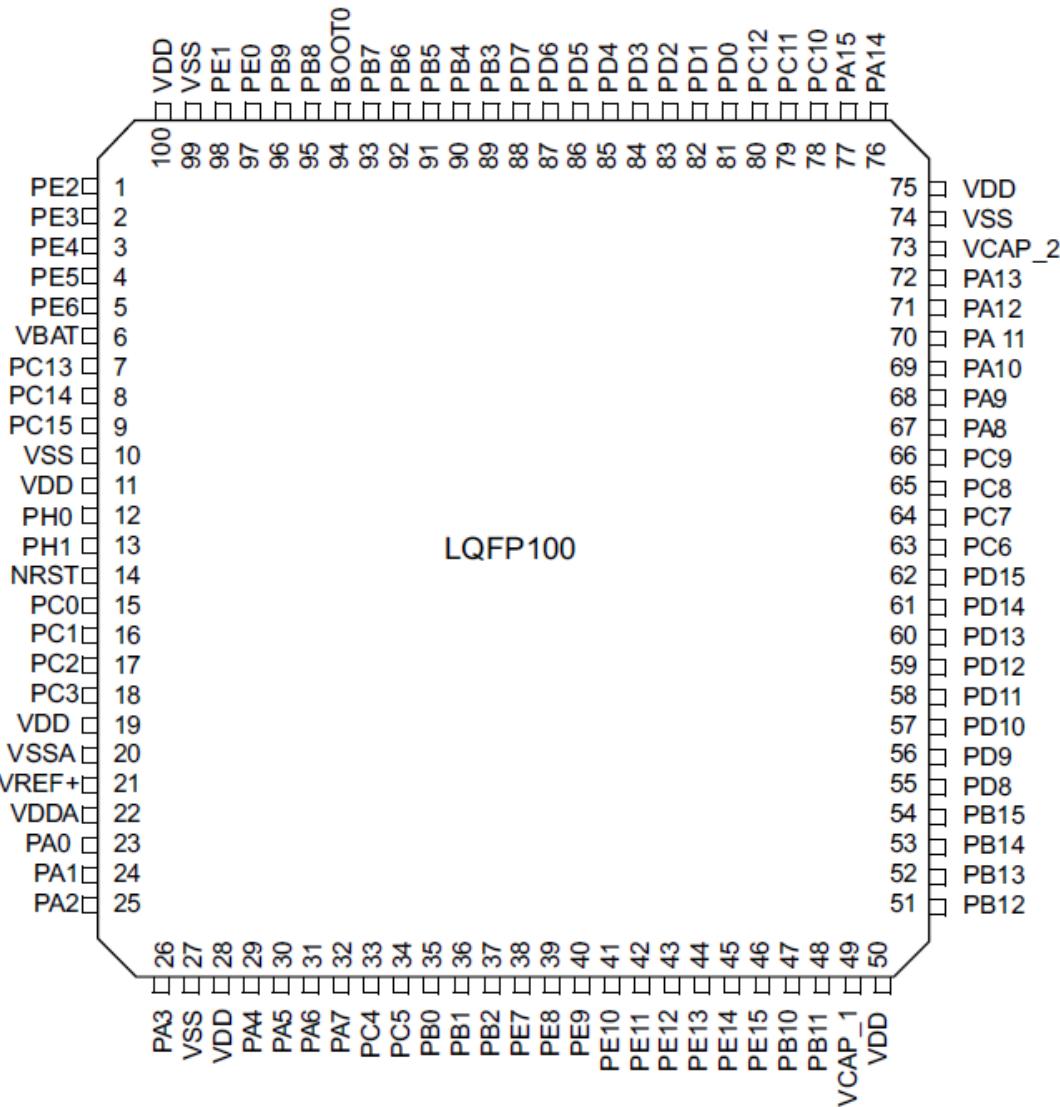
- Development kit
 - STM32F407VGT6 microcontroller
 - 168MHz/210 DMIPS
 - DSP MAC and floating point unit
 - 1 MB Flash, 192 KB RAM
 - On-board ST-LINK/V2
 - Power supply: 3 V and 5 V
 - 3-axis accelerometer
 - Audio sensor, omni-directional digital microphone
 - audio DAC with integrated class D speaker driver
 - Eight LEDs:
 - Two push buttons (user and reset)



Discovery kit for STM32F407

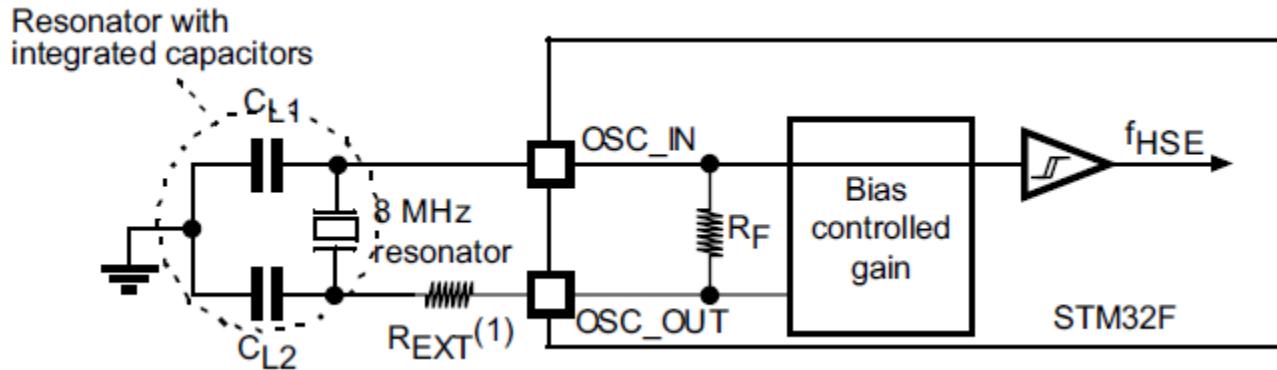
4.3. ARM Cortex-M4 – STM32F4

- Package LQFP100
- GPIO
 - Port A: 16 bit
 - Port B: 16 bit
 - Port C: 16 bit
 - Port D: 16 bit
 - Port E: 16 bit
- can sink or source up to $\pm 8\text{mA}$
- except PC13, PC14 and PC15 which can sink or source up to $\pm 3\text{mA}$

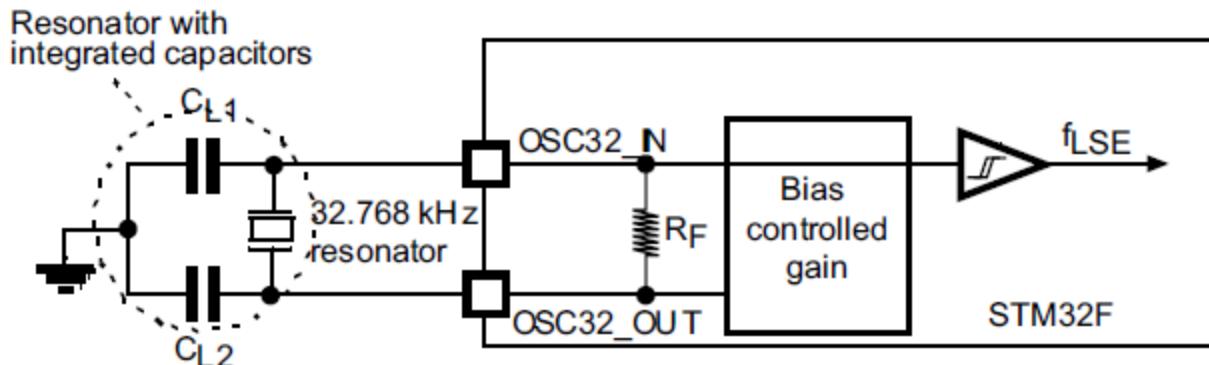


4.3. ARM Cortex-M4 – STM32F4

Typical application with an 8 MHz crystal

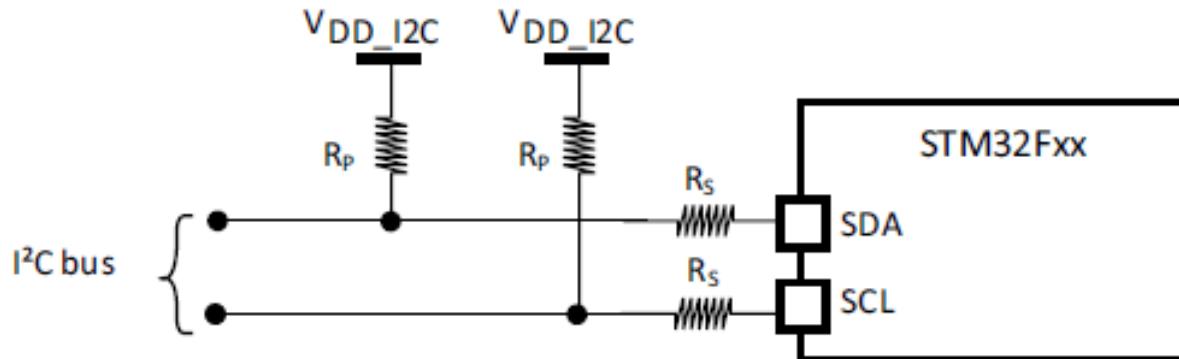
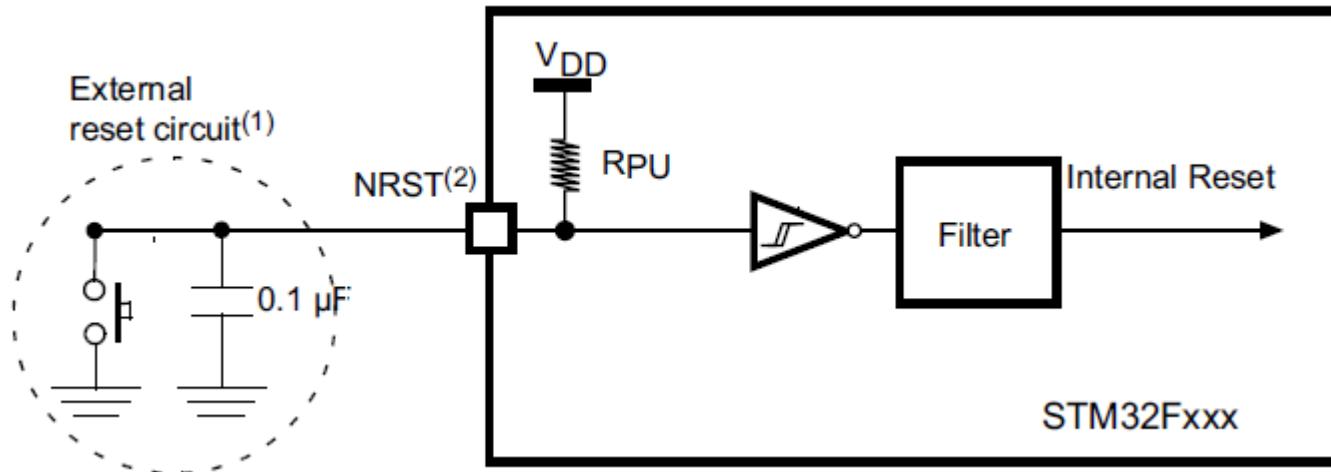


Typical application with a 32.768 kHz crystal



4.3. ARM Cortex-M4 – STM32F4

- Reset circuit



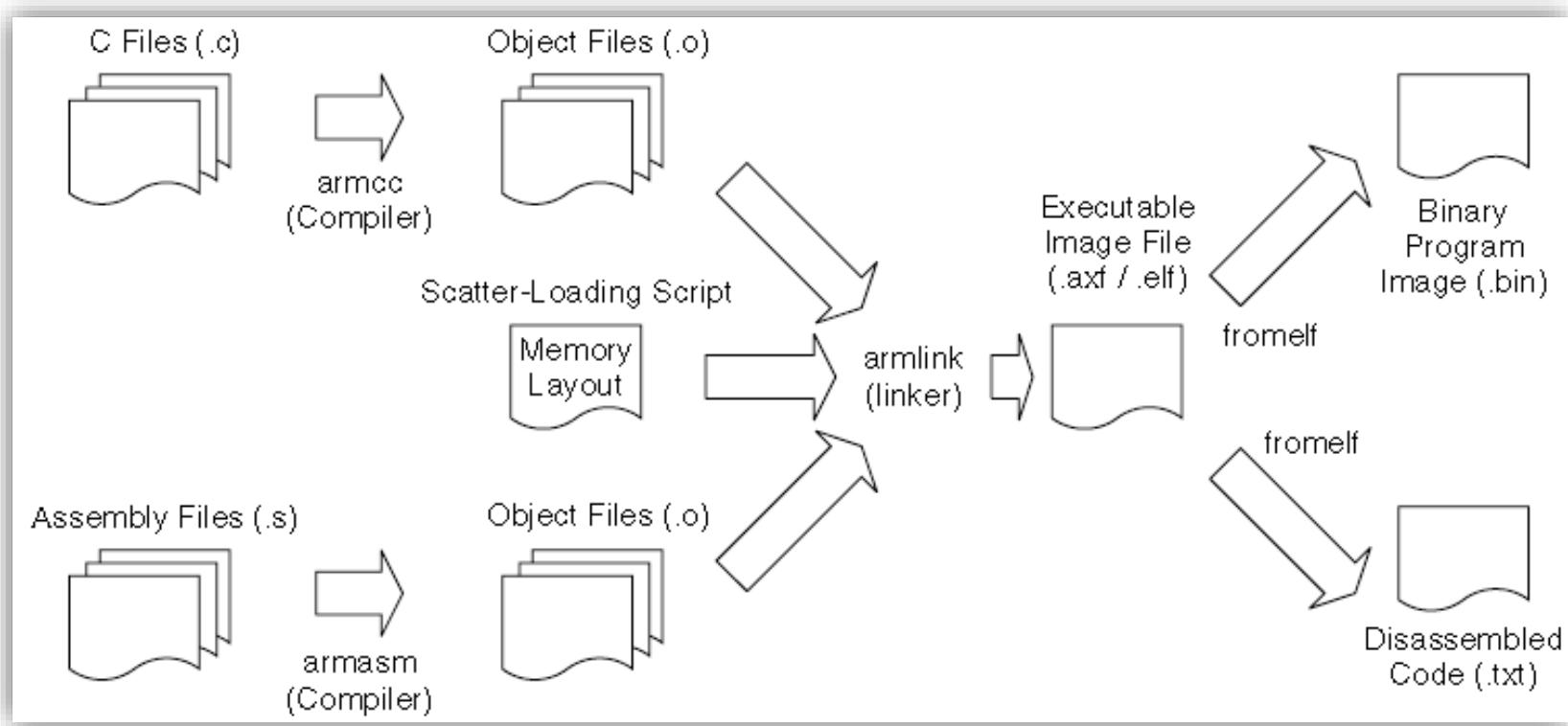


5. ARM Programming

- Using Assembly
 - for small projects
 - can get the best optimization, smallest memory size
 - increase development time, easy to make mistakes
- Using C
 - easier for implementing complex operations
 - larger memory size
 - able to include assembly code (*inline assembler*)
 - Tools: RealView Development Suite (RVDS), KEIL RealView Microcontroller Development Kit, Code Composer, IAR

5. ARM Programming

- Typical development flow



5. ARM Programming – Simple program

This simple program contains the initial SP value, the initial PC value, and setup registers and then does the required calculation in a loop.

```
STACK_TOP      EQU 0x20002000 ; constant for SP starting value
                AREA |Header Code|, CODE
                DCD STACK_TOP    ; Stack top
                DCD Start        ; Reset vector
                ENTRY            ; Indicate program execution start here
                ; Start of main program initialize registers
Start
                MOV r0, #10       ; Starting loop counter value
                MOV r1, #0         ; starting result
loop
                ADD r1, r0       ; R1 = R1 + R0
                SUBS r0, #1       ; Decrement R0, update flag ("S" suffix)
                BNE loop         ; If result not zero jump to loop,
                ; result is now in R1
deadloop
                B deadloop       ; Infinite loop
END             ; End of file
```

5. ARM Programming - Simple program

- Compile a assembly code
 - armasm --cpu cortex-m3 -o test1.o test1.s
- Link to executable image
 - armlink --rw_base 0x20000000 --ro_base 0x0 --map -o test1.elf test1.o
- Create the binary image
 - fromelf --bin --output test1.bin test1.elf
- generate a disassembled code list file
 - fromelf -c --output test1.list test1.elf

5. ARM Programming

- Using EQU to define constants

```
NVIC_IRQ_SETENO      EQU      0xE000E100
NVIC_IRQ0_ENABLE    EQU      0x1
LDR R0,NVIC_IRQ_SETENO
MOV R1,#NVIC_IRQ0_ENABLE ; Move immediate data to register
STR R1, [R0]          ; Enable IRQ 0 by writing R1 to address in R0
```

- Using DCI to code an instruction

```
DCI 0xBE00 ; Breakpoint (BKPT 0), a 16-bit instruction
```

- Using DCB and DCD to define binary data

```
MY_NUMBER
DCD 0x12345678
HELLO_TXT
DCB "Hello\n",0 ; null terminated string
```

5. ARM Programming – Moving data

- Data transfers can be of one of the following types:
 - Moving data between register and register
 - Moving data between memory and register
 - Moving data between special register and register
 - Moving an immediate data value into a register

MOV	R8, R3	; moving data from register R3 to register R8
MOV	R0, #0x12	; Set R0 = 0x12 (hexadecimal)
MOV	R1, #'A'	; Set R1 = ASCII character A

MRS	R0, PSR	; Read Processor status word into R0
MSR	CONTROL, R1	; Write value of R1 into control register

LDR R0, address1	; R0 set to 0x4001
...	
address1	
0x4000: MOV R0, R1	; address1 contains program code

5. ARM Programming – Using Stack

- Stack PUSH and POP

subroutine_1

```
PUSH {R0-R7, R12, R14}      ; Save registers
... ; Do your processing
POP {R0-R7, R12, R14}       ; Restore registers
BX R14                      ; Return to calling function
```

- Link register (LR or R14)

main

```
BL function1                 ; Main program
                                ; Call function1 using Branch with Link
                                ; instruction.
                                ; PC function1 and
                                ; LR the next instruction in main
```

...

function1

...

; Program code for function 1

```
BX LR ; Return
```

5. ARM Programming – Special Register

- Special registers can only be accessed via MSR and MRS instructions

MRS	<reg>, <special_reg>	; Read special register
MSR	<special_reg>, <reg>	; write to special register

- ASP can be changed by using MSR instruction, but EPSR and IPSR are read-only

MRS	r0, APSR	; Read Flag state into R0
MRS	r0, IPSR	; Read Exception/Interrupt state
MRS	r0, EPSR	; Read Execution state
MSR	APSR, r0	; Write Flag state
MRS	r0, PSR	; Read the combined program status word
MSR	PSR, r0	; Write combined program state word

5. ARM Programming – Special Register

- To access the Control register, the MRS and MSR instructions are used:

MRS	r0, CONTROL	; Read CONTROL register into R0
MSR	CONTROL, r0	; Write R0 into CONTROL register

Bit	Function
CONTROL[1]	Stack status: 1 = Alternate stack is used 0 = Default stack (MSP) is used If it is in the Thread or base level, the alternate stack is the PSP. There is no alternate stack for handler mode, so this bit must be zero when the processor is in handler mode.
CONTROL[0]	0 = Privileged in Thread mode 1 = User state in Thread mode If in handler mode (not Thread mode), the processor operates in privileged mode.

5. ARM Programming

- 16-Bit Load and Store Instructions

Instruction	Function
LDR	Load word from memory to register
LDRH	Load halfword from memory to register
LDRB	Load byte from memory to register
LDRSH	Load halfword from memory, sign extend it, and put it in register
LDRSB	Load byte from memory, sign extend it, and put it in register
STR	Store word from register to memory
STRH	Store half word from register to memory
STRB	Store byte from register to memory
LDMIA	Load multiple increment after
STMIA	Store multiple increment after
PUSH	Push multiple registers
POP	Pop multiple registers

5. ARM Programming

- 16-Bit Branch Instructions

Instruction	Function
B	Branch
B<cond>	Conditional branch
BL	Branch with link; call a subroutine and store the return address in LR
BLX	Branch with link and change state (BLX <reg> only) ¹
CBZ	Compare and branch if zero (architecture v7)
CBNZ	Compare and branch if nonzero (architecture v7)
IT	IF-THEN (architecture v7)

5. ARM Programming – Arithmetic Instructions

Instruction	Operation
ADD Rd, Rn, Rm ; Rd = Rn + Rm ADD Rd, Rm ; Rd = Rd + Rm ADD Rd, #immed ; Rd = Rd + #immed	ADD operation
ADC Rd, Rn, Rm ; Rd = Rn + Rm + carry ADC Rd, Rm ; Rd = Rd + Rm + carry ADC Rd, #immed ; Rd = Rd + #immed + ; carry	ADD with carry
ADDW Rd, Rn, #immed ; Rd = Rn + #immed	ADD register with 12-bit immediate value
SUB Rd, Rn, Rm ; Rd = Rn - Rm SUB Rd, #immed ; Rd = Rd - #immed SUB Rd, Rn, #immed ; Rd = Rn - #immed	SUBTRACT
SBC Rd, Rm ; Rd = Rd - Rm - ; carry flag SBC.W Rd, Rn, #immed ; Rd = Rn - #immed - ; carry flag SBC.W Rd, Rn, Rm ; Rd = Rn - Rm - ; carry flag	SUBTRACT with borrow (carry)
RSB.W Rd, Rn, #immed ; Rd = #immed - Rn RSB.W Rd, Rn, Rm ; Rd = Rm - Rn MUL Rd, Rm ; Rd = Rd * Rm MUL.W Rd, Rn, Rm ; Rd = Rn * Rm	Reverse subtract Multiply

5. ARM Programming – IF-THEN

- The IF-THEN (IT) instructions allow up to four succeeding instructions (called an *IT block*) to be conditionally executed.
- They are in the following formats:

IT<x> <cond>

IT<x><y> <cond>

IT<x><y><z> <cond>

where:

- *<x> specifies the execution condition for the second instruction*
- *<y> specifies the execution condition for the third instruction*
- *<z> specifies the execution condition for the fourth instruction*

5. ARM Programming – IF-THEN

Symbol	Condition	Flag
EQ	Equal	Z set
NE	Not equal	Z clear
CS/HS	Carry set/unsigned higher or same	C set
CC/LO	Carry clear/unsigned lower	C clear
MI	Minus/negative	N set
PL	Plus/positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N set or V set, or N clear and V clear ($N == V$)
LT	Signed less than	N set and V clear, or N clear and V set ($N != V$)
GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear ($Z == 0, N == V$)
LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set ($Z == 1$ or $N != V$)
AL	Always (unconditional)	—

5. ARM Programming – IF-THEN

- An example of a simple conditional execution

```
if (R1<R2) then
    R2=R2-R1
    R2=R2/2
else
    R1=R1-R2
    R1=R1/2
```

- In assembly:

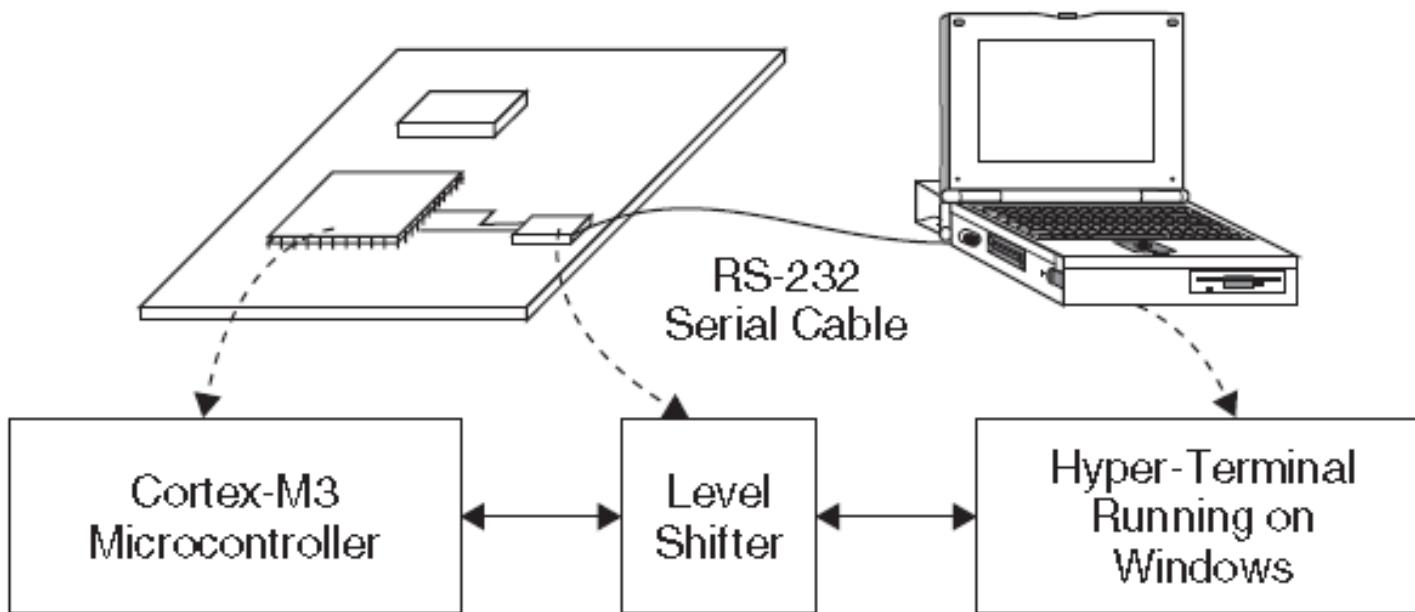
```
CMP    R1, R2      ; If R1 < R2 (less then)
ITTEE   LT          ; then execute instruction 1 and 2
           ; (indicated by T)
           ; else execute instruction 3 and 4
           ; (indicated by E)
SUBLT.W R2,R1      ; 1st instruction
LSRLT.W R2,#1       ; 2nd instruction
SUBGE.W R1,R2       ; 3rd instruction (notice the GE is opposite of LT)
LSRG.E.W R1,#1       ; 4th instruction
```

5. ARM Programming – Using Data Memory

```
STACK_TOP      EQU 0x20002000 ; constant for SP starting value
                AREA | Header Code|, CODE
                DCD STACK_TOP    ; SP initial value
                DCD Start        ; Reset vector
                ENTRY
Start          ; Start of main program, initialize registers
                MOV r0, #10       ; Starting loop counter value
                MOV r1, #0        ; starting result. Calculated 10+9+8+...+1
loop           ADD r1, r0      ; R1 = R1 + R0
                SUBS r0, #1      ; Decrement R0, update flag ("S" suffix)
                BNE loop        ; If result not zero jump to loop; Result is now in R1
                LDR r0,=MyData1 ; Put address of MyData1 into R0
                STR r1,[r0]      ; Store the result in MyData1
                B deadloop      ; Infinite loop
                AREA | Header Data|, DATA
                ALIGN 4
MyData1        DCD 0        ; Destination of calculation result
MyData2        DCD 0
                END            ; End of file
```

5. ARM Programming

- A Low-Cost Test Environment for Outputting Text Messages
 - UART interface is common output method to send messages to a console
 - Hyper-Terminal program can be used as a console



5. ARM Programming

- A simple routine to output a character through UART

```
UART0_BASE      EQU 0x4000C000
UART0_FLAG      EQU UART0_BASE+0x018
UART0_DATA      EQU UART0_BASE+0x000
Putc           ; Subroutine to send a character via UART
                ; Input R0 = character to send
        PUSH {R1,R2, LR}    ; Save registers
        LDR R1,=UART0_FLAG

PutcWaitLoop
        LDR R2,[R1]          ; Get status flag
        TST R2, #0x20         ; Check transmit buffer full flag bit
        BNE PutcWaitLoop ; If busy then loop
        LDR R1,=UART0_DATA ; otherwise
        STRB R0, [R1]          ; Output data to transmit buffer
        POP {R1,R2, PC}       ; Return
```

The register addresses and bit definitions here are just examples

TI's ARM Cortex-M Development Kit



- LM3S9B96 development Kit
 - Stellaris LM3S9B96 MCU with fully-integrated Ethernet, CAN, and USB OTG/Host/Device
 - Bright 3.5" QVGA LCD touch-screen display
 - Navigation POT switch and select pushbuttons
 - Integrated Interchip Sound (I2S) Audio Interface
- The Tiva C Series EK-TM4C123GXL LaunchPad Evaluation Kit
 - A TM4C123G LaunchPad Evaluation board
 - On-board In-Circuit Debug Interface (ICDI)
 - USB Micro-B plug to USB-A plug cable
 - Preloaded RGB quickstart application
 - ReadMe First quick-start guide





Quiz

1. What are different features between ARM Cortex M3 and M4?
2. What are differences between Thumb and Thumb-2 instructions?
3. Compare the features between TM4C and STM32F4 microcontroller
4. What are extra instructions that ARM Cortex-M4 supports?



Assignments

1. Write a program to move 10 words from 0x20000000 to 0x30000000.
2. Write a program to read STATUS register and write to 0x20000004
3. Write a program to write a value in 0x30000000 to CONTROL register
4. Write a subroutine to perform a function $40*X + 50$
5. Write a subroutine to convert data of 10 words form big endian to little endian.
6. Write a program as pseudo code below:

```
if (R0 equal R1) then {
```

```
    R3 = R4 + R5
```

```
    R3 = R3 / 2 }
```

```
else {
```

```
    R3 = R6 + R7
```

```
    R3 = R3 / 2
```

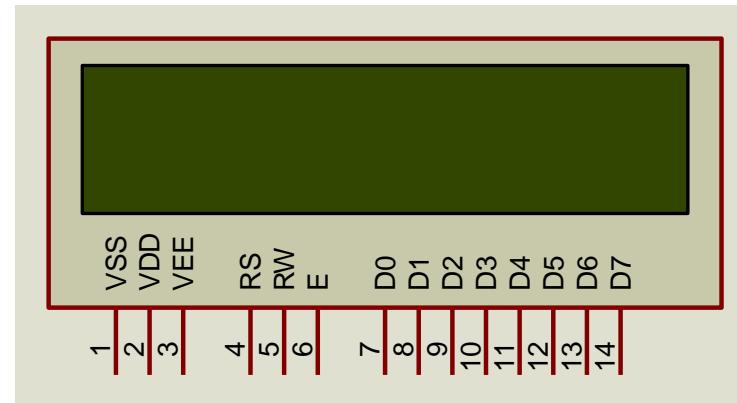
```
}
```

Assignments

- Design a circuit described as follows:
 - Using Cortex-M4 processor LM4F120H5QR
 - Port A connects to 8 single LEDs
 - Port B connects to 8 buttons
 - Write a program to control 8 LEDs by 8 buttons

Assignments

- Design a circuit described as follows:
 - Using Cortex-M4 processor STM32F407VGT6
 - Port A connects to a character LCD
 - Port B connects to 3 buttons START, STOP, CLEAR
 - Write a program to control as follows:
 - START: start to count number in millisecond
 - STOP: stop to count
 - CLEAR: clear the number to zero





Embedded System Design

Chapter 3: C Programming for ARM Microcontroller

1. C Program Basics
2. ARM Cortex-M C Compiler
3. ARM software library
4. Embedded software development tools



1. Basic Programming for Embedded C

- Simple structure for embedded C program

```
#include <...> ; // Library declaration

int x, y, z; // Global variables

void function1 () { } // Function declaration
void funtction2() { }

void main() // main program
{
    int i, j, k; // Local variable
    ...
    // Initialization

    while (1) // main process, loop forever
    {
    }
}
```

Example program

```
#include "inc/lm4f120h5qr.h"
//***** Blinky LED *****
int main(void) {
    volatile unsigned long ulLoop;
    SYSCTL_RCGC2_R = SYSC_TL_RCGC2_GPIOF; // Enable the GPIO port
    ulLoop = SYSCTL_RCGC2_R; // Do a dummy read to insert a few cycles
    GPIO_PORTF_DIR_R = 0x08; // Set the direction as output
    GPIO_PORTF_DEN_R = 0x08; // Enable the GPIO pin for digital function.
    while(1) // Loop forever
    { GPIO_PORTF_DATA_R |= 0x08; // Turn on the LED.
        for(ulLoop = 0; ulLoop < 200000; ulLoop++) { }
        GPIO_PORTF_DATA_R &= ~0x08; // Turn off the LED
        for(ulLoop = 0; ulLoop < 200000; ulLoop++) { }
    }
}
```

Data Types

Type	Size (bits)	Range
unsigned char	8	0 ÷ 255
unsigned short int	8	0 ÷ 255
unsigned int	16	0 ÷ 65535
unsigned long int	32	0 ÷ 4294967295
signed char	8	-128 ÷ 127
signed short int	8	-128 ÷ 127
signed int	16	-32768 ÷ 32767
signed long int	32	-2147483648 ÷ 2147483647
float	32	±1.17549435082E-38 ÷ ±6.80564774407E38
double	32	±1.17549435082E-38 ÷ ±6.80564774407E38
long double	32	±1.17549435082E-38 ÷ ±6.80564774407E38



Keywords for Embedded C (1)

No.	Keyword	Meaning
1	asm	Insert assembly code
2	auto	Specifies a variable as automatic (created on the stack)
3	break	Causes the program control structure to finish
4	case	One possibility within a switch statement
5	char	8 bit integer
6	const	Defines parameter as constant in ROM
7	continue	Causes the program to go to beginning of loop
8	default	Used in switch statement for all other cases
9	do	Used for creating program loops
10	double	Specifies variable as double precision floating point



Keywords for Embedded C (2)

No.	Keyword	Meaning
11	else	Alternative part of a conditional
12	extern	Defined in another module
13	float	Specifies variable as single precision floating point
14	for	Used for creating program loops
15	goto	Causes program to jump to specified location
16	if	Conditional control structure
17	int	16 bit integer (same as short on the 6811 and 6812)
18	long	32 bit integer
19	register	Specifies how to implement a local
20	return	Leave function



Keywords for Embedded C (3)

No.	Keyword	Meaning
21	short	16 bit integer
22	signed	Specifies variable as signed (default)
23	sizeof	Built-in function returns the size of an object
24	static	Stored permanently in memory, accessed locally
25	struct	Used for creating data structures
26	switch	Complex conditional control structure
27	typedef	Used to create new data types
28	unsigned	Always greater than or equal to zero
29	void	Used in parameter list to mean no parameter
30	volatile	Can change implicitly

Scope

- The **scope** of a variable is the portion of the program from which it can be referenced.
- If we declare a local variable with the **same name** as a global object or another local in a superior block, the new variable temporarily supersedes the higher level declarations.

```
unsigned char x; /* a regular global variable*/  
void sub(void){  
    x=1;  
    { unsigned char x; /* a local variable*/  
        x=2;  
        { unsigned char x; /* a local variable*/  
            x=3;  
            PORTA=x;}  
            PORTA=x;}  
            PORTA=x;}  
}
```



Static Variables

- **Static variables** are defined in RAM permanently.
 - **Static global**: can only be accessed within the file where it is defined.
 - **Static local**: can only be accessed within the function where it is defined

```
static short TheGlobal; /* a static global variable*/  
void main(void){  
    TheGlobal=1000;  
}
```

```
void main(void){  
    static short TheLocal; /* a static local variable*/  
    TheLocal=1000;  
}
```

Volatile variables

- **Volatile** is a variable that can change value outside the scope of the function
- Applications:
 - memory-mapped peripheral
 - Global variables which can be changed by interrupts
 - Global variables which are access by many tasks

```
void main(void)
{
    volatile unsigned char *p = (char *) 0x8000;
    while (*p == 0);
}
```



Externals

- Objects that are defined outside of the present source module have the external storage class.
- The compiler knows an external variable by the keyword **extern** that must precede its declaration.
- Only global declarations can be designated extern

```
extern short ExtGlobal; /* an external global variable*/  
void main(void){  
    ExtGlobal=1000;  
}
```

Delay in C programming

- Delay techniques:
 - Loop
 - Simple, not precise
 - Timer / Interrupt
 - Complex, precise

```
void loop_delay()
{
    unsigned int i;
    for(i=0;i<1000;i++);
}
```

De-bouncing

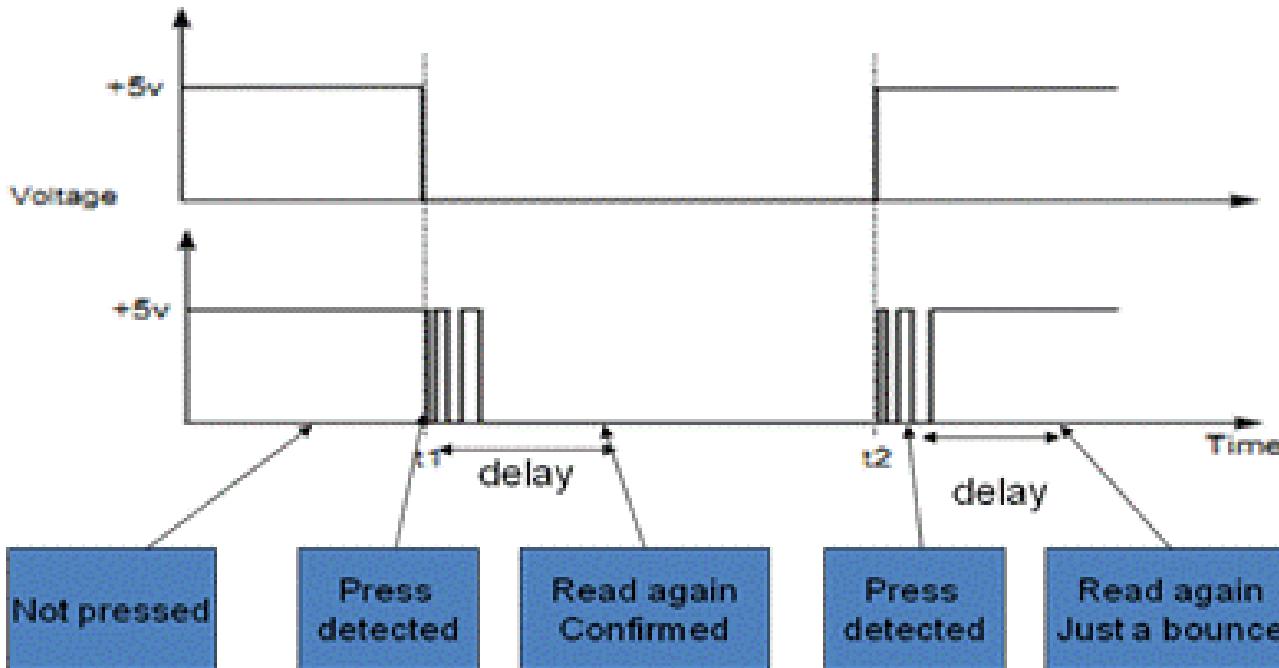
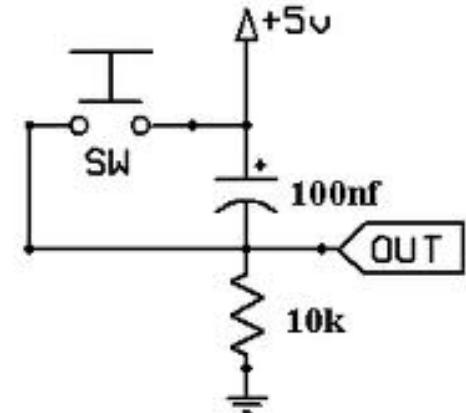
- De-bouncing techniques

- Hardware

- Using a capacitor

- Software

- Check twice the status of the button





Timeout

- Timeout: solve the problem when it has to be waiting an event for long time.
- Solution
 - Counter loop
 - Timer

```
long timeout_loop = TIMEOUT_INIT;  
...  
while(++timeout_loop !=0);
```

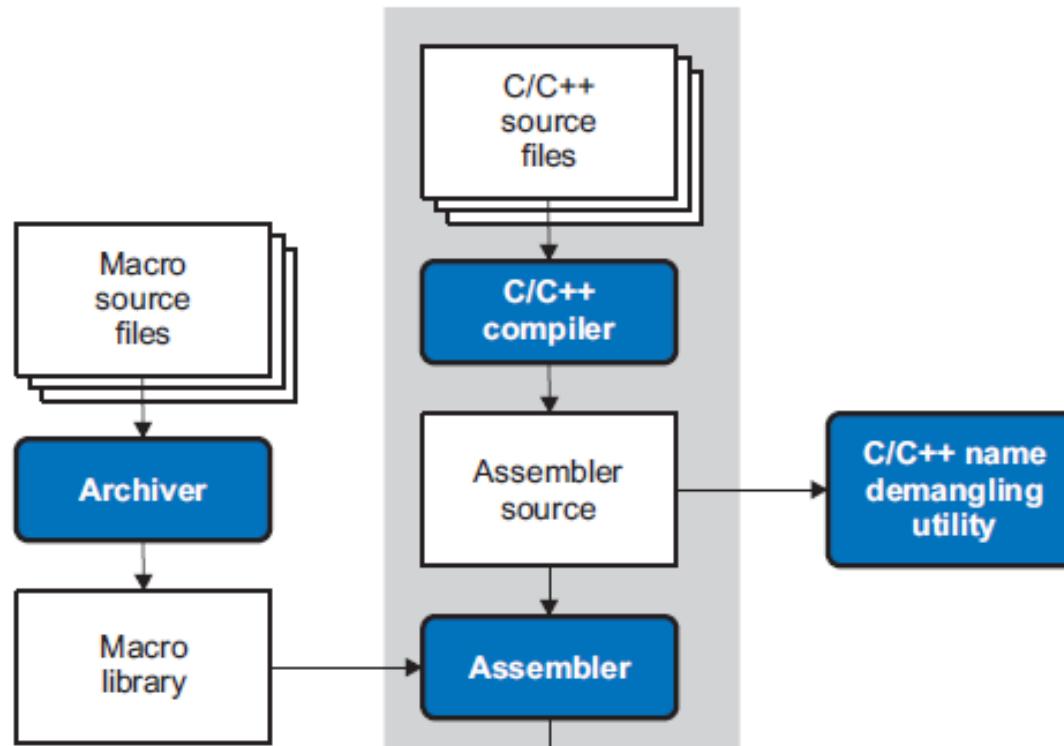


2. ARM Cortex-M C Compiler

- Tool chains:
 - Keil™ RealView® Microcontroller Development Kit
 - MentorGraphics Sourcery CodeBench for ARM EABI
 - IAR Embedded Workbench®
 - Texas Instruments Code Composer Studio™
- References
 - Texas Instrument, “ARM Optimizing C/C++ Compiler”

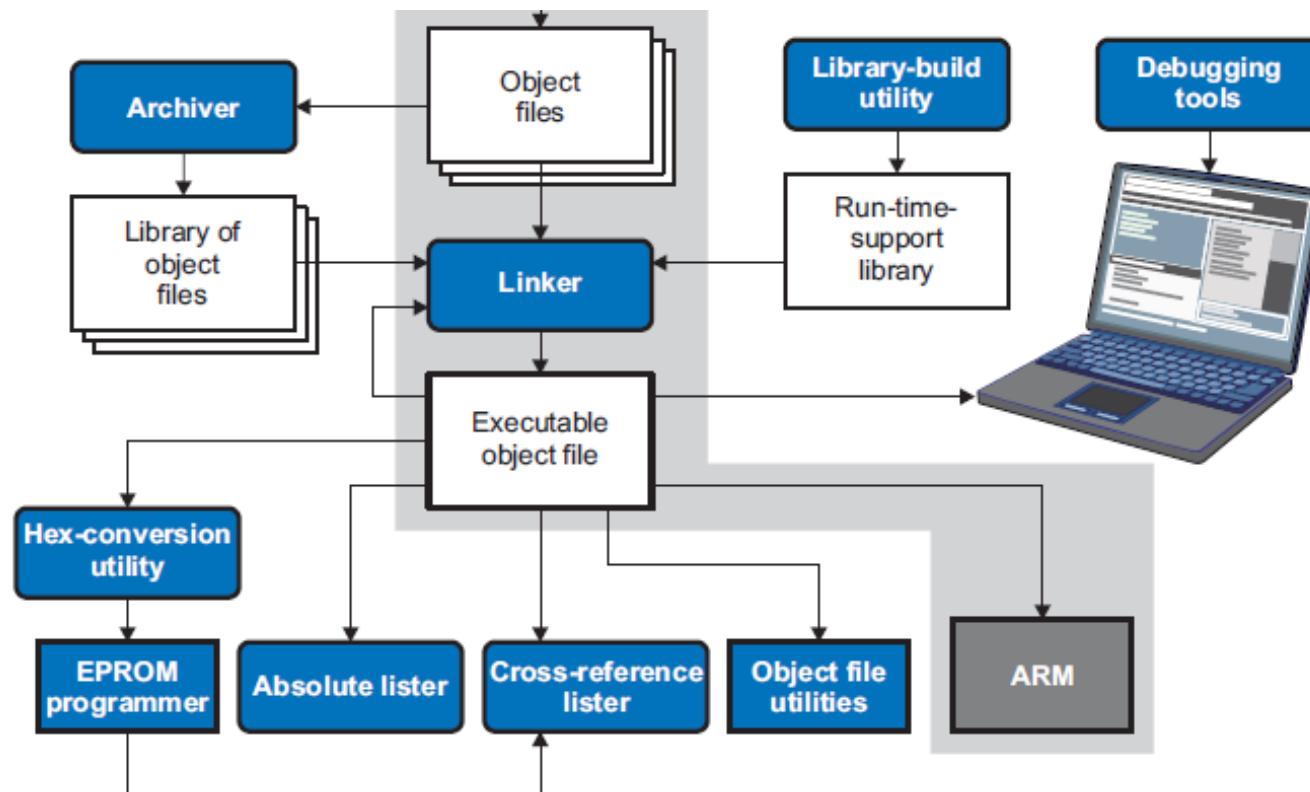
2. ARM Cortex-M C Compiler

- Software development flow for ARM Cortex-M
 - The **compiler** accepts C/C++ source code and produces ARM assembly language source code
 - The **assembler** translates assembly language source files into machine language



2. ARM Cortex-M C Compiler

- The **linker** combines relocatable object files into a single absolute executable object file.
- The **archiver** allows you to collect a group of files into a single archive file, called a library





2. ARM Cortex-M C Compiler

- **armcl [options] [filenames] [--run_linker [link_options] object files]**
 - **Armcl:** Command that runs the compiler and the assembler.
 - **Options:** Options that affect the way the compiler processes input files.
 - **Filenames:** One or more C/C++ source files, assembly language source files, or object files.
 - **--run_linker:** Option that invokes the linker. The --run_linker option's short form is -z.
 - **link_options:** Options that control the linking process.
 - **object files:** Name of the additional object files for the linking process.
- **Example:**
 - `armcl symtab.c file.c seek.asm --run_linker --library=lnk.cmd --output_file=myprogram.out`

2. ARM Cortex-M C Compiler

- Examples:

armcl *c ; compiles and links

armcl --compile_only *.c ; only compiles

armcl *.c --run_linker Lnk.cmd ; compiles and links
using a command file

armcl --compile_only *.c --run_linker Lnk.cmd
; only compiles (--compile_only overrides --
run_linker)

2. ARM Cortex-M C Compiler

- Invoking the Linker Separately

```
armcl --run_linker {--rom_model | --ram_model} filenames  
[options] [--output_file= name.out] --library= library [Ink.cmd]
```

Example:

- armcl --run_linker --rom_model prog1 prog2 prog3 --
output_file=prog.out --library=rtsv4_A_be_eabi.lib

3. ARM Software Library

- TI's ARM Cortex-M microcontroller
 - StellarisWare
 - TivaWare
- ST's ARM Cortex-M microcontroller
 - ST8 firmware
 - STM32 firmware
- Documents
 - Texas Instruments, "StellarisWare Peripheral Driver Library", 2013,
www.ti.com/stellarisware
 - Texas Instruments, "TivaWare Peripheral Driver Library", 2013,
www.ti.com/tiva-c
 - ST Electronics, "STM32 MCUs Software", 2013,
<http://www.st.com/web/en/catalog/tools/FM147/CL1794/SC961>





StellarisWare

- an **extensive suite of software** designed to simplify and speed development of Stellaris-based microcontroller applications
- operates with **all LM3S and LM4F series Stellaris MCUs**
- StellarisWare software includes:
 - Stellaris Peripheral Driver Library
 - Stellaris Graphics Library
 - Stellaris USB Library
 - Stellaris Code Examples



TivaWare

- On 15 Apr. 2012, TI recommends that new design should use TivaWare and Tiva family MCUs
- TivaWare for C Series library includes:
 - TivaWare Peripheral Driver Library
 - TivaWare Graphics Library
 - TivaWare USB Library
 - TivaWare IQMath Library





GPIO

- **GPIO driver:**
 - Driverlib/gpio.c
 - Driverlib/gpio.h
- **Most useful functions:**
 - long **GPIOPinRead**(unsigned long ulPort, unsigned char ucPins)
 - void **GPIOPinWrite**(unsigned long ulPort, unsigned char ucPins, unsigned char ucVal)
- **Examples:**
 - X = GPIOPINRead(GPIO_PORTF_BASE, GPIO_PIN_0);
 - GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3 | GPIO_PIN_2, 7);

GPIO

- void **GPIODirModeSet**(unsigned long ulPort, unsigned char ucPins, unsigned long ulPinIO)
 - Description: Sets the direction and mode of the specified pin(s).
 - **ulPort** is the base address of the GPIO port
 - **ucPins** is the bit-packed representation of the pin(s).
 - **ulPinIO** is the pin direction and/or mode.
 - GPIO_DIR_MODE_IN: software controlled input
 - GPIO_DIR_MODE_OUT: software controlled output
 - GPIO_DIR_MODE_HW: under hardware control

System Clock

- void **SysCtlClockSet** (unsigned long ulConfig)
 - This function configures the clocking of the device
- ulConfig:
 - **Clock divider:** SYSCTL_SYSDIV_1, SYSCTL_SYSDIV_2, ...
SYSCTL_SYSDIV_64
 - **Use of PLL:** SYSCTL_USE_PLL, SYSCTL_USE_OSC
 - **External crystal frequency:** SYSCTL_XTAL_1MHZ,
SYSCTL_XTAL_4MHZ, SYSCTL_XTAL_8MHZ,
 - **Oscillator source:** SYSCTL_OSC_MAIN, SYSCTL_OSC_INT
- Examples:
 - SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL |
SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);

System Clock

- `unsigned long SysCtlClockGet(void)`
 - return the processor clock rate
- `void SysCtlDelay(unsigned long ulCount)`
 - Provides a small delay.
 - `ulCount` is the number of delay loop
 - The loop takes 3 cycles/loop
- Example: delay 0.1s
 - `SysCtlDelay(SysCtlClockGet() / 10 / 3);`

Class Assignment

The following assignments are applied for MCU **LM4F120H5QR**

1. Write a program to generate a clock signal 0.5Hz at **PF1**
2. Write a function to read the status of a button with de-bounced capability.
3. Write a program to control 8 single LEDs at port PB. Each LED is ON alternately from LSB LED to MSB LED.
4. Write a program to control 7-segment LED with the control signal A,B,C,D,E,F,G at port PB0 to PB6. The 7-segment LED shows the counted number form 0 to 9 for every 0.5s.
5. Write a function to read a 4x4 matrix keyboard with 16 buttons using key-scanning method.

STM32F4 Library - GPIO

- **GPIO configuration**

```
GPIO_InitTypeDef GPIO_InitStructure;//Declare a variable GPIO_InitStructure
```

Example 1:

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12 | GPIO_Pin_13| GPIO_Pin_14| GPIO_Pin_15;  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;  
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;  
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;  
GPIO_Init(GPIOD, &GPIO_InitStructure);
```

Example 2:

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All;  
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;  
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;  
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;  
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;  
GPIO_Init(GPIOD, &GPIO_InitStructure);
```

STM32F4 Library - GPIO

- `uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx,
 uint16_t GPIO_Pin)`
 - `GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_12);`
- `uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx)`
 - `uint16_t D = GPIO_ReadInputData(GPIOD);`
- `uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef* GPIOx,
 uint16_t GPIO_Pin)`
 - `GPIO_ReadOutputDataBit(GPIOD, GPIO_Pin_12);`
- `uint16_t GPIO_ReadOutputData(GPIO_TypeDef* GPIOx)`
 - `uint16 X = GPIO_ReadOutputData(GPIOD);`

STM32F4 Library - GPIO

- void **GPIO_SetBits**(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
 - GPIO_SetBits(GPIOD, GPIO_Pin_12);
 - GPIO_SetBits(GPIOB, GPIO_Pin_13);
- void **GPIO_ResetBits**(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
 - GPIO_ResetBits(GPIOD, GPIO_Pin_12 | GPIO_Pin_13);
- void **GPIO_Write**(GPIO_TypeDef* GPIOx, uint16_t PortVal)
 - GPIO_Write(GPIOB, 0x000F);
- void **GPIO_ToggleBits**(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
 - GPIO_ToggleBits(GPIOC, GPIO_Pin_1);

STM32F4 Library - Systick

- `uint32_t SysTickConfig (uint32_t ticks)`
 - `SysTick_Config(SystemCoreClock / 1000);`
 - `Delay(100); //delay 100 ms`

```
void Delay(__IO uint32_t nTime)
{
    TimingDelay = nTime;
    while(TimingDelay != 0);
}
```

```
void TimingDelay_Decrement(void)
{
    if (TimingDelay != 0x00)
        TimingDelay--;
}
```

```
void SysTick_Handler(void)
{
    TimingDelay_Decrement();
}
```

Assignments

The following assignments are applied for MCU **STM32F407VGT6**

1. Write a program to generate a clock signal 5Hz at PB0, and a clock signal 10Hz at PB1.
2. Write a program to count a 8bit number and display on 8 single LEDs at port PB[7:0].
3. Write a program described as followings:
 1. There are 3 buttons: START, STOP, MODE
 2. 8 single LEDs are controlled by Port B [7:0]
 3. START: display LEDs according to the MODE
 4. STOP: turn off all the LEDS
 5. MODE: change the Mode for LED display
 1. Mode 1: each LED turns ON from LED0 to LED7
 2. Mode 2: each LED turns ON from LED7 to LED0

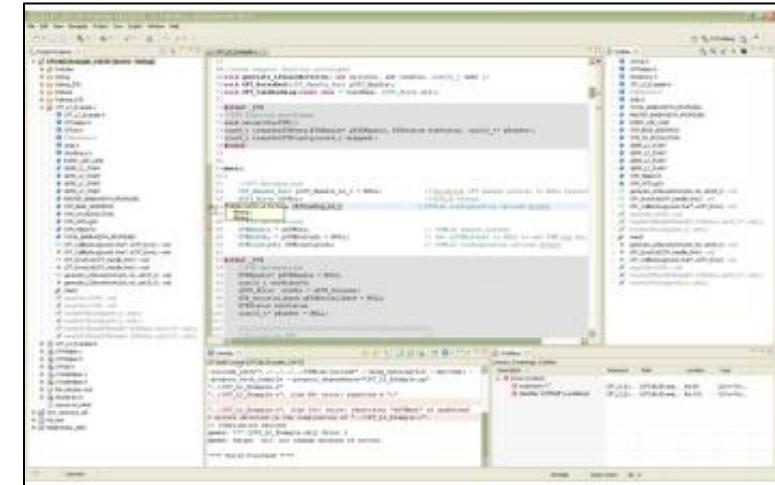


4. Embedded software development tools

- Code Composer Studio
- Keil µVision

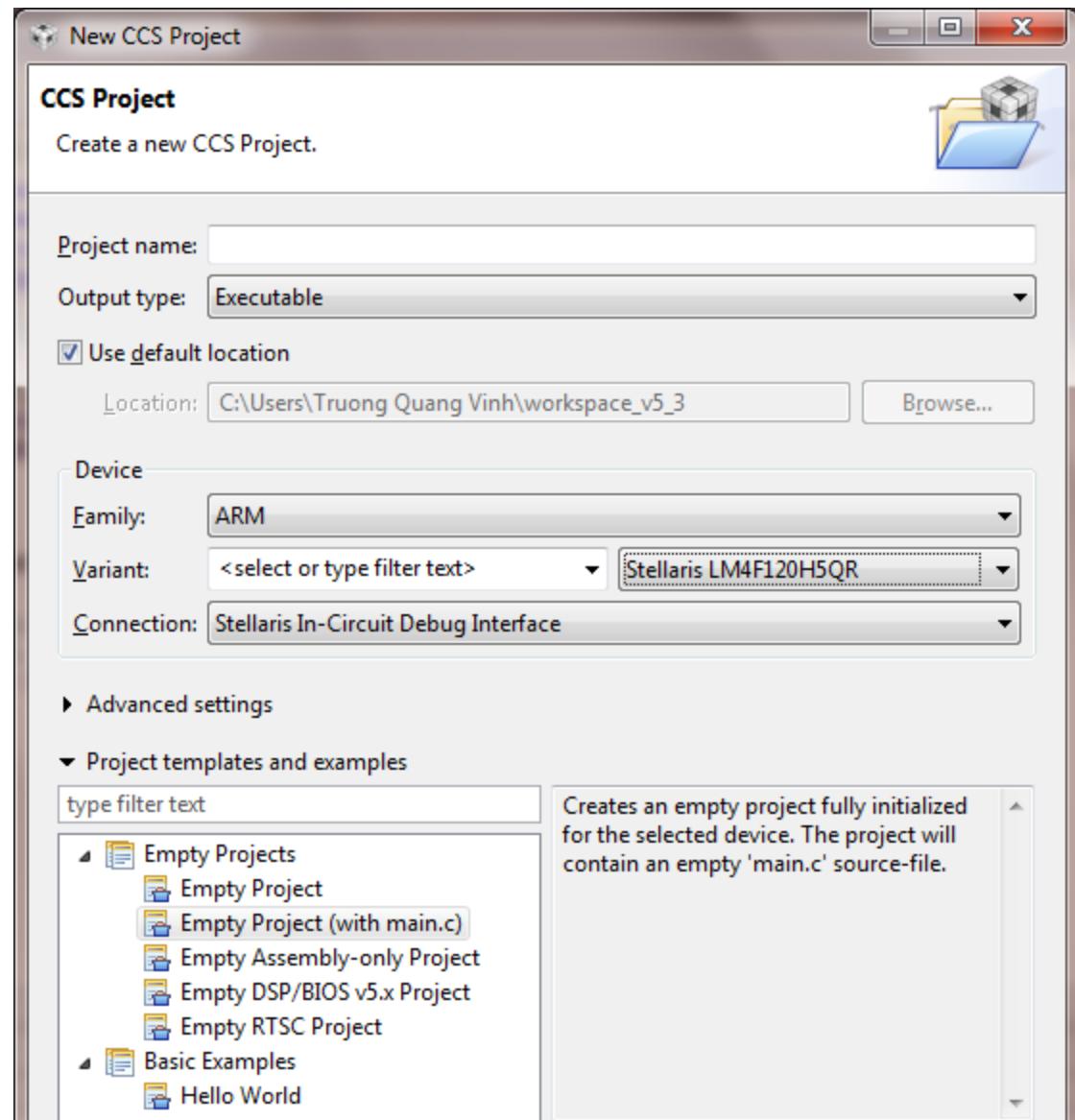
1. Code Composer Studio - Overview

- Code Composer Studio™ (CCStudio) is an integrated development environment (IDE) for Texas Instruments (TI) **embedded processor families**.
- CCStudio comprises a suite of tools used to develop and debug **embedded applications**



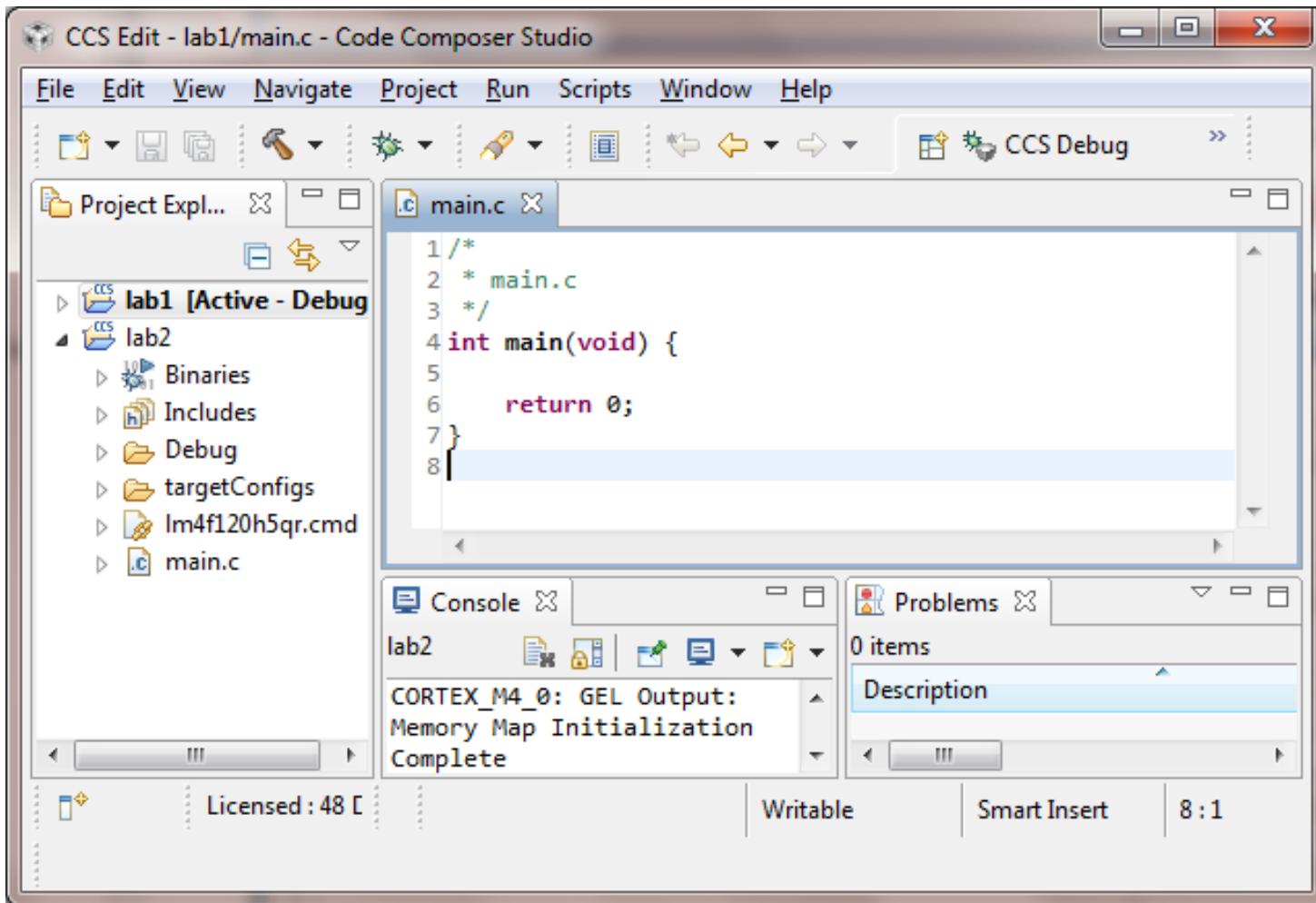
1. Code Composer Studio – Create a project

- Create new project
 - Family: ARM
 - Variant: LM4F120H5QR
 - Connection: ICDI
 - Project templates: Empty project



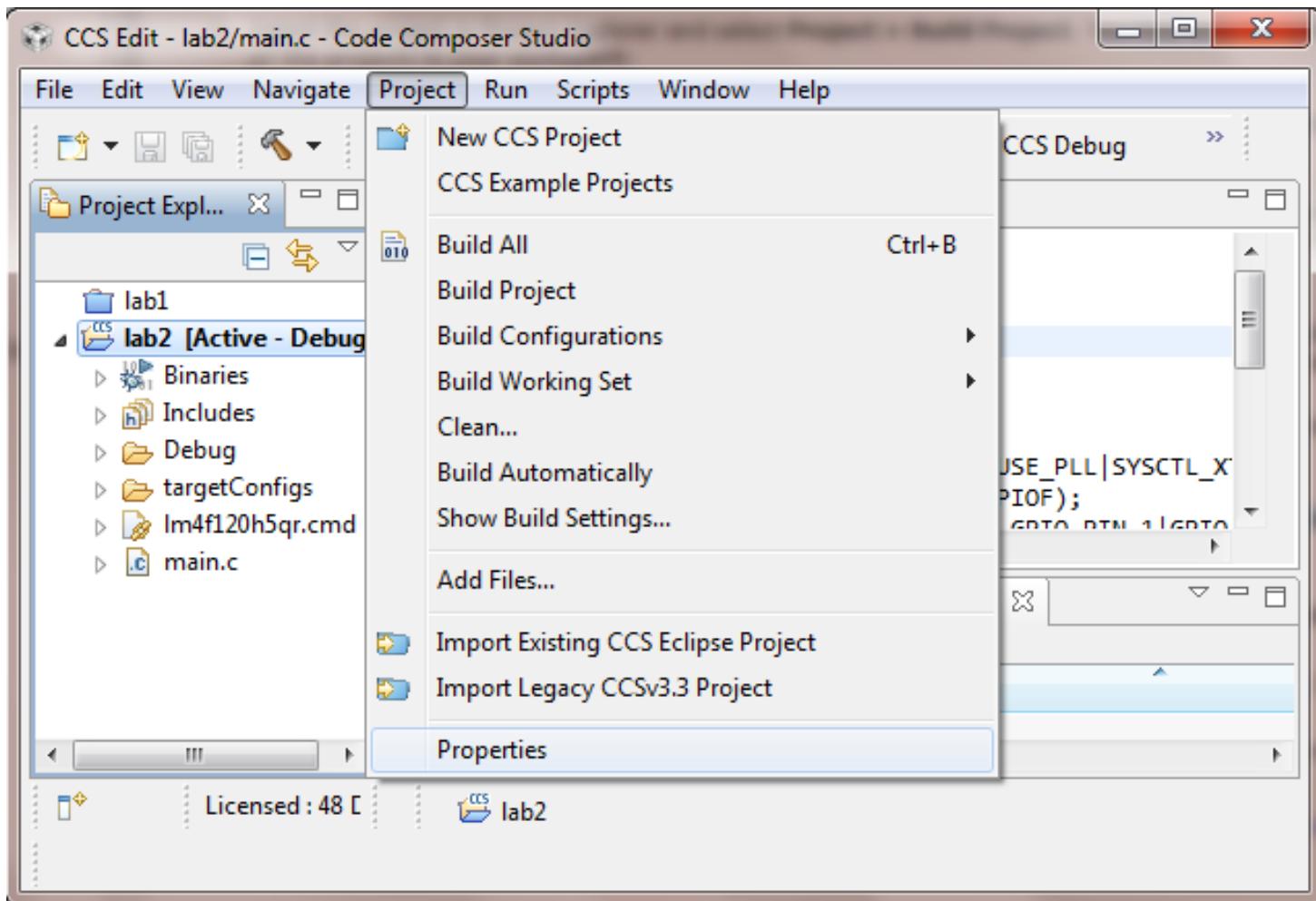
1. Code Composer Studio – Project view

- C/C++ Projects view



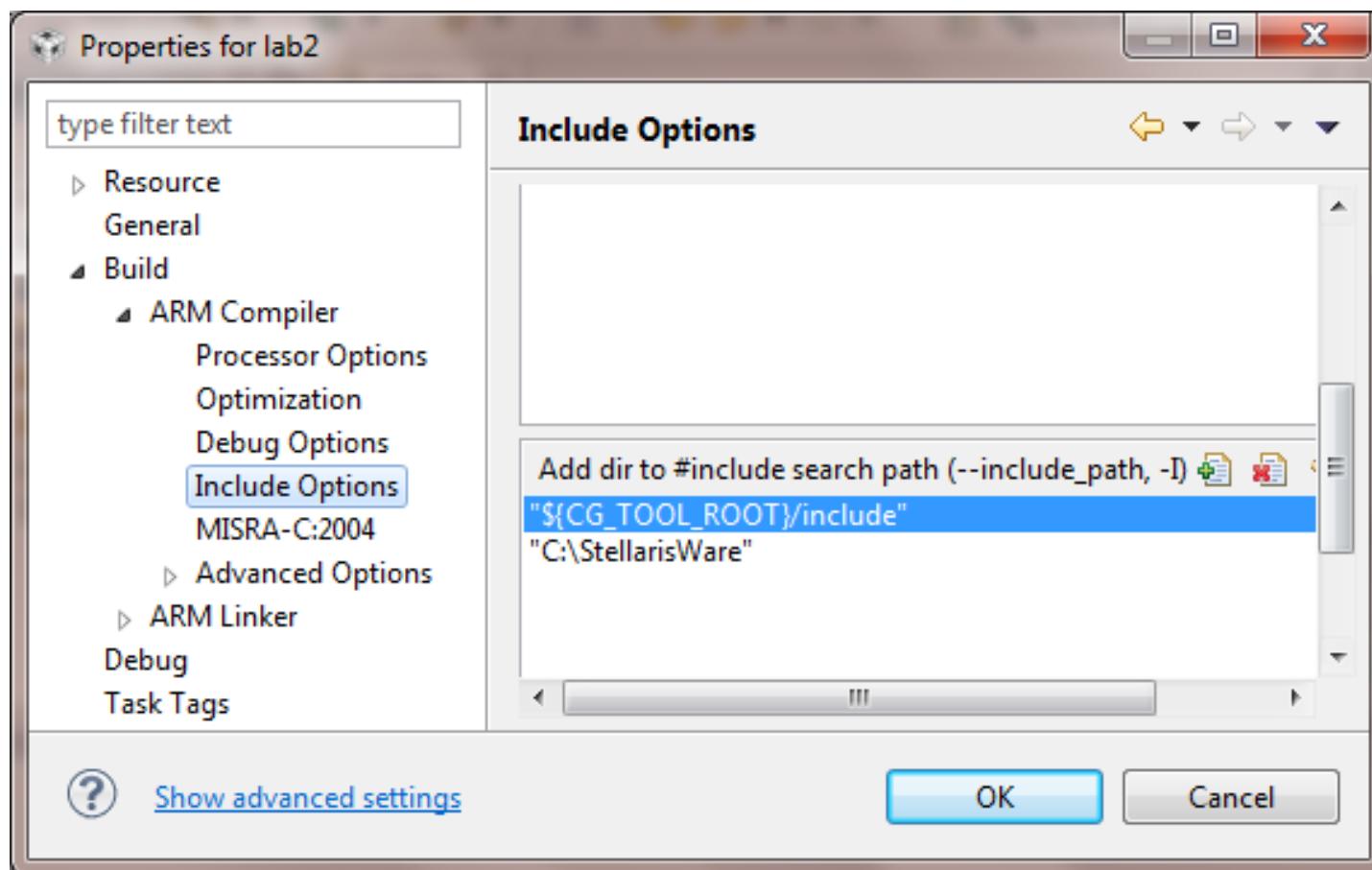
1. Code Composer Studio – Properties

- Setup properties...



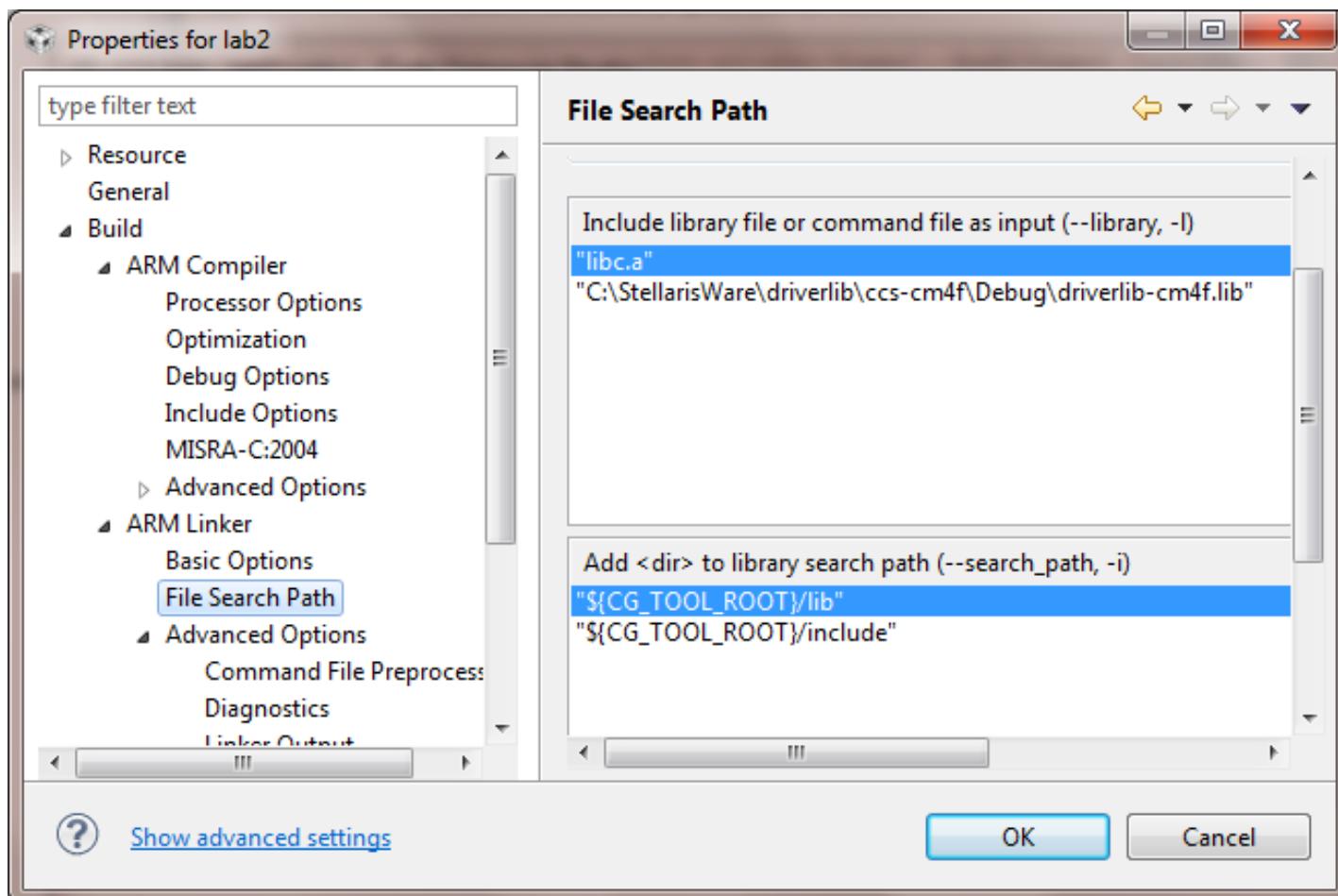
1. Code Composer Studio – Properties

- Include options:
 - C:\StellarisWare



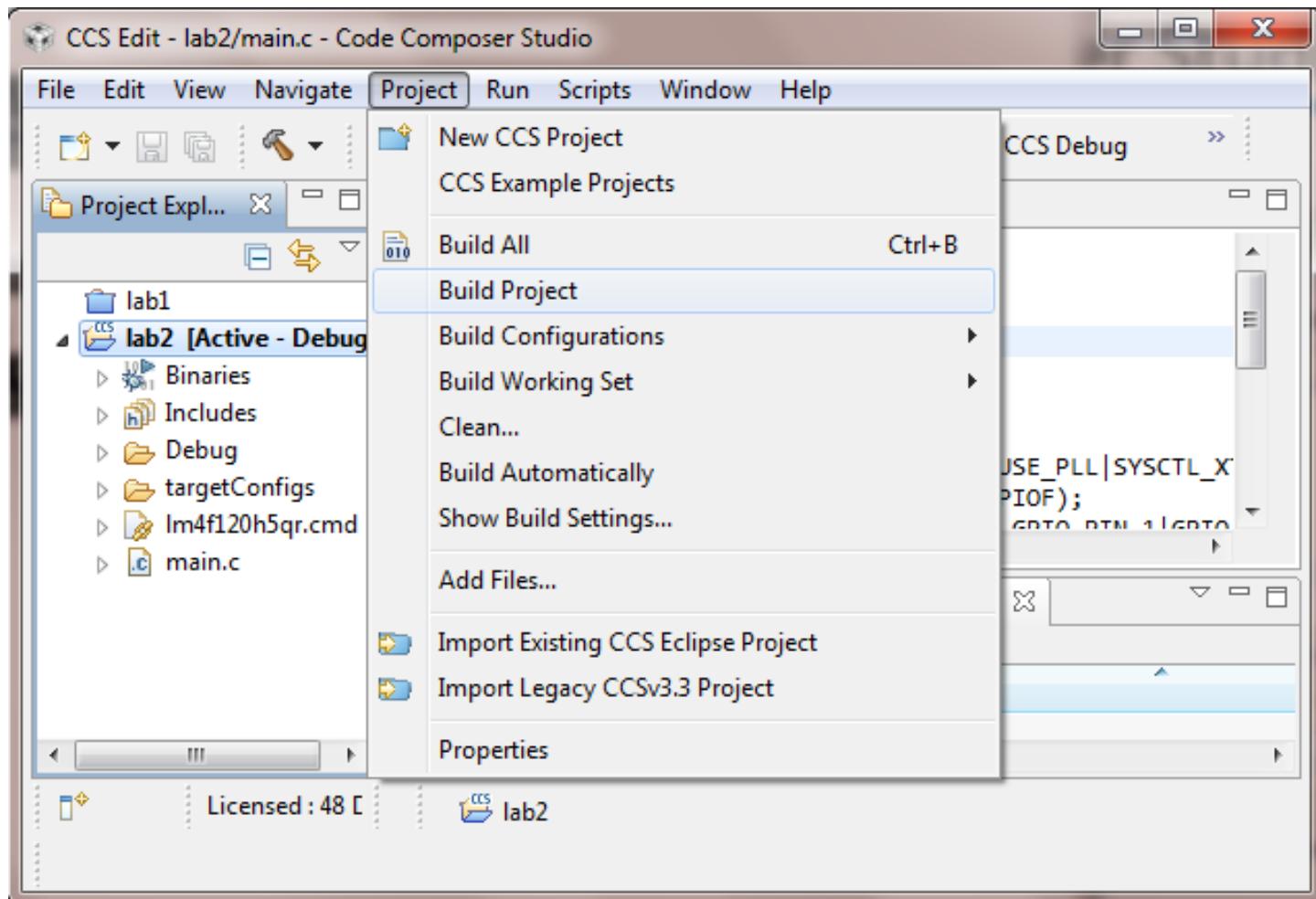
1. Code Composer Studio – Properties

- File Search Path
 - Driverlib-cm4f.lib



1. Code Composer Studio – Build a project

- Build your project



1. Code Composer Studio – Debug

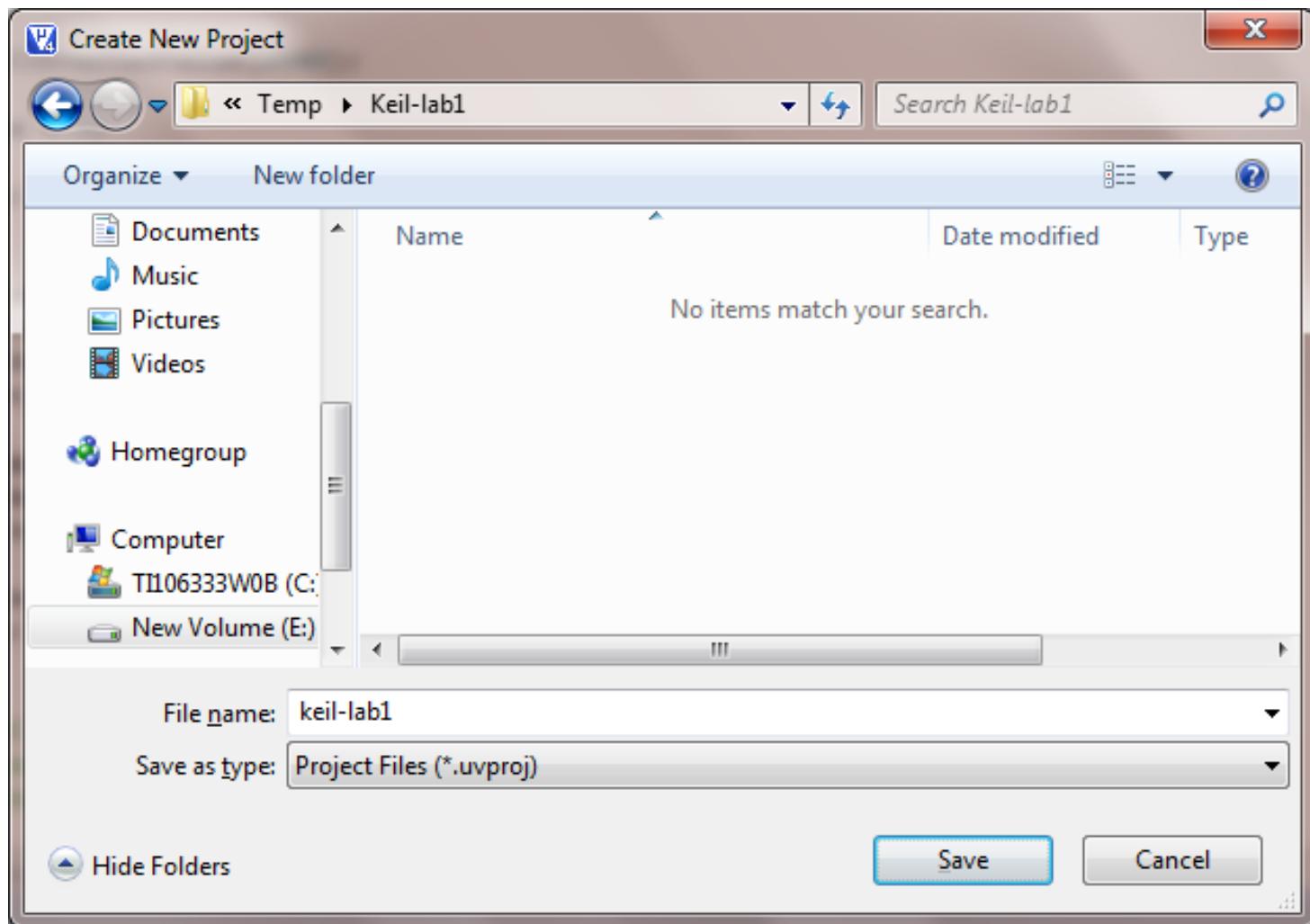
Image	Name	Description	Availability
	New Target Configuration	Creates a new target configartion file.	File New Menu Target Menu
	Debug	Opens a dialog to modify existing debug configurations. Its drop down can be used to access other launching options.	Debug Toolbar Target Menu
	Connect Target	Connect to hardware targets.	TI Debug Toolbar Target Menu Debug View Context Menu
	Terminate All	Terminates all active debug sessions.	Target Menu Debug View Toolbar

2. Keil Tools by ARM

- The Keil products from ARM support over 700 of the most popular ARM microcontrollers.
 - Includes RealView® Compilation Tools including C/C++ Compiler, Macro Assembler, and Linker
 - Includes Debuggers, Real-time Kernels, Single-board Computers, and Emulators
 - All tools are integrated into µVision which provides interfaces to ULINK and other third-party debug adapters.

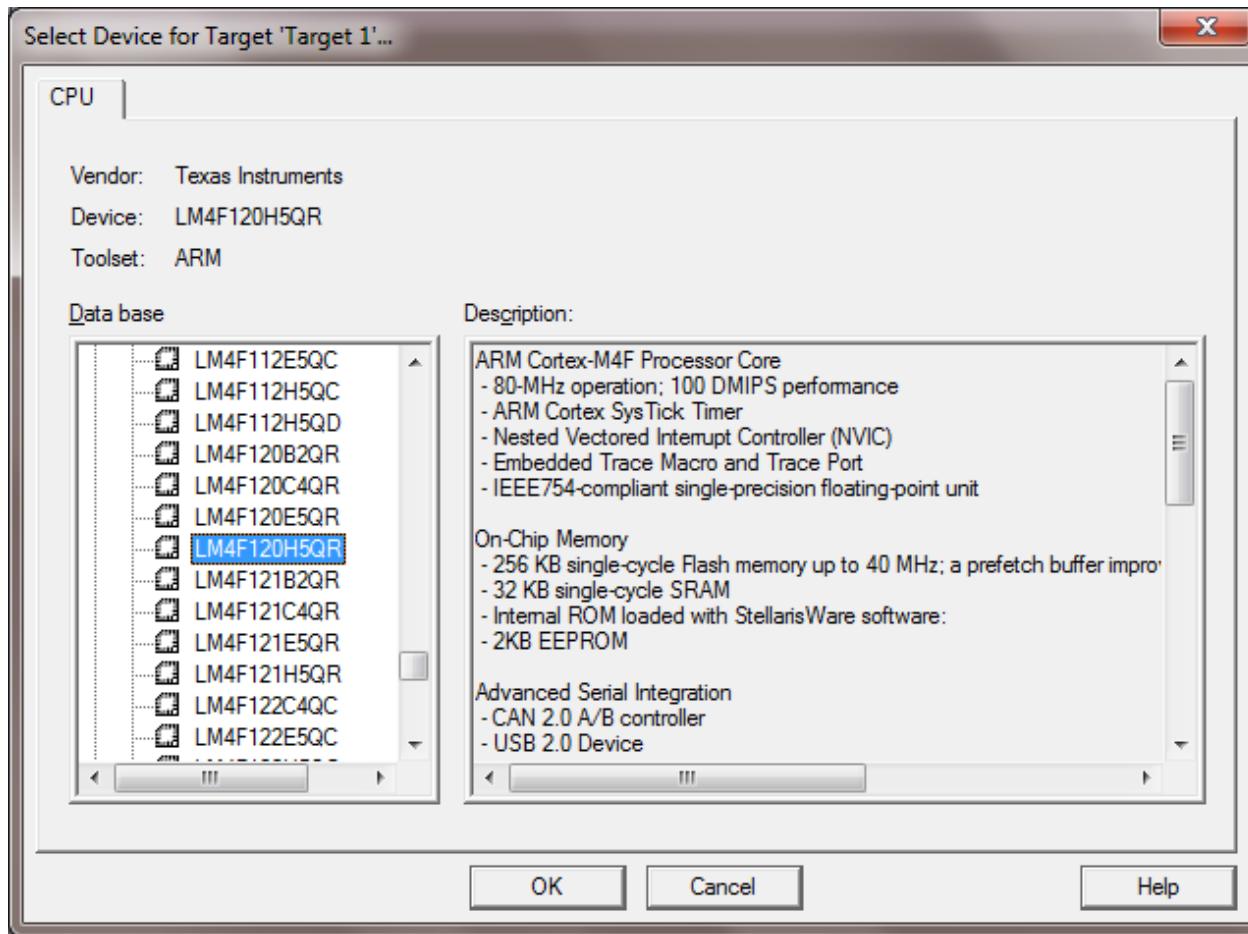


2. Keil Tools by ARM – Create a project



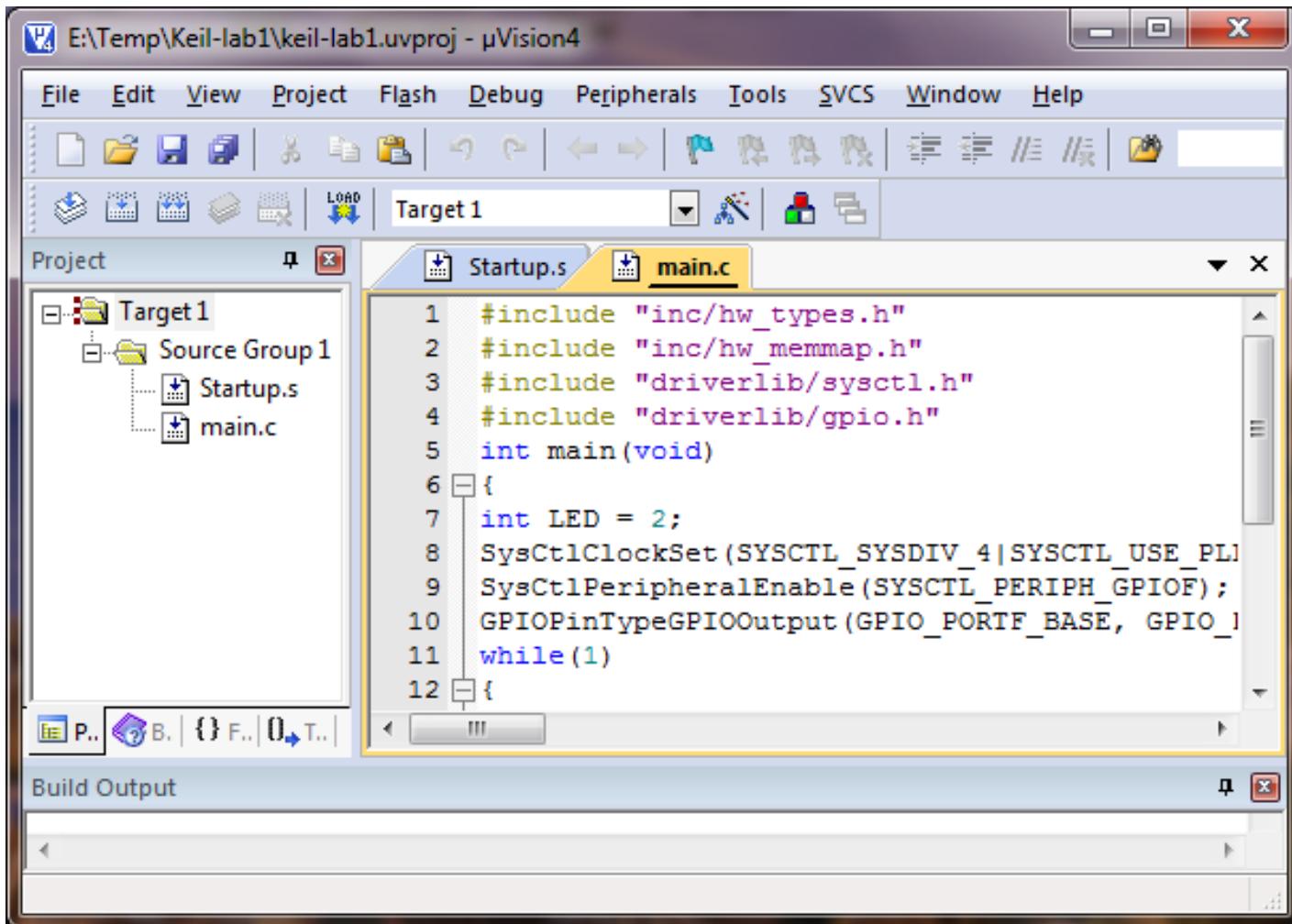
2. Keil Tools by ARM – Select device

- Vendor: Texas Instruments
- Device: LM4F120H5QR



2. Keil Tools by ARM – Add files

- Add main.c

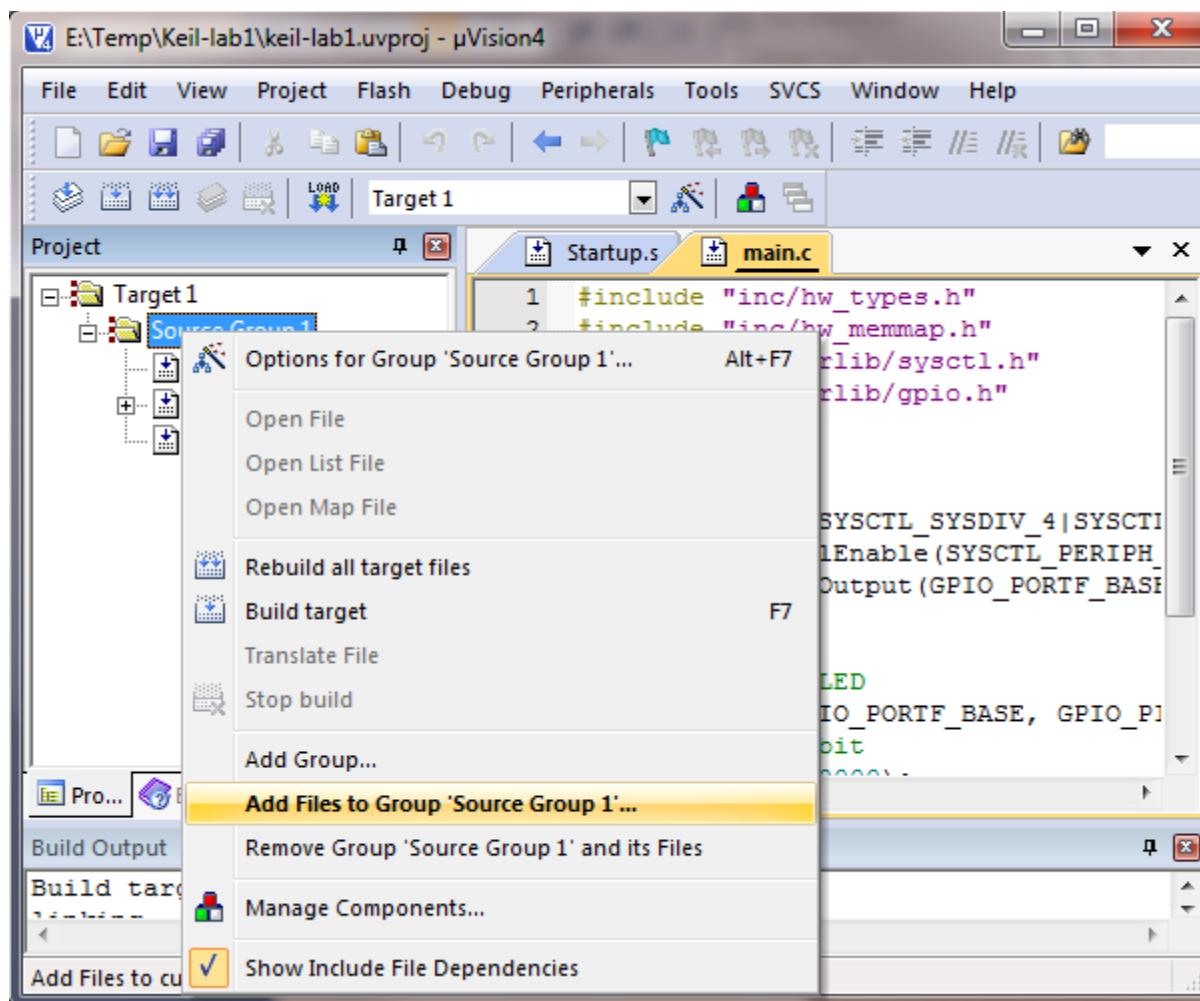


The screenshot shows the Keil µVision 4 IDE interface. The title bar reads "E:\Temp\Keil-lab1\keil-lab1.uvproj - µVision4". The menu bar includes File, Edit, View, Project, Flash, Debug, Peripherals, Tools, SVCS, Window, and Help. The toolbar has various icons for file operations like Open, Save, and Build. The target selection bar shows "Target 1". The left pane is the "Project" view, which displays a tree structure for "Target 1" under "Source Group 1", showing "Startup.s" and "main.c". The right pane is the code editor, currently displaying the "main.c" file. The code in "main.c" is:

```
1 #include "inc/hw_types.h"
2 #include "inc/hw_memmap.h"
3 #include "driverlib/sysctl.h"
4 #include "driverlib/gpio.h"
5 int main(void)
6 {
7     int LED = 2;
8     SysCtlClockSet(SYSCLOCK_SYSCLK_4 | SYSCLOCK_USE_PLL);
9     SysCtlPeripheralEnable(SYSCLOCK_PERIPH_GPIOF);
10    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_1);
11    while(1)
12    {
```

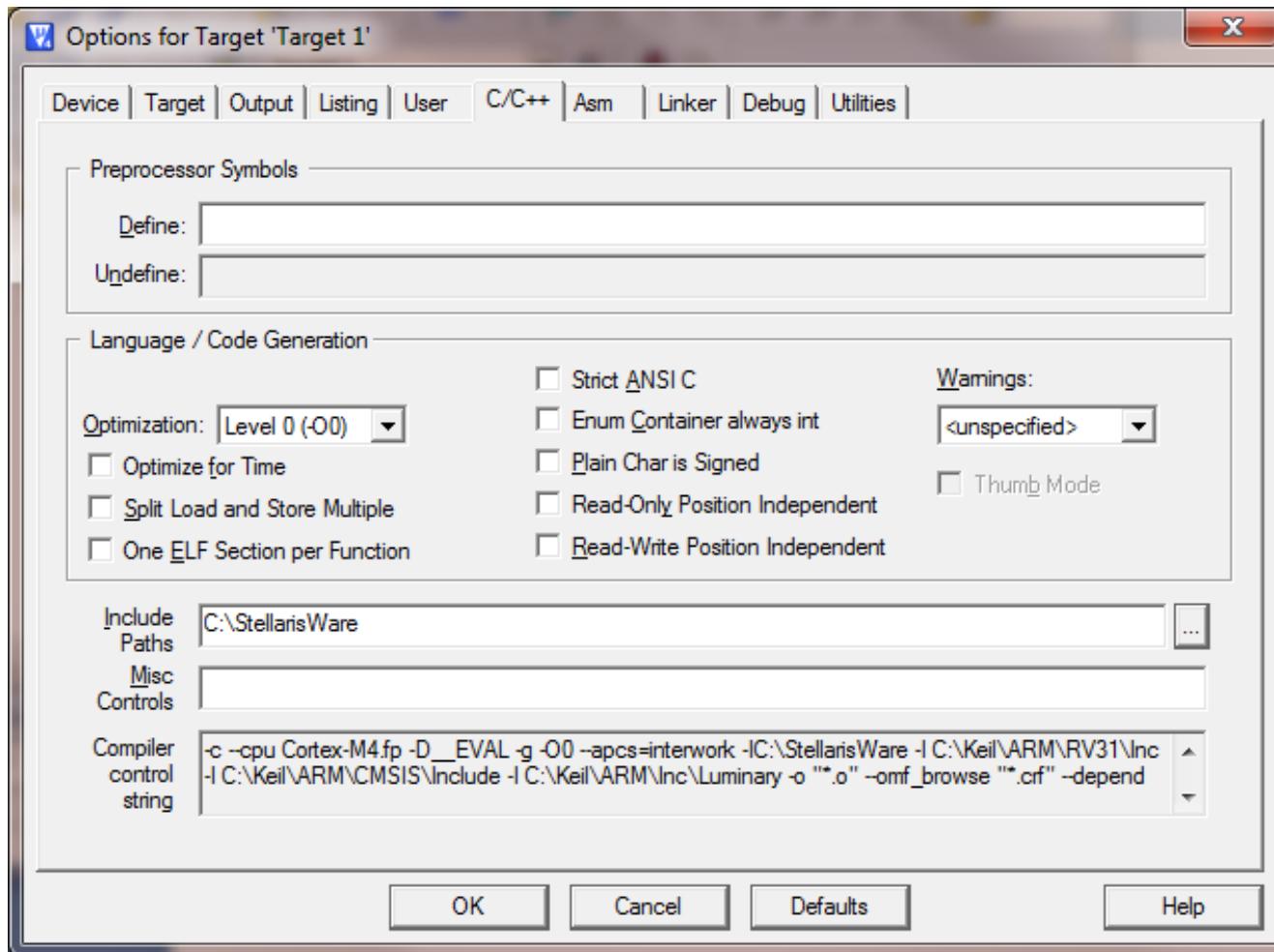
2. Keil Tools by ARM – Add files

- Add C:\StellarisWare\driverlib\rvmdk-cm4\driverlib-cm4f.lib

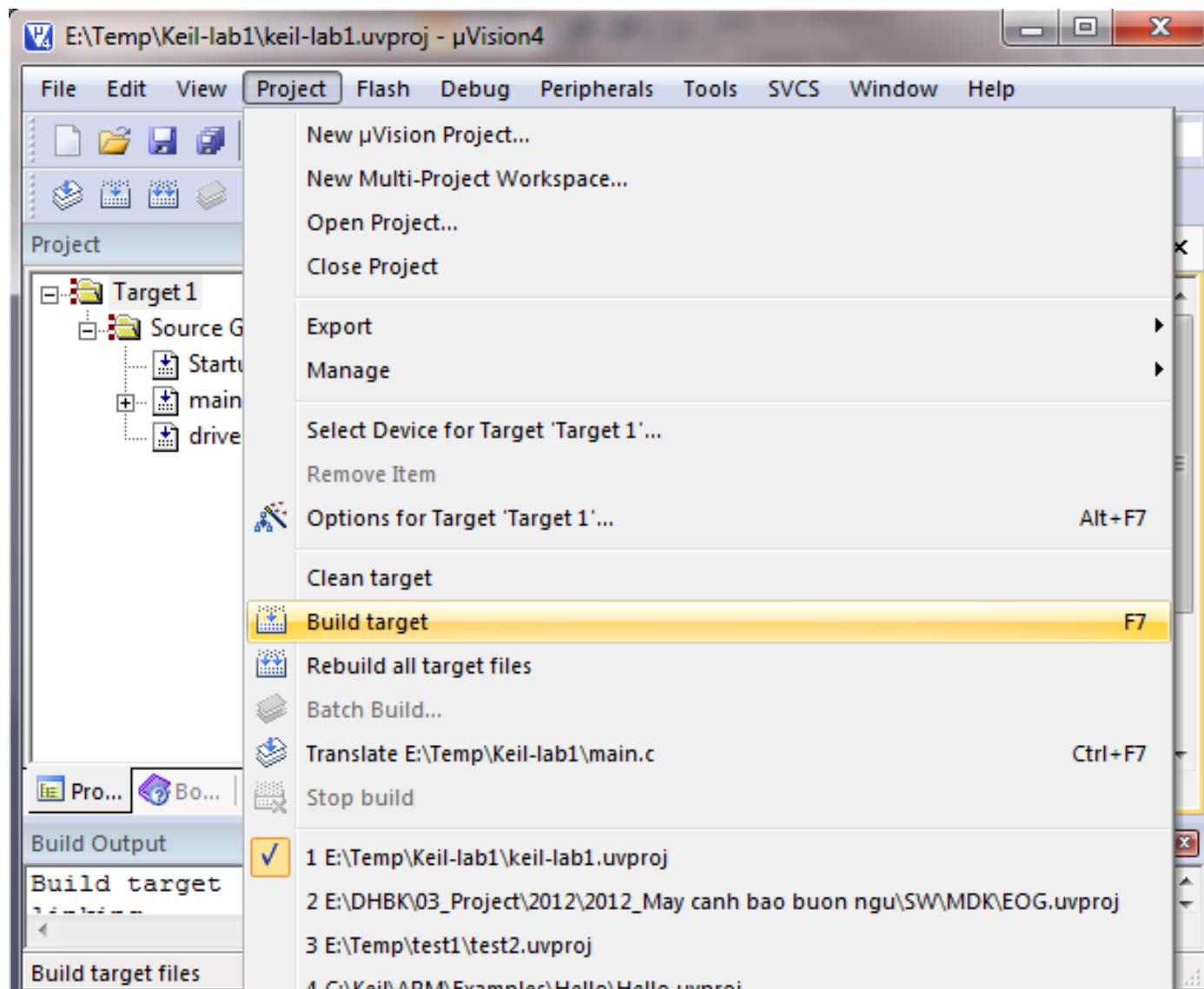


2. Keil Tools by ARM – Options

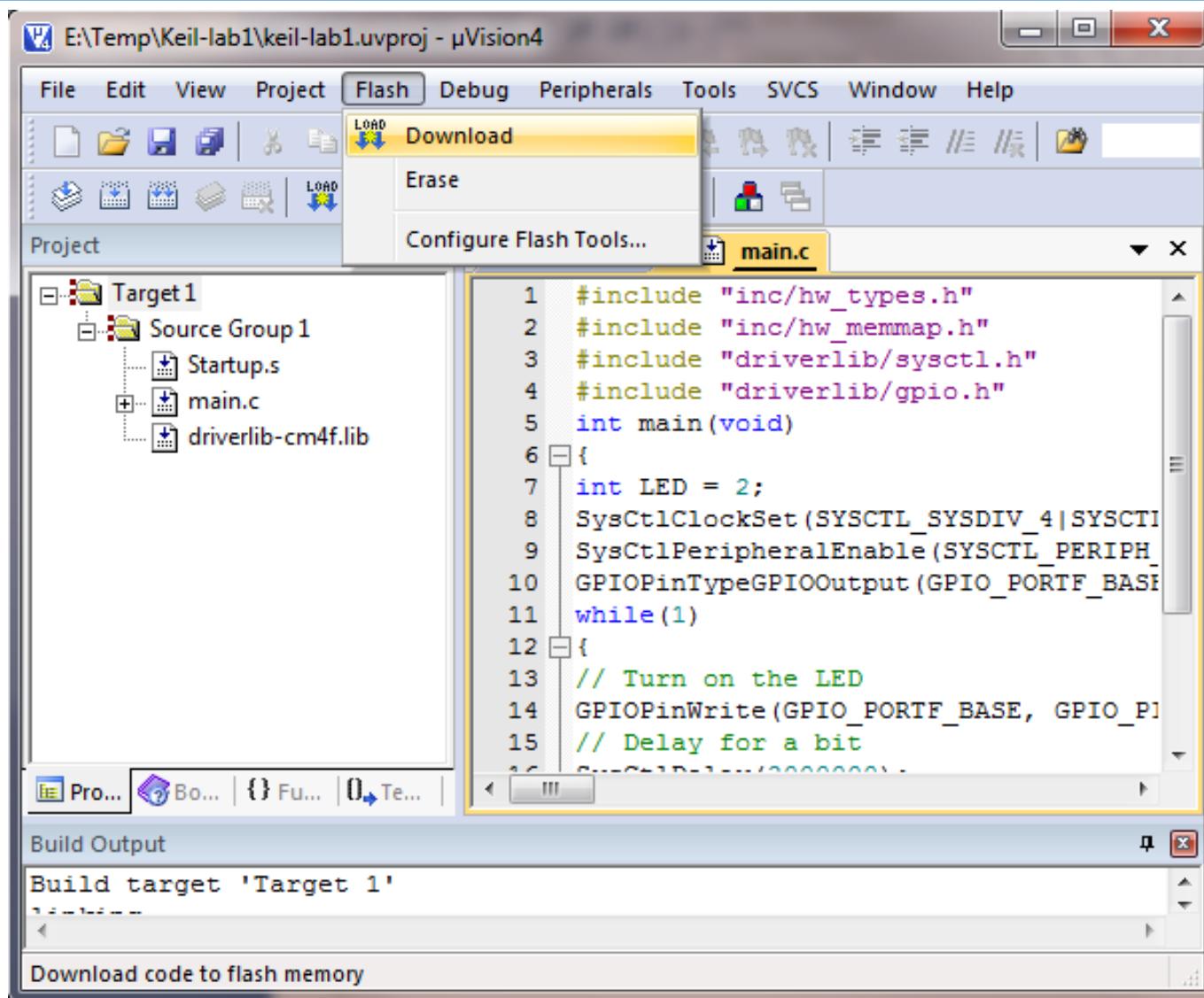
- Include Paths: C:\StellarisWare



2. Keil Tools by ARM – Build target

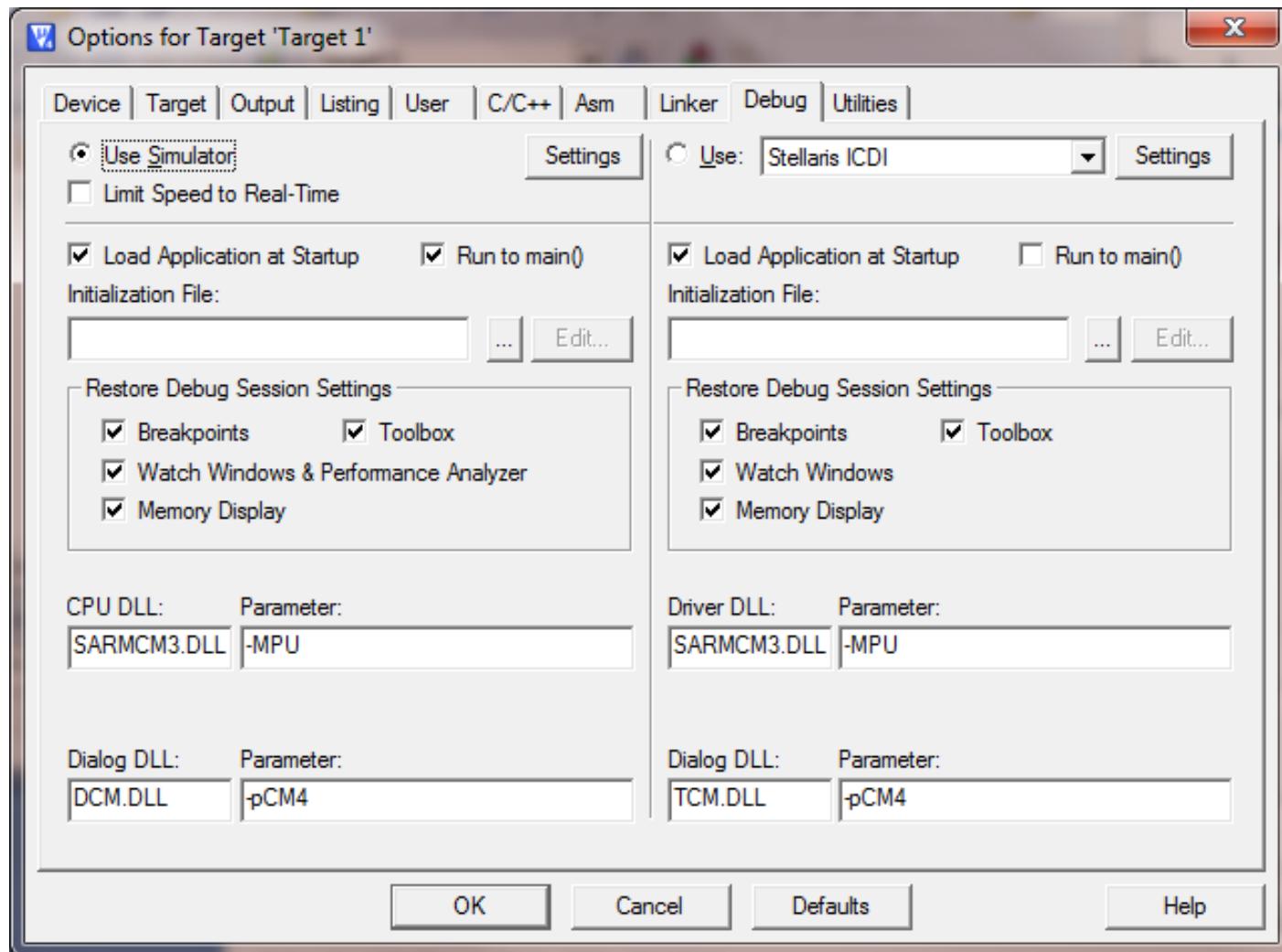


2. Keil Tools by ARM – Run on Kit

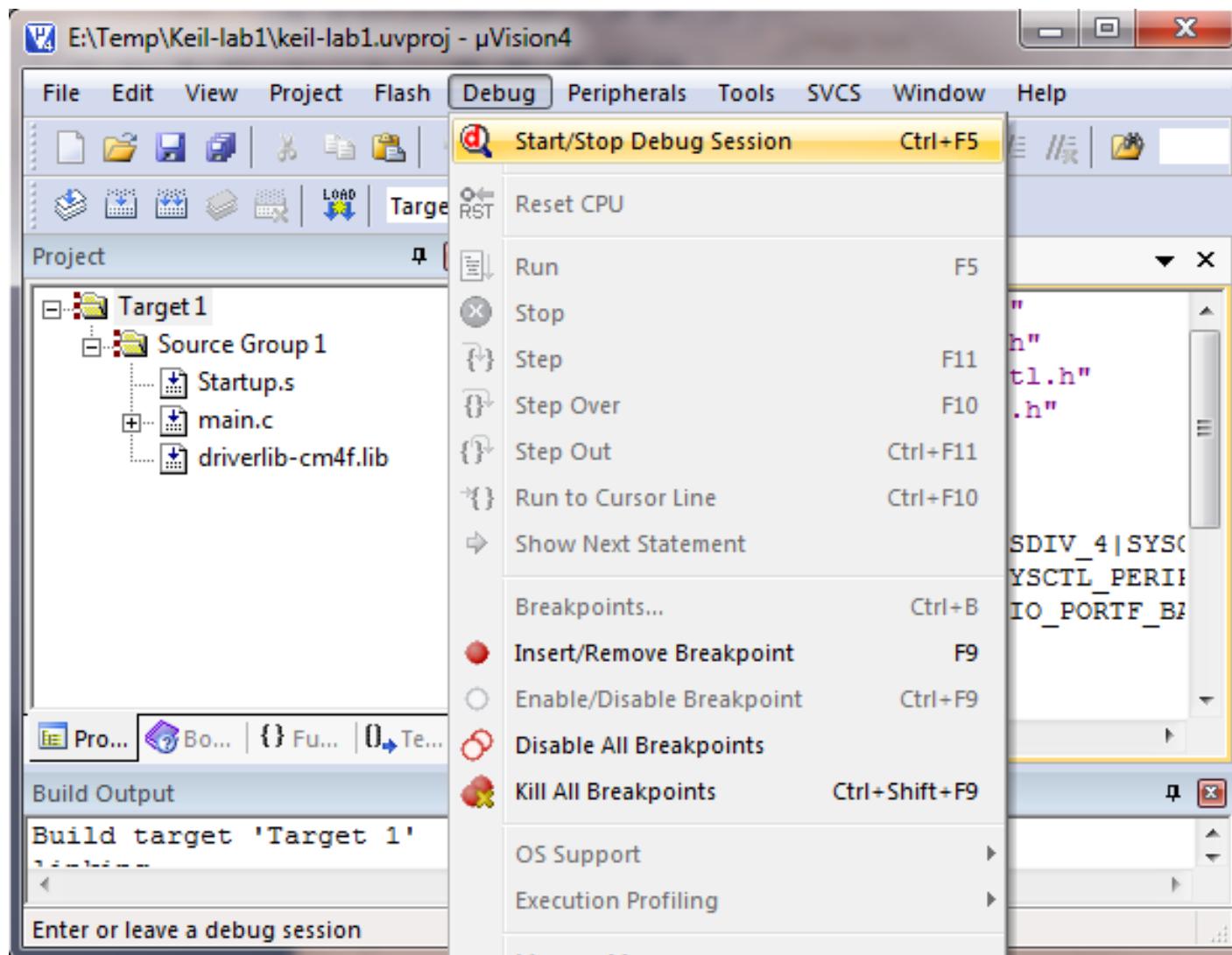


2. Keil Tools by ARM – Debug

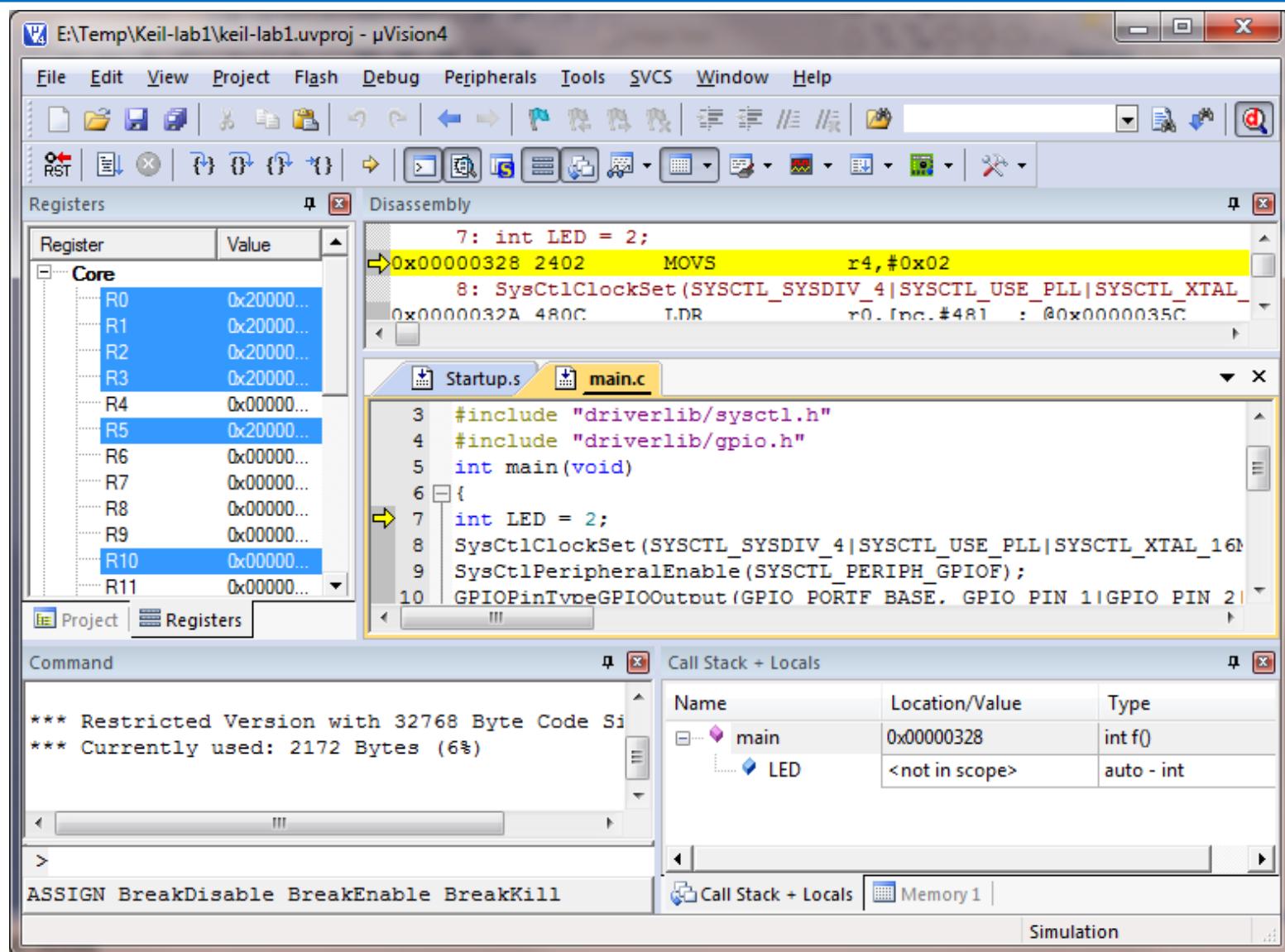
- Use Simulator or Use Stellaris ICDI



2. Keil Tools by ARM – Debug



2. Keil Tools by ARM – Simulation



Sample Codes

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/gpio.c"
#include "driverlib/sysctl.h"
unsigned int i;
int main()
{
    //Enable GPIO Port F
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
    // Set pins PF3 as output.
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,GPIO_PIN_3);
    while(1)
    {
        GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_3,GPIO_PIN_3); //SET PF3
        for(i=0;i<1000000;i++){}; //Delay
        GPIOPinWrite(GPIO_PORTF_BASE,GPIO_PIN_3,0); //Clear PF3
        for(i=0;i<1000000;i++){}; //Delay
    }
}
```



Embedded System Design

Chapter 5: Using Peripherals and Interrupts

5.1 Timers

5.2 Interrupts

5.3 ADC

5.4 UART

5.5 USB

1. Timer

- **Six** 16/32-bit and **Six** 32/64-bit general purpose timers
- **Twelve** 16/32-bit and **Twelve** 32/64-bit capture / compare / PWM pins
- Timer modes:
 - One-shot
 - Periodic
 - Input edge count or time capture with 16-bit prescaler
 - PWM generation (separated only)
 - Real-Time Clock (concatenated only)
- Count up or down
- PWM

1. Timer: Configuration

Timer configuration

- void **SysCtlPeripheralEnable**(unsigned long ulPeripheral)
 - ulPeripheral: SYSCTL_PERIPH_TIMER0, ..., SYSCTL_PERIPH_TIMER5,
SYSCTL_PERIPH_WTIMER0, ..., SYSCTL_PERIPH_WTIMER5
- void **TimerConfigure**(unsigned long ulBase, unsigned long ulConfig)
 - ulConfig: TIMER_CFG_ONE_SHOT, TIMER_CFG_PERIODIC,
TIMER_CFG_RTC, TIMER_CFG_SPLIT_PAIR, TIMER_CFG_32_BIT_PER, ...
- Example:
 - SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
 - TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);

1. Timer: Delay

- void **TimerLoadSet**(unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
 - ulBase is the base address of the timer module.
 - ulTimer specifies the timer(s) to adjust; must be one of TIMER_A, TIMER_B, or TIMER_BOTH. Only TIMER_A should be used when the timer is configured for full-width operation.
 - ulValue is the load value.
- Example: to toggle a GPIO at 10Hz and a 50% duty cycle, we need to delay 0.05s
 - ulPeriod = (SysCtlClockGet() / 10) / 2;
 - TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod -1);



Sample Codes

```
#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/interrupt.h"
#include "driverlib/gpio.h"
#include "driverlib/timer.h"

void Timer0IntHandler(void) {
    // Clear the timer interrupt
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    // Read the current state of the GPIO pin and write back the opposite state
    if(GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_2)) {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0);
    }
    else {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 4);
    }
}
```

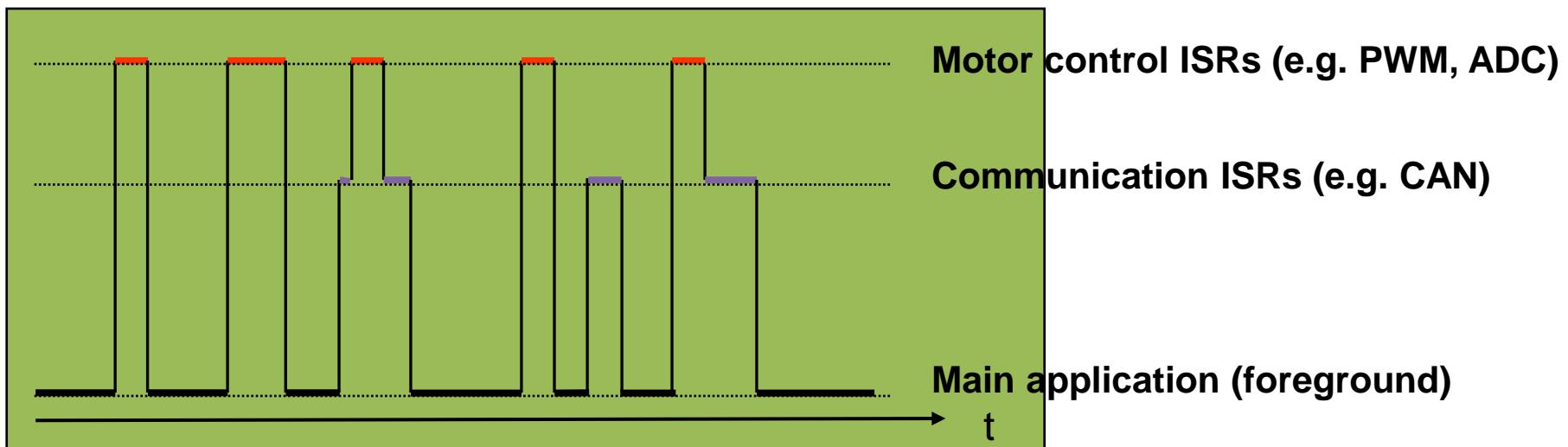
Sample Codes

```
int main(void) {  
    unsigned long ulPeriod;  
    SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_  
        OSC_MAIN);  
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);  
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,  
        GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);  
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMERO);  
    TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);  
    ulPeriod = (SysCtlClockGet() / 10) / 2;  
    TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod -1);  
    IntEnable(INT_TIMER0A);  
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);  
    IntMasterEnable();  
    TimerEnable(TIMER0_BASE, TIMER_A);  
    while(1) { }  
}
```

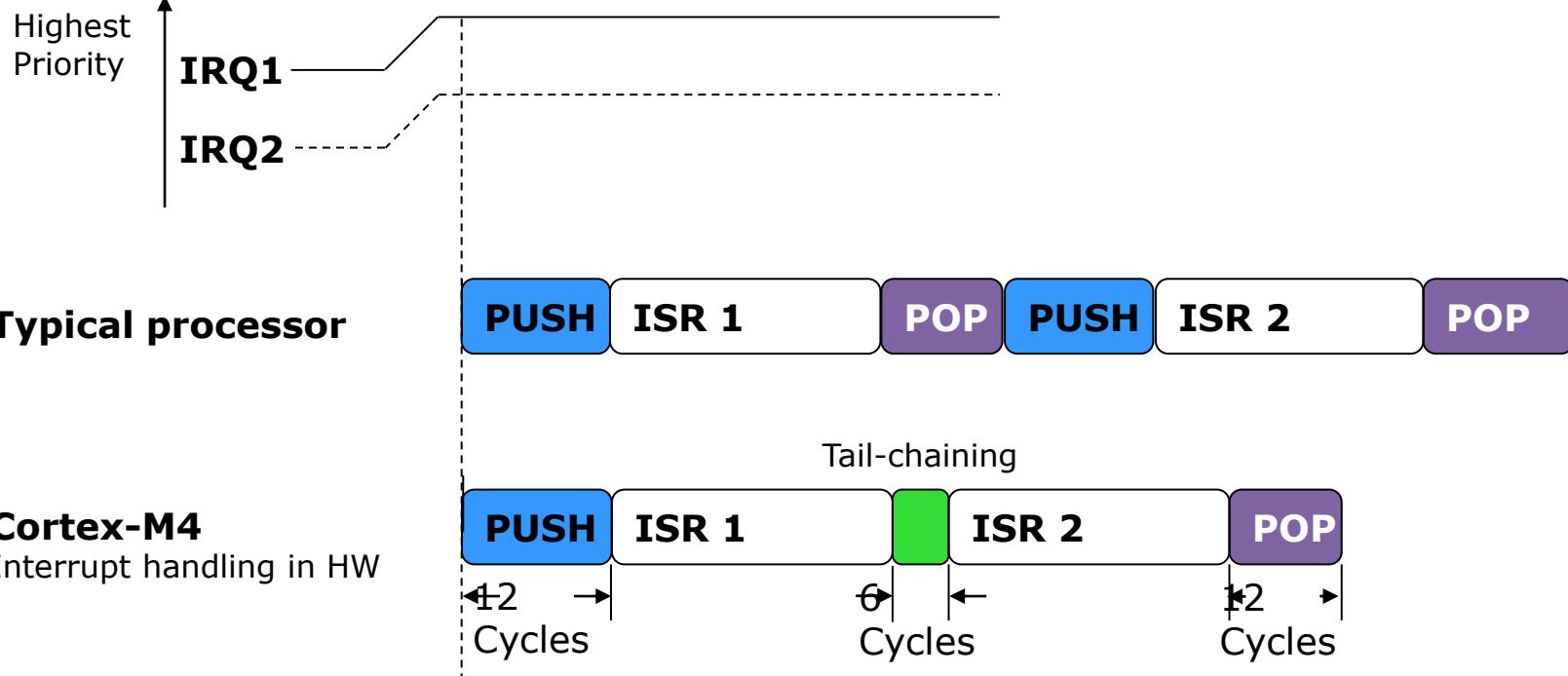
2. Interrupt

Nested Vectored Interrupt Controller (NVIC)

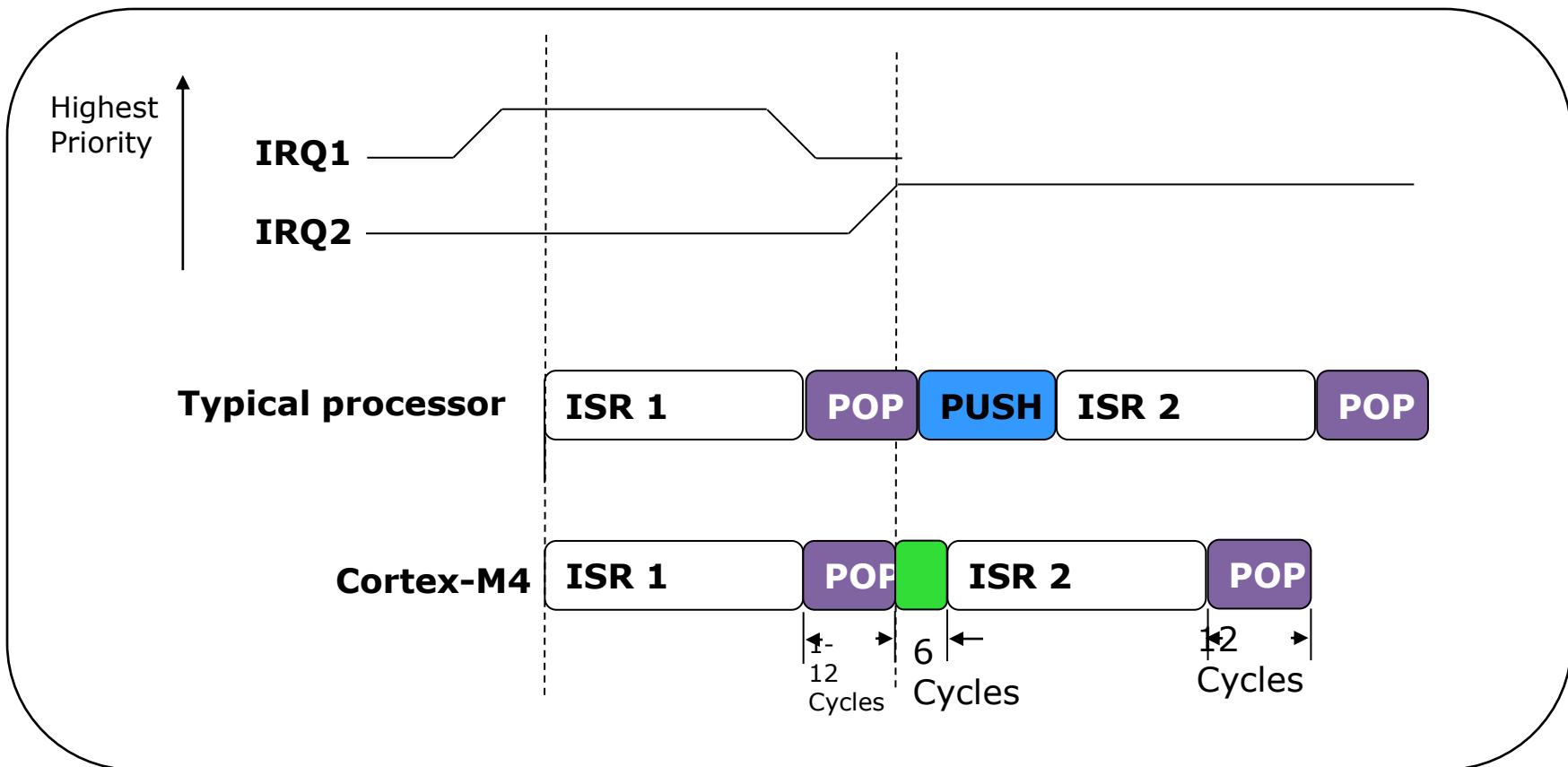
- Handles exceptions and interrupts
- 8 programmable priority levels, priority grouping
- 7 exceptions and 65 Interrupts
- Automatic state saving and restoring
- Automatic reading of the vector table entry
- Pre-emptive/Nested Interrupts
- Tail-chaining
- Deterministic: always 12 cycles or 6 with tail-chaining



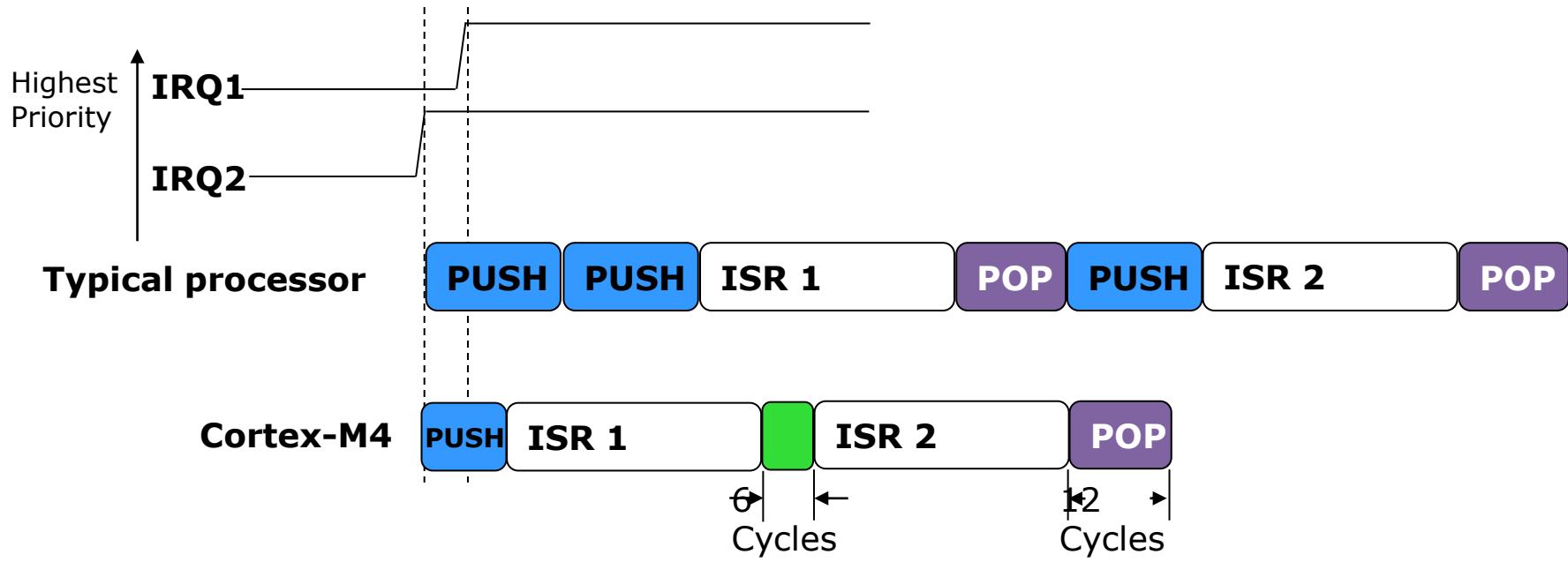
Interrupt Latency - Tail Chaining



Interrupt Latency – Pre-emption



Interrupt Latency – Late Arrival





Cortex-M4® Interrupt Handling

Interrupt handling is automatic. No instruction overhead.

Entry

Automatically pushes registers R0–R3, R12, LR, PSR, and PC onto the stack

In parallel, ISR is pre-fetched on the instruction bus. ISR ready to start executing as soon as stack PUSH complete

Exit

Processor state is automatically restored from the stack

In parallel, interrupted instruction is pre-fetched ready for execution upon completion of stack POP

Cortex-M4® Exception Types

Vector Number	Exception Type	Priority	Vector address	Descriptions
1	Reset	-3	0x04	Reset
2	NMI	-2	0x08	Non-Maskable Interrupt
3	Hard Fault	-1	0x0C	Error during exception processing
4	Memory Management Fault	Programmable	0x10	MPU violation
5	Bus Fault	Programmable	0x14	Bus error (Prefetch or data abort)
6	Usage Fault	Programmable	0x18	Exceptions due to program errors
7-10	Reserved	-	0x1C - 0x28	
11	SVCall	Programmable	0x2C	SVC instruction
12	Debug Monitor	Programmable	0x30	Exception for debug
13	Reserved	-	0x34	
14	PendSV	Programmable	0x38	
15	SysTick	Programmable	0x3C	System Tick Timer
16 and above	Interrupts	Programmable	0x40	External interrupts (Peripherals)

Cortex-M4® Vector Table

- After reset, vector table is located at address 0
- Each entry contains the address of the function to be executed
- The value in address 0x00 is used as starting address of the Main Stack Pointer (MSP)
- Vector table can be relocated by writing to the VTABLE register (must be aligned on a 1KB boundary)
- Open startup_ccs.c to see vector table coding

Exception number	IRQ number	Offset	Vector
154	138	0x0268	IRQ131
.	.	.	.
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12		0x002C	Reserved for Debug
11	-5	0x002C	SVCall
10		0x002C	Reserved
9		0x0018	Usage fault
8		0x0014	Bus fault
7		0x0010	Memory management fault
6	-10	0x000C	Hard fault
5	-11	0x0008	NMI
4	-12	0x0004	Reset
3	-13		
2	-14		
1		0x0000	Initial SP value

Interrupt Configuration

- Interrupt Enable
 - void IntDisable (unsigned long ullInterrupt)
 - void IntEnable(unsigned long ullInterrupt)
 - tBoolean IntMasterDisable (void)
 - tBoolean IntMasterEnable (void)
 - void TimerIntEnable (unsigned long ulBase, unsigned long ullIntFlags)
- Example
 - IntEnable(INT_TIMEROA);
 - TimerIntEnable(TIMERO_BASE, TIMER_TIMA_TIMEOUT);
 - IntMasterEnable();



Interrupt Handler

Interrupt handler sample: toggle a LED at GPIO_PIN2

```
void Timer0IntHandler(void)
{
    // Clear the timer interrupt
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    // Read the current state of the GPIO pin and
    // write back the opposite state
    if(GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_2))
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0);
    }
    else
    {
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 4);
    }
}
```

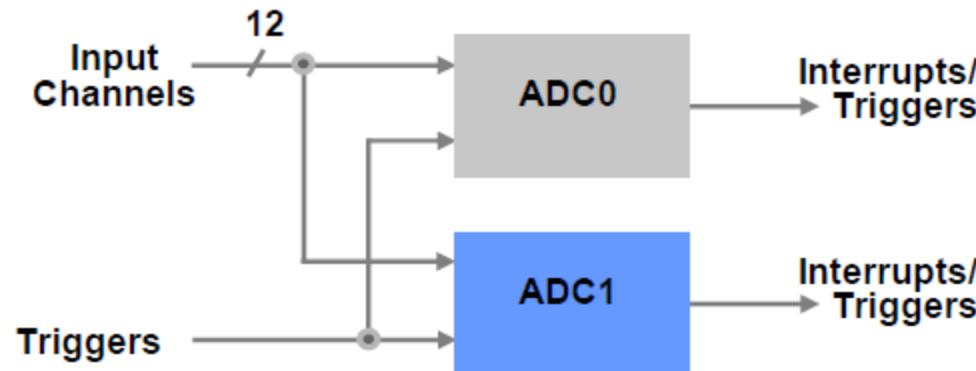


Class Assignment

- Write a program to toggle the LED on the LM4F120 kit with the frequency 20Hz using Timer1 and interrupt

3. Analog to Digital Converter (ADC)

- Stellaris LM4F MCUs feature two ADC modules (ADC0 and ADC1)
 - Each ADC module has 12-bit resolution
 - Each ADC module operates independently and can:
 - Execute different sample sequences
 - Sample any of the shared analog input channels
 - Generate interrupts & triggers



ADC Sample Sequencers

- Stellaris LM4F ADC's collect and sample data using programmable sequencers.
- Each sample sequence is a fully programmable series of consecutive (back-to-back) samples that allows the ADC module to collect data from multiple input sources without having to be re-configured.
- Each ADC module has **4 sample sequencers** that control sampling and data capture.
- All sample sequencers are identical except for the number of samples they can capture and the depth of their FIFO.
- To configure a sample sequencer, the following information is required:
 - Input source for each sample
 - Mode (single-ended, or differential) for each sample
 - Interrupt generation on sample completion for each sample
 - Indicator for the last sample in the sequence
- Each sample sequencer can transfer data independently through a dedicated µDMA channel.

Sequencer	Number of Samples	Depth of FIFO
SS 3	1	1
SS 2	4	4
SS 1	4	4
SS 0	8	8

3. ADC - Configuration

- Enable ADC
 - `SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);`
- Set speed
 - `void SysCtlADCSSpeedSet(unsigned long ulSpeed)`
 - `ulSpeed:`
 - `SYSCTL_ADCSPEED_1MSPS`
 - `SYSCTL_ADCSPEED_500KSPS`
 - `SYSCTL_ADCSPEED_250KSPS`
 - `SYSCTL_ADCSPEED_125KSPS`
- Example
 - `SysCtlADCSSpeedSet(SYSCTL_ADCSPEED_250KSPS);`

3. ADC - Configuration

- Configure ADC sequencer
 - void **ADCSequenceConfigure**(unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulTrigger, unsigned long ulPriority)
 - ulBase is the base address of the ADC module.
 - ulSequenceNum is the sample sequence number.
 - ulTrigger is the trigger source that initiates the sample sequence; must be one of the ADC_TRIGGER_ values.
 - ulPriority is the relative priority of the sample sequence with respect to the other sample sequences.
- Example
 - ADCSequenceDisable(ADC0_BASE, 1); //disable before configuring
 - ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);

3. ADC - Configuration

- Configure a step of the sample sequencer
 - void **ADCSequenceStepConfigure**(unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulStep, unsigned long ulConfig)
 - ulBase is the base address of the ADC module.
 - ulSequenceNum is the sample sequence number.
 - ulStep is the step to be configured.
 - ulConfig is the configuration of this step; must be a logical OR of ADC_CTL_TS, ADC_CTL_IE, ADC_CTL_END, ADC_CTL_D, one of the input channel selects (ADC_CTL_CH0 through ADC_CTL_CH23), and one of the digital comparator selects (ADC_CTL_CMP0 through ADC_CTL_CMP7).
- Example: steps 0-2 sample the temperature sensor, step 3 configure the interrupt and end the conversion
 - ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_TS); *//CTL_TS: Temperature Sensor*
 - ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_TS); *//CTL_IE: interrupt enable*
 - ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_TS); *//CTL_END: end conversion*
 - ADCSequenceStepConfigure(ADC0_BASE, 1, 3, ADC_CTL_TS | ADC_CTL_IE | ADC_CTL_END);



3. ADC: Sample code

```
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/debug.h"
#include "driverlib/sysctl.h"
#include "driverlib/adc.h"

#ifndef DEBUG
void __error__(char *pcFilename, unsigned long ulLine)
{
}
#endif

int main(void)
{
    unsigned long ulADC0Value[4];
    volatile unsigned long ulTempAvg;
    volatile unsigned long ulTempValueC;
    volatile unsigned long ulTempValueF;
```

3. ADC: Sample code

```
SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTA  
L_16MHZ);  
SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0);  
SysCtlADCSSpeedSet(SYSCTL_ADCSPEED_250KSPS);  
ADCSequenceDisable(ADC0_BASE, 1);  
ADCSequenceConfigure(ADC0_BASE, 1, ADC_TRIGGER_PROCESSOR, 0);  
ADCSequenceStepConfigure(ADC0_BASE, 1, 0, ADC_CTL_TS);  
ADCSequenceStepConfigure(ADC0_BASE, 1, 1, ADC_CTL_TS);  
ADCSequenceStepConfigure(ADC0_BASE, 1, 2, ADC_CTL_TS);  
ADCSequenceStepConfigure(ADC0_BASE, 1, 3, ADC_CTL_TS | ADC_CTL_IE  
|ADC_CTL_END);  
ADCSequenceEnable(ADC0_BASE, 1);
```

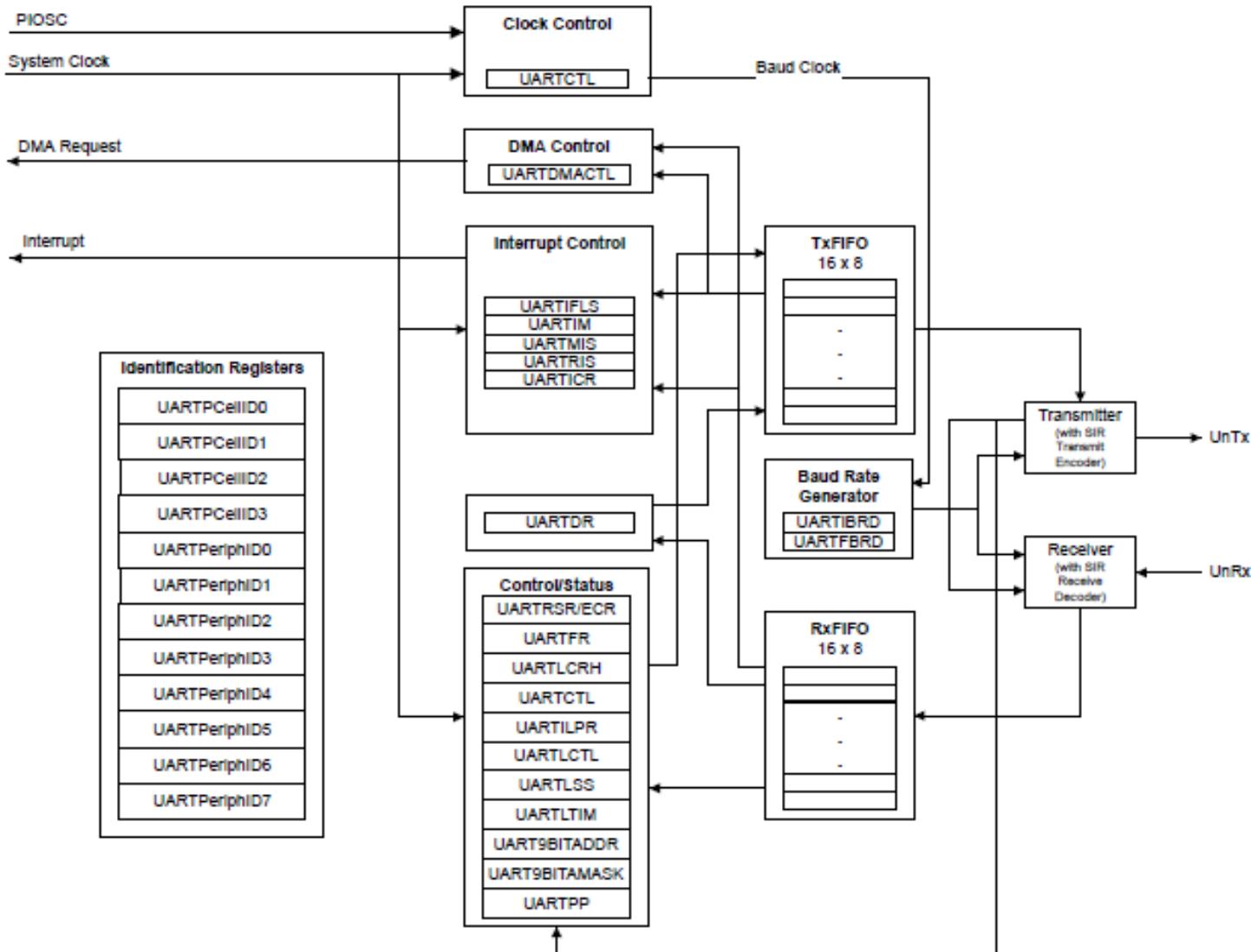
3. ADC: Sample code

```
while(1)
{
    ADCIntClear(ADC0_BASE, 1);
    ADCProcessorTrigger(ADC0_BASE, 1);
    while(!ADCIntStatus(ADC0_BASE, 1, false))
    {
        }
    ADCSequenceDataGet(ADC0_BASE, 1, ulADC0Value);
    ulTempAvg = (ulADC0Value[0] + ulADC0Value[1] + ulADC0Value[2] + ulADC0Value[3]
+ 2)/4;
    ulTempValueC = (1475 - ((2475 * ulTempAvg)) / 4096)/10;
    ulTempValueF = ((ulTempValueC * 9) + 160) / 5;
}
}
```

4. UART: Features

- Separate 16x8 bit transmit and receive FIFOs
- Programmable baud rate generator
- Auto generation and stripping of start, stop, and parity bits
- Line break generation and detection
- Programmable serial interface
 - 5, 6, 7, or 8 data bits
 - even, odd, stick, or no parity bits
 - 1 or 2 stop bits
 - baud rate generation, from DC to processor clock/16
- Modem control/flow control
- IrDA and EIA-495 9-bit protocols
- μ DMA support

4. UART: Block Diagram





4. UART: Basic Operation

- Initialize the UART

- Enable the UART peripheral, e.g.

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);  
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
```

- Set the Rx/Tx pins as UART pins

```
GPIOPinConfigure(GPIO_PA0_U0RX);  
GPIOPinConfigure(GPIO_PA1_U0TX);  
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
```

- Configure the UART baud rate, data configuration

```
ROM_UARTConfigSetExpClk(UART0_BASE, ROM_SysCtlClockGet(), 115200,  
                         UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |  
                         UART_CONFIG_PAR_NONE));
```

- Configure other UART features (e.g. interrupts, FIFO)

- Send/receive a character

- Single register used for transmit/receive
 - Blocking/non-blocking functions in driverlib:

```
UARTCharPut(UART0_BASE, 'a');  
newchar = UARTCharGet(UART0_BASE);  
UARTCharPutNonBlocking(UART0_BASE, 'a');  
newchar = UARTCharGetNonBlocking(UART0_BASE);
```



UART Interrupts

Single interrupt per module, cleared automatically

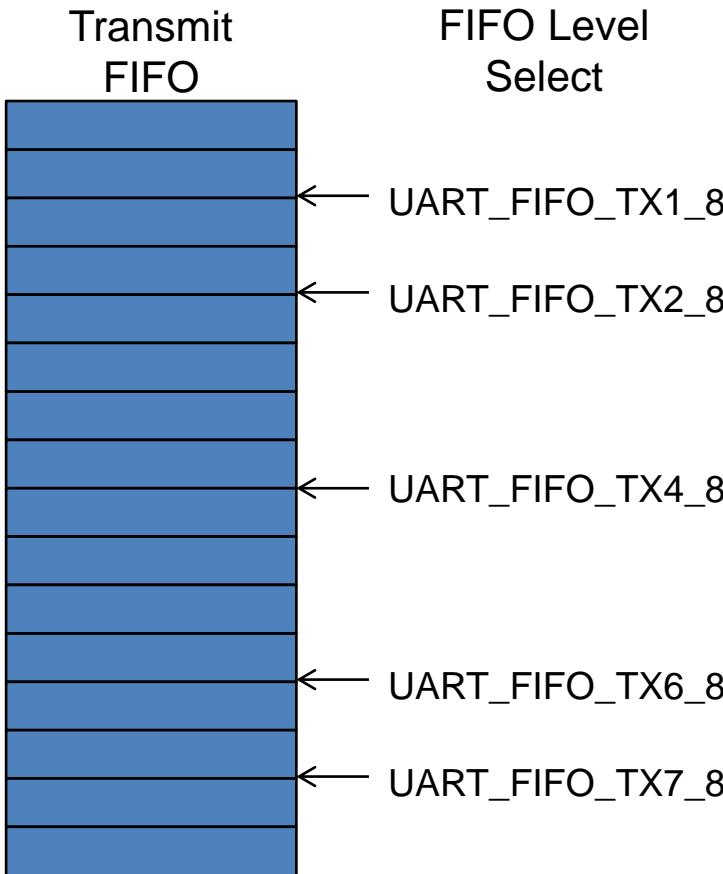
Interrupt conditions:

- Overrun error
- Break error
- Parity error
- Framing error
- Receive timeout – when FIFO is not empty and no further data is received over a 32-bit period
- Transmit – generated when no data present (if FIFO enabled, see next slide)
- Receive – generated when character is received (if FIFO enabled, see next slide)

Interrupts on these conditions can be enabled individually

Your handler code must check to determine the source of the UART interrupt and clear the flag(s)

Using the UART FIFOs



- Both FIFOs are accessed via the UART Data register (UARTDR)
- After reset, the FIFOs are enabled*, you can disable by resetting the FEN bit in UARTLCRH, e.g.
`UARTFIFODisable (UART0_BASE) ;`
- Trigger points for FIFO interrupts can be set at 1/8, 1/4, 1/2, 3/4, 7/8 full, e.g.
`UARTFIFOLevelSet (UART0_BASE,
UART_FIFO_TX4_8,
UART_FIFO_RX4_8) ;`

* Note: the datasheet says FIFOs are disabled at reset



UART “stdio” Functions

- StellarisWare “utils” folder contains functions for C stdio console functions:

c:\StellarisWare\utils\uartstdio.h

c:\StellarisWare\utils\uartstdio.c

- Usage example:

```
UARTStdioInit(0); //use UART0, 115200
```

```
UARTprintf("Enter text: ");
```

- See `uartstdio.h` for other functions

- Notes:

- Use the provided interrupt handler `UARTStdioIntHandler()` code in `uartstdio.c`
- Buffering is provided if you define `UART_BUFFERED` symbol
 - Receive buffer is 128 bytes
 - Transmit buffer is 1024 bytes



Other UART Features

- Modem control/flow control
- IrDA serial IR (SIR) encoder/decoder
 - External infrared transceiver required
 - Supports half-duplex serial SIR interface
 - Minimum of 10-ms delay required between transmit/receive, provided by software
- ISA 7816 smartcard support
 - UnTX signal used as a bit clock
 - UnRx signal is half-duplex communication line
 - GPIO pin used for smartcard reset, other signals provided by your system design
- LIN (Local Interconnect Network) support: master or slave
- μDMA support
 - Single or burst transfers support
 - UART interrupt handler handles DMA completion interrupt
- EIA-495 9-bit operation
 - Multi-drop configuration: one master, multiple slaves
 - Provides “address” bit (in place of parity bit)
 - Slaves only respond to their address



Example Code

```
#include "inc/hw_types.h"
#include "driverlib/gpio.h"
#include "driverlib/pin_map.h"
#include "driverlib/sysctl.h"
#include "driverlib/uart.h"

int main(void) {
    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
    SYSCTL_XTAL_16MHZ);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    GPIOPinConfigure(GPIO_PA0_U0RX);
    GPIOPinConfigure(GPIO_PA1_U0TX);
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
    UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200,
    (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_ONE |
    UART_CONFIG_PAR_NONE));
```



Example code

```
UARTCharPut(UART0_BASE, 'E');
UARTCharPut(UART0_BASE, 'n');
UARTCharPut(UART0_BASE, 't');
UARTCharPut(UART0_BASE, 'e');
UARTCharPut(UART0_BASE, 'r');
UARTCharPut(UART0_BASE, ' ');
UARTCharPut(UART0_BASE, 'T');
UARTCharPut(UART0_BASE, 'e');
UARTCharPut(UART0_BASE, 'x');
UARTCharPut(UART0_BASE, 't');
UARTCharPut(UART0_BASE, ':');
UARTCharPut(UART0_BASE, ' ');
while (1)
{
    if (UARTCharsAvail(UART0_BASE)) UARTCharPut(UART0_BASE, UARTCharGet(UART0_BASE));
}
```

5. USB Basics

Multiple connector sizes

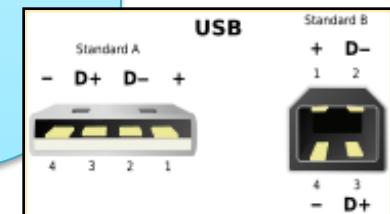
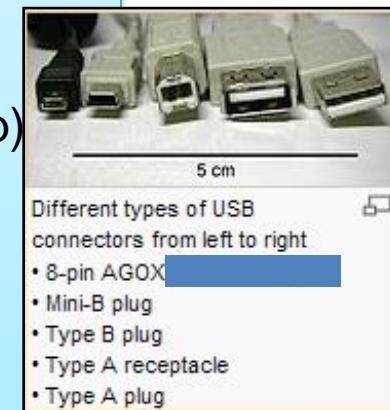
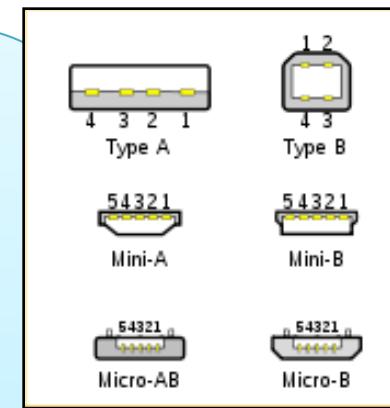
4 pins – power, ground and 2 data lines

(5th pin ID for USB 2.0 connectors)

Configuration connects power 1st, then data

Standards:

- ◆ USB 1.1
 - Defines Host (master) and Device (slave)
 - Speeds to 12Mbits/sec
 - Devices can consume 500mA (100mA for startup)
- ◆ USB 2.0
 - Speeds to 480Mbits/sec
 - OTG addendum
- ◆ USB 3.0
 - Speeds to 4.8Gbits/sec
 - New connector(s)
 - Separate transmit/receive data lines



USB Basics

USB Device ... most USB products are slaves

USB Host ... usually a PC, but can be embedded

USB OTG ... On-The-Go

- Dynamic switching between host and device roles
- Two connected OTG ports undergo host negotiation

Host polls each Device at power up. Information from Device includes:

- Device Descriptor (Manufacturer & Product ID so Host can find driver)
- Configuration Descriptor (Power consumption and Interface descriptors)
- Endpoint Descriptors (Transfer type, speed, etc)
- Process is called *Enumeration* ... allows Plug-and-Play



LM4F120H5QR USB

- USB 2.0 Device mode full speed (12 Mbps) and low speed (1.5 Mbps) operation
- Integrated PHY
- Transfer types: Control, Interrupt, Bulk and Isochronous
- Device Firmware Update (DFU) device in ROM



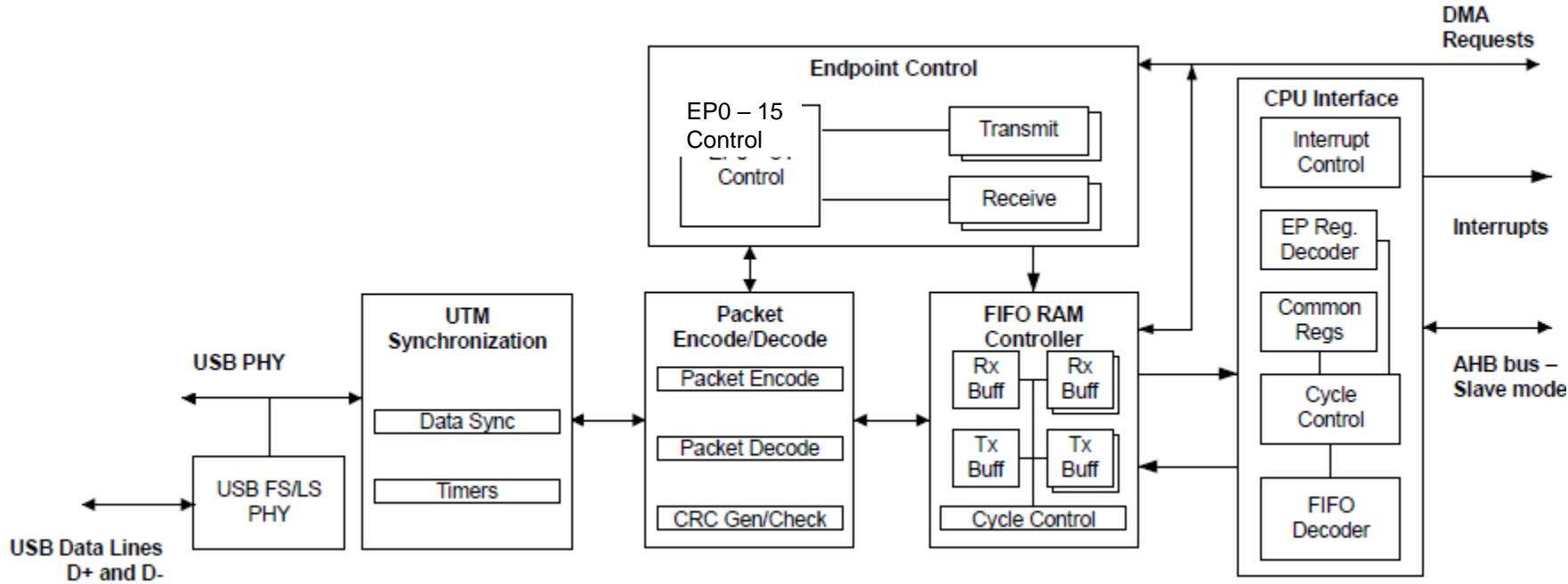
Stellaris collaterals

- Texas Instruments is a member of the USB Implementers Forum.
- Stellaris is approved to use the USB logo
- Vendor/Product ID sharing

<http://www.ti.com/lit/pdf/spml001>



USB Peripheral Block Diagram



Integrated USB Controller and PHY with up to 16 Endpoints

- 1 dedicated control IN endpoint and 1 dedicated control OUT endpoint
- Up to 7 configurable IN endpoints and 7 configurable OUT endpoints
- 4 KB dedicated endpoint memory (not part of device SRAM)
- Separate DMA channels (up to three IN Endpoints and three OUT Endpoints)
- 1 endpoint may be defined for double-buffered 1023-bytes isochronous packet size

StellarisWare USBLib

- License-free & royalty-free drivers, stack and example applications for Stellaris MCUs
- USBLib supports Host/Device and OTG, but the LM4F120H5QR USB port is Device only
- Builds on DriverLib API
 - Adds framework for generic Host and Device functionality
 - Includes implementations of common USB classes
- Layered structure
- Drivers and .inf files included where appropriate
- Stellaris MCUs have passed USB Device and Embedded Host compliance testing



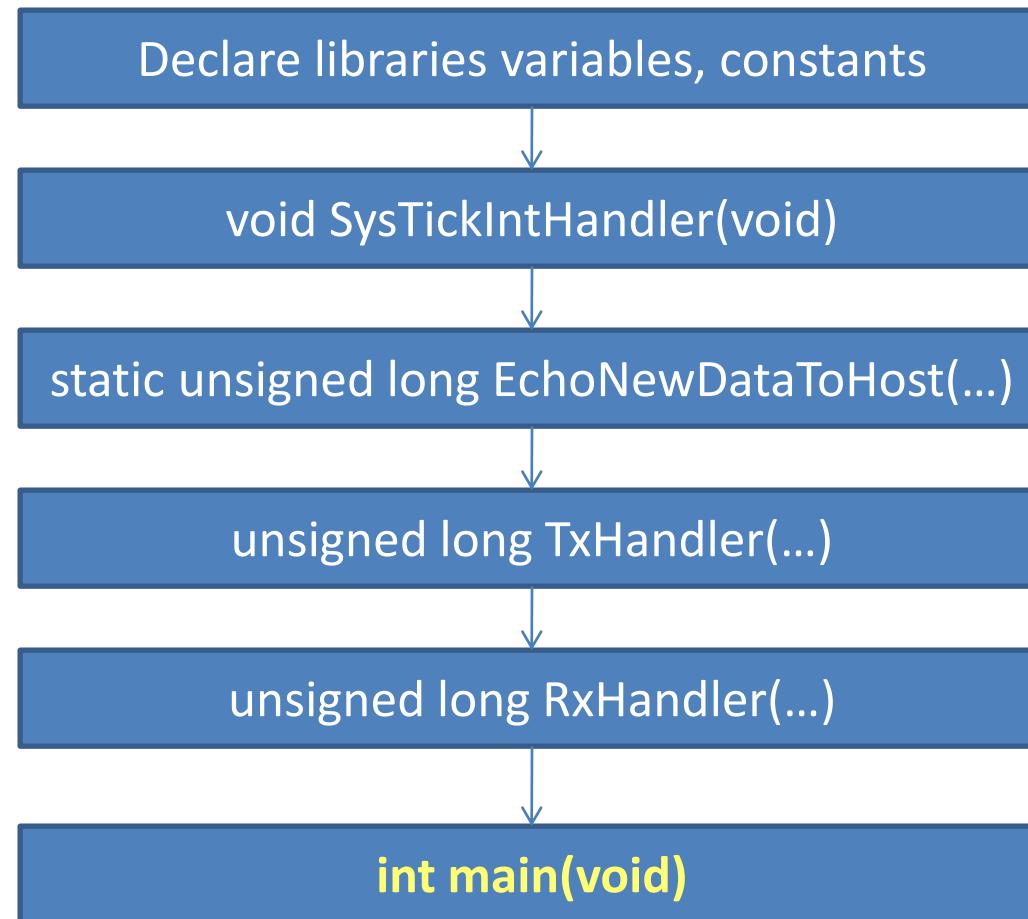
- Device Examples
 - HID Keyboard
 - HID Mouse
 - CDC Serial
 - Mass Storage
 - Generic Bulk
 - Audio
 - Device Firmware Upgrade
 - Oscilloscope
- Windows INF for supported devices
 - Points to base Windows drivers
 - Sets config string
 - Sets PID/VID
 - Precompiled DLL saves development time
- Device framework integrated into USBLib

USB: Example

- Lab 7:
 - c:\StellarisWare\boards\ek-lm4f120xl\usb_dev_bulk\
- Content:
 - This example provides a **generic USB device** offering simple bulk data transfer to and from the host
 - Data received from the host is assumed to be ASCII text and it is **echoed back** with the case of all alphabetic characters swapped.

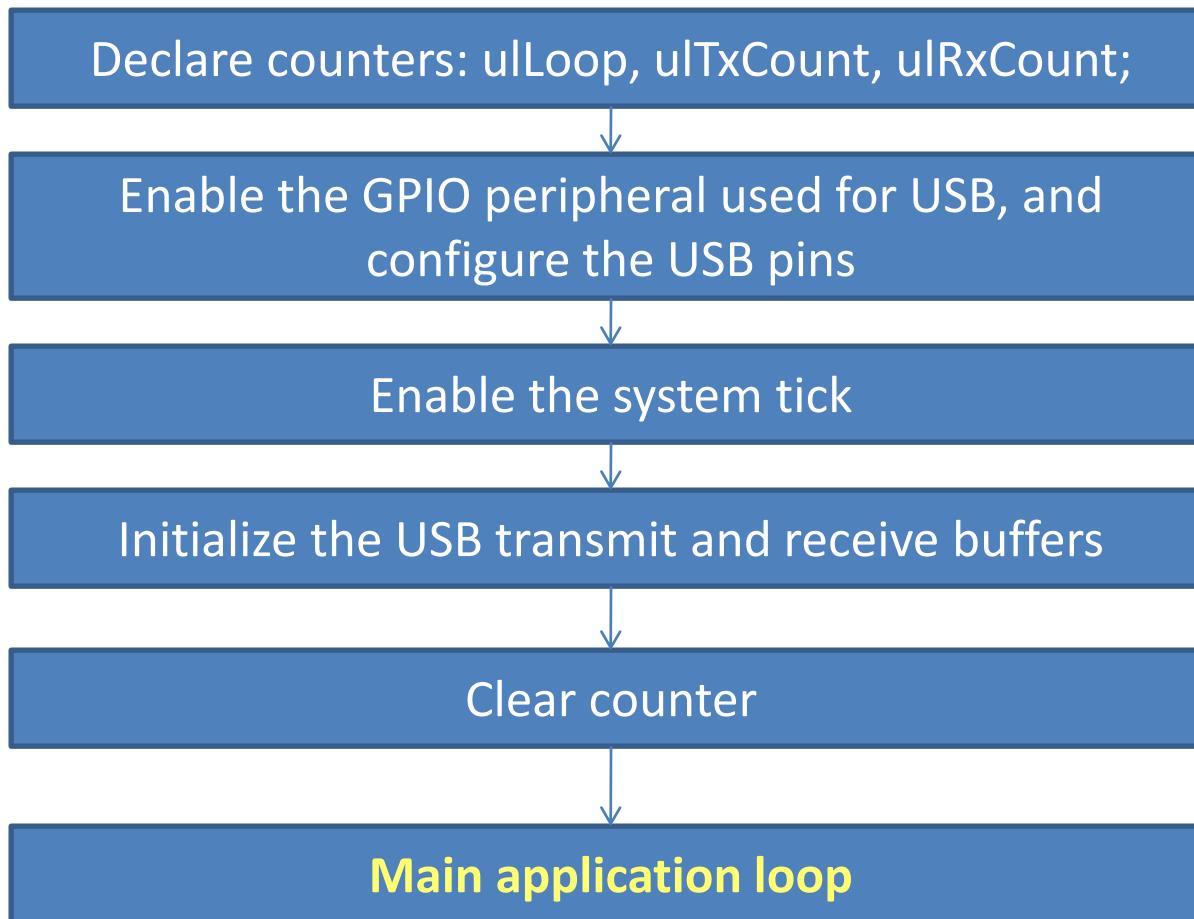
USB: Example

- Program flow



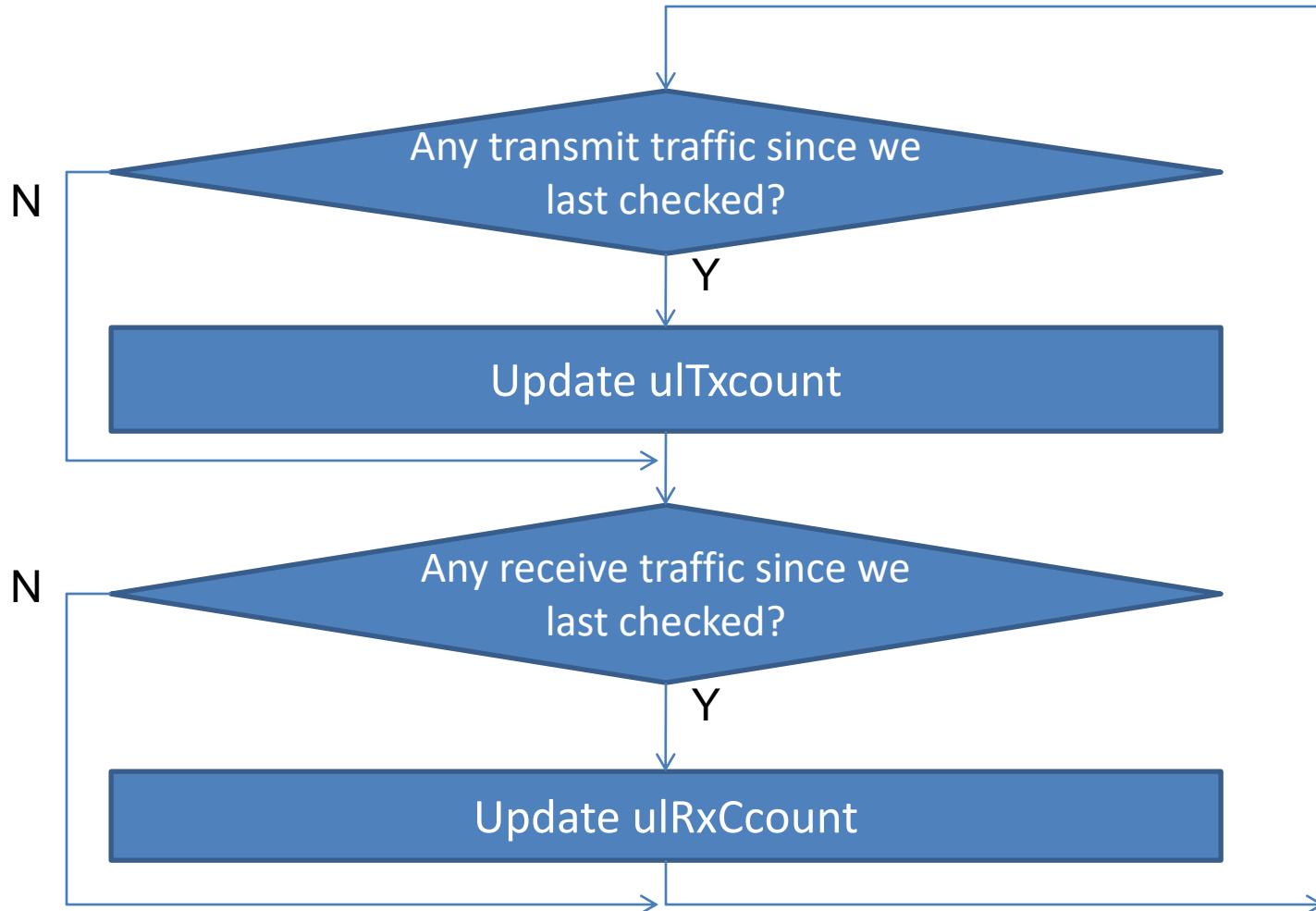
USB: Example

- Main flow



USB: Example

- Main application loop



USB: Example – C code

```
int main(void)
{ volatile unsigned long ulLoop;
  unsigned long ulTxCount;
  unsigned long ulRxCount;
  // Enable lazy stacking for interrupt handlers.
  ROM_FPULazyStackingEnable();
  // Set the clocking to run from the PLL at 50MHz
  ROM_SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                      SYSCTL_XTAL_16MHZ);
  // Enable the GPIO peripheral used for USB, and configure the USB pins.
  ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
  ROM_GPIOPinTypeUSBAnalog(GPIO_PORTD_BASE, GPIO_PIN_4 | GPIO_PIN_5);
  // Enable the system tick.
  ROM_SysTickPeriodSet(ROM_SysCtlClockGet() / SYSTICKS_PER_SECOND);
  ROM_SysTickIntEnable();
  ROM_SysTickEnable();
```



USB: Example – C code

```
// Initialize the transmit and receive buffers.  
USBBufferInit((tUSBBuffer *)&g_sTxBuffer);  
USBBufferInit((tUSBBuffer *)&g_sRxBuffer);  
  
// Set the USB stack mode to Device mode with VBUS monitoring.  
USBStackModeSet(0, USB_MODE_FORCE_DEVICE, 0);  
// Pass our device information to the USB library and place the device  
// on the bus.  
USDBulkInit(0, (tUSDBulkDevice *)&g_sBulkDevice);  
  
// Clear our local byte counters.  
ulRxCount = 0;  
ulTxCount = 0;
```

USB: Example – C Code

```
// Main application loop.  
while(1)  
{ // See if any data has been transferred.  
    if((ulTxCount != g_ulTxCount) || (ulRxCount != g_ulRxCount))  
    {  
        // Has there been any transmit traffic since we last checked?  
        if(ulTxCount != g_ulTxCount)  
        { // Turn on the Green LED.  
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, GPIO_PIN_3);  
            // Delay for a bit.  
            for(ulLoop = 0; ulLoop < 150000; ulLoop++) {}  
            // Turn off the Green LED.  
            GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, 0);  
  
            ulTxCount = g_ulTxCount;  
        }  
    }  
}
```



USB: Example – C Code

```
// Has there been any receive traffic since we last checked?  
if(ulRxCount != g_ulRxCount)  
{ // Turn on the Blue LED.  
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);  
    // Delay for a bit.  
    for(ulLoop = 0; ulLoop < 150000; ulLoop++) {}  
    // Turn off the Blue LED.  
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);  
    // Take a snapshot of the latest receive count.  
    ulRxCount = g_ulRxCount;  
}  
}  
}
```

USB: Example

```
// Receive new data and echo it back to the host.  
  
static unsigned long  
EchoNewDataToHost(tUSBDBulkDevice *psDevice, unsigned char *pcData,  
                      unsigned long ulNumBytes)  
{  
    unsigned long ulLoop, ulSpace, ulCount;  
    unsigned long ulReadIndex;  
    unsigned long ulWriteIndex;  
    tUSBRingBufObject sTxRing;  
    // Get the current buffer information  
USBBufferInfoGet(&g_sTxBuffer, &sTxRing);  
    // How much space is there in the transmit buffer?  
    ulSpace = USBBufferSpaceAvailable(&g_sTxBuffer);  
    // How many characters can we process this time round?  
    ulLoop = (ulSpace < ulNumBytes) ? ulSpace : ulNumBytes;  
    ulCount = ulLoop;
```



USB: Example

```
// Update our receive counter.  
g_ulRxCount += ulNumBytes;  
// Dump a debug message.  
DEBUG_PRINT("Received %d bytes\n", ulNumBytes);  
// Set up to process the characters by directly accessing the USB buffers.  
ulReadIndex = (unsigned long)(pcData - g_pucUSBRxBuffer);  
ulWriteIndex = sTxRing.ulWriteIndex;  
while(ulLoop)  
{  
    // Copy from the receive buffer to the transmit buffer converting  
    // character case on the way.  
    if((g_pucUSBRxBuffer[ulReadIndex] >= 'a') &&  
        (g_pucUSBRxBuffer[ulReadIndex] <= 'z'))  
    { // Convert to upper case and write to the transmit buffer.  
        g_pucUSBTxBuffer[ulWriteIndex] = (g_pucUSBRxBuffer[ulReadIndex] - 'a') + 'A';  
    }  
}
```

USB: Example

```
else
    { if((g_pucUSBRxBuffer[ulReadIndex] >= 'A') && (g_pucUSBRxBuffer[ulReadIndex]
        <= 'Z'))
        { // Convert to lower case and write to the transmit buffer.
            g_pucUSBTxBuffer[ulWriteIndex] = (g_pucUSBRxBuffer[ulReadIndex] - 'Z') + 'z';
        }
        else { // Copy the received character to the transmit buffer.
            g_pucUSBTxBuffer[ulWriteIndex] = g_pucUSBRxBuffer[ulReadIndex];
        }
    }
    // Move to the next character taking care to adjust the pointer for the buffer wrap
    ulWriteIndex++;
    ulWriteIndex = (ulWriteIndex == BULK_BUFFER_SIZE) ? 0 : ulWriteIndex;
    ulReadIndex++;
    ulReadIndex = (ulReadIndex == BULK_BUFFER_SIZE) ? 0 : ulReadIndex;
    ulLoop--;
}
```



USB: Example

```
USBBufferDataWritten(&g_sTxBuffer, ulCount);
DEBUG_PRINT("Wrote %d bytes\n", ulCount);

//
// We processed as much data as we can directly from the receive buffer so
// we need to return the number of bytes to allow the lower layer to
// update its read pointer appropriately.
//
return(ulCount);
}
```



Advanced Embedded System Design

Chapter 5: Embedded System Design Analysis and Optimization

1. Design Analysis
2. Design Optimization
3. Design Verification



1. Design Analysis

- 1) Quality function deployment
- 2) Cost analysis

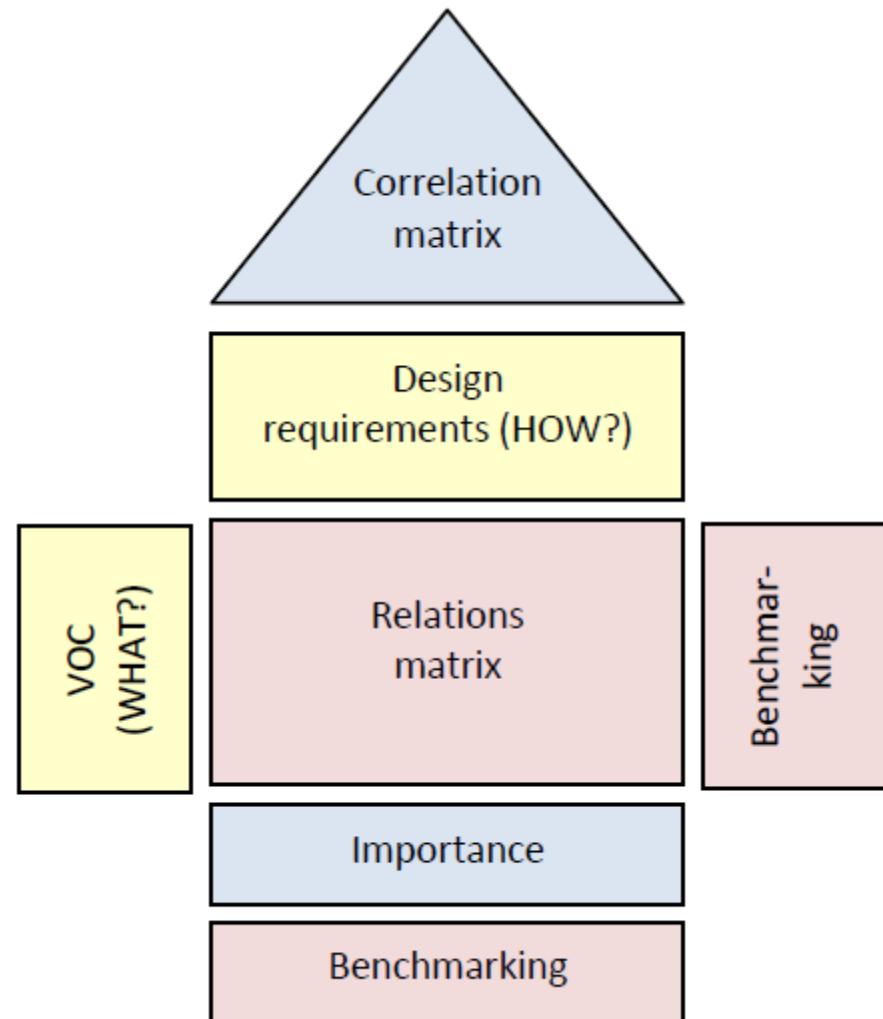


1.1. Quality function deployment

- **House of Quality** is the basic design tool of quality function deployment (QFD):
 - proposed by Dr. Yoji Akao in 1996
 - to transform qualitative user demands into quantitative parameters
 - to deploy the functions forming quality
 - to deploy methods for achieving the design quality into subsystems and component parts,
 - to specific elements of the manufacturing process.

1.1. Quality function deployment

- House of Quality: is a techniques based on QFD
 - Appeared in 1972
 - The house can be divided in “rooms”

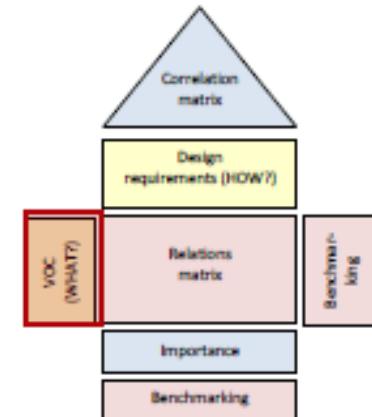


1.1. Quality function deployment

- House of Quality:

– Table 1: What?

- What is desired in order to reach the new service's development?

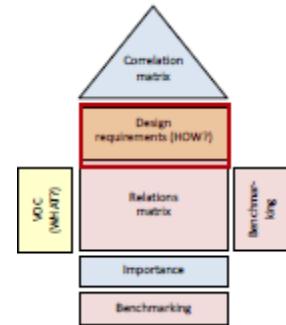


Example: Automatic light control system

No.	Customer's requirements
1	Long life cycle
2	Easy to charge energy
3	Low power
4	Low cost

1.1. Quality function deployment

- House of Quality:
 - **Table 2: How list**
 - How are the design requirements of the product?



Example: Automatic light control system

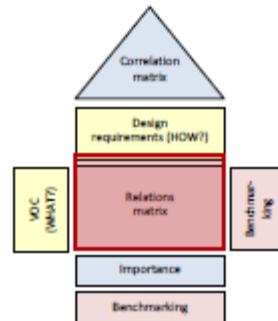
No.	Design requirements
1	Low power microcontroller
2	LED light bulb
3	Solar panel
4	Lithium-ion battery

1.1. Quality function deployment

- House of Quality:

– Table 3: Relation matrix

- shows the relationships between “What” and “How”
- defined by three strength levels: weak, medium, and strong relation.



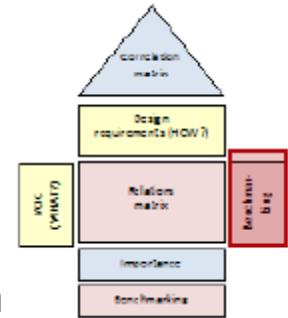
Customer's requirement	Design requirement	Low power micro-controller	LED light bulb	Solar panel	Lithium-ion battery
	Low power micro-controller				
Long life cycle	S	S	S	S	S
Easy to charge energy	W	W	S	S	M
Low power	M	S	W	W	M
Low cost	M	M	M	M	S

1.1. Quality function deployment

- House of Quality:

– Table 4: Benchmarking

- Benchmarking is carried out for “What” and “How”.
- Persons in charge of the product design make this evaluation



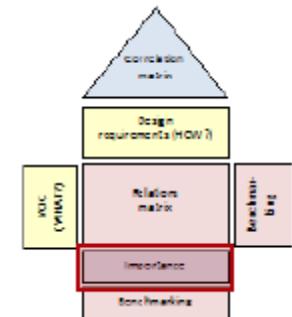
Customer's requirement	Design requirement					Evaluator 1	Evaluator 2	Evaluator 3
		Low power micro-controller	LED light bulb	Solar panel	Lithium-ion battery			
Long life cycle	S	S	S	S				X
Easy to charge energy	W	W	S	M		X		
Low power	M	S	W	M		X		
Low cost	M	M	M	S		X		

1.1. Quality function deployment

- House of Quality:

- **Table 5: Importance level**

- Create a value for each relationship between client and design requirement
 - The personnel in charge of the system design make this evaluation.



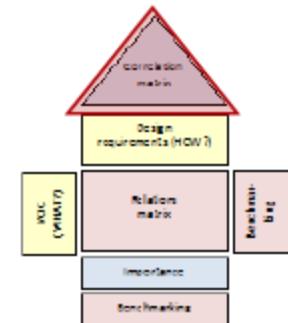
1.1. Quality function deployment

- House of Quality:
 - Table 5: Importance level

Customer's requirement		Design requirement	Low power micro-controller	LED light bulb	Solar panel	Lithium-ion battery	Evaluator 1	Evaluator 2	Evaluator 3
What	Importance						Bad	Average	Good
Long life cycle	4	S	S	S	S				X
Easy to charge energy	1	W	W	S	M		X		
Low power	2	M	S	W	M		X		
Low cost	3	M	M	M	S		X		
Importance		52	64						
Strong = 9			Medium = 3			Weak = 1			

1.1. Quality function deployment

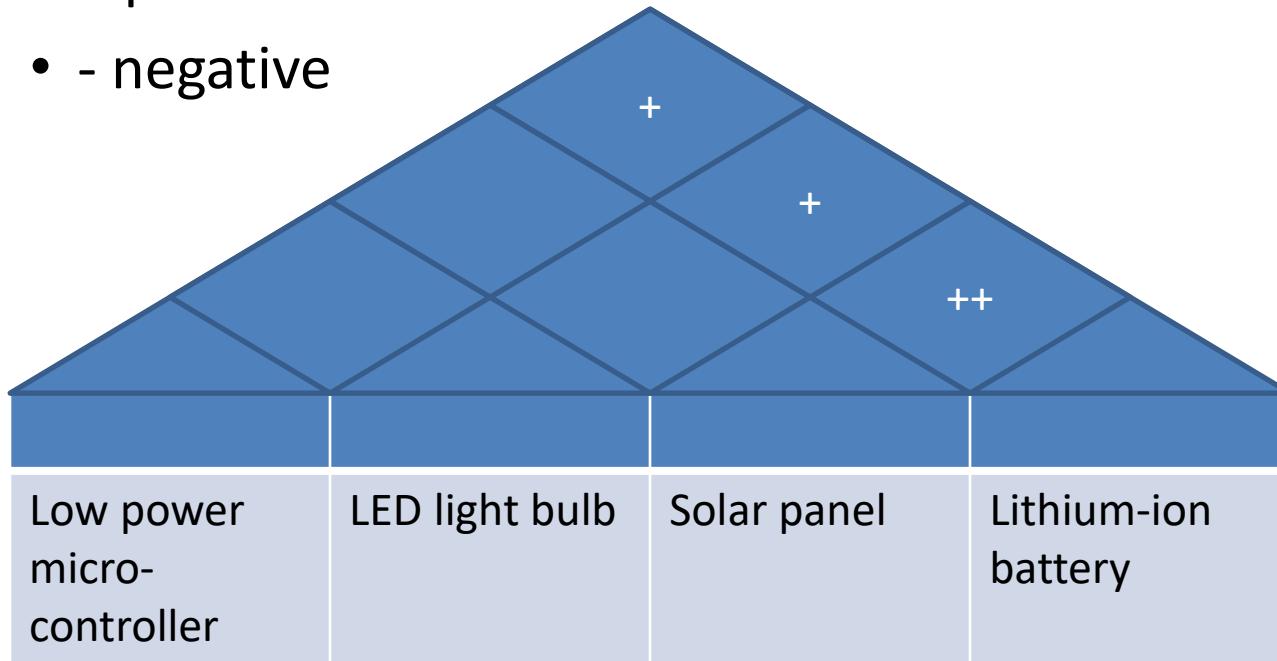
- House of Quality:
 - **Table 6: Correlation matrix**
 - is a triangular table.
 - describes the strength of the relationships between the design requirements.
 - is to identify which requirements support each other and which ones do not.



1.1. Quality function deployment

- House of Quality:
 - **Table 6: Correlation matrix**

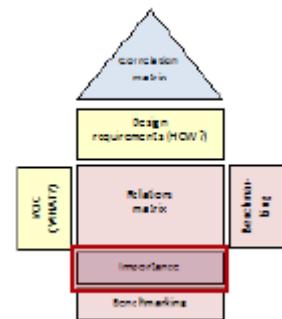
- ++ Strong positive
- + positive
- - negative



1.1. Quality function deployment

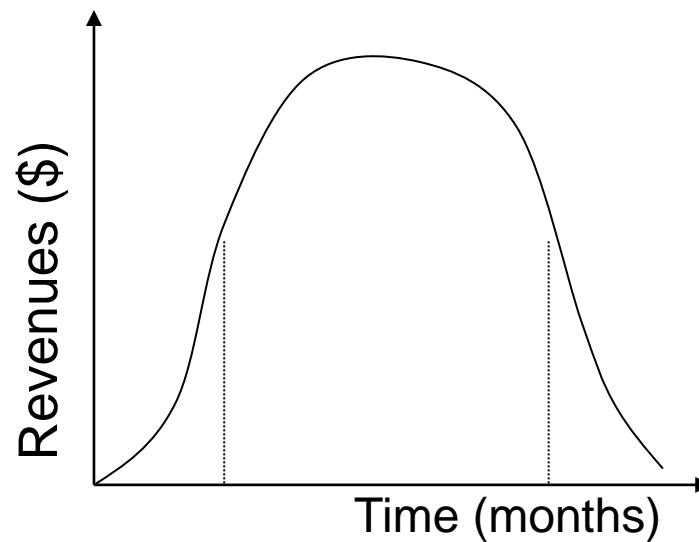
Conclusion:

- The House of Quality helps to identify :
 - **critical technical components** that require change.
 - **issues** that may never have surfaced before.
- Quality control measures needed to produce a product that fulfills both customer needs and producer needs within a shorter development cycle time.



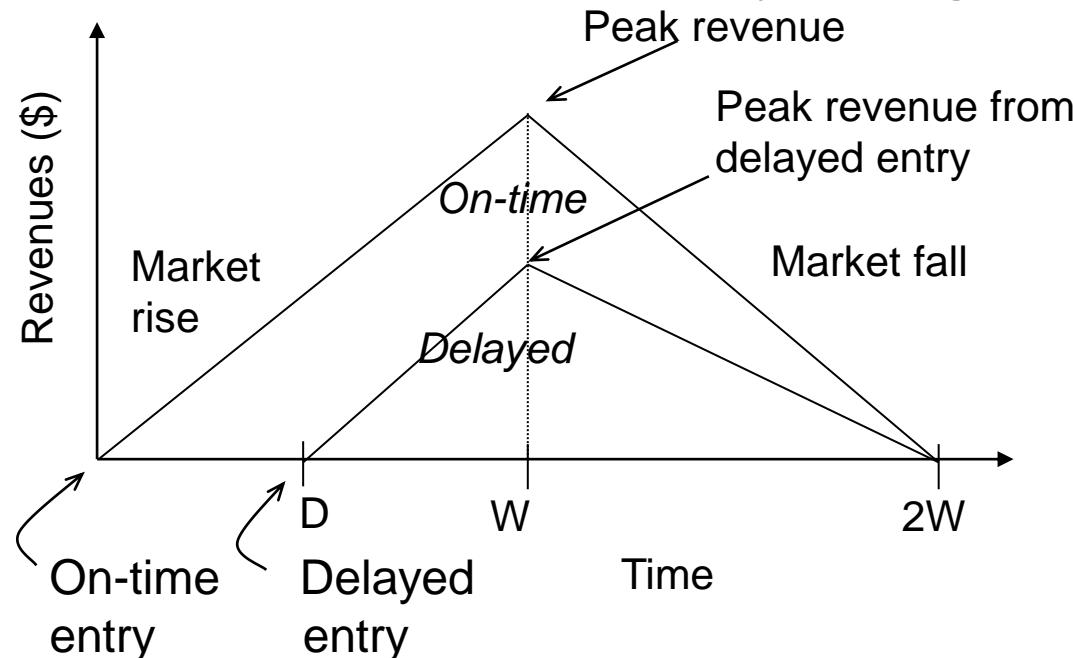
1.2. Cost Analysis

- Time required to develop a product to the point it can be sold to customers
- Market window
 - Period during which the product would have highest sales
- Average time-to-market constraint is about 8 months
- Delays can be costly



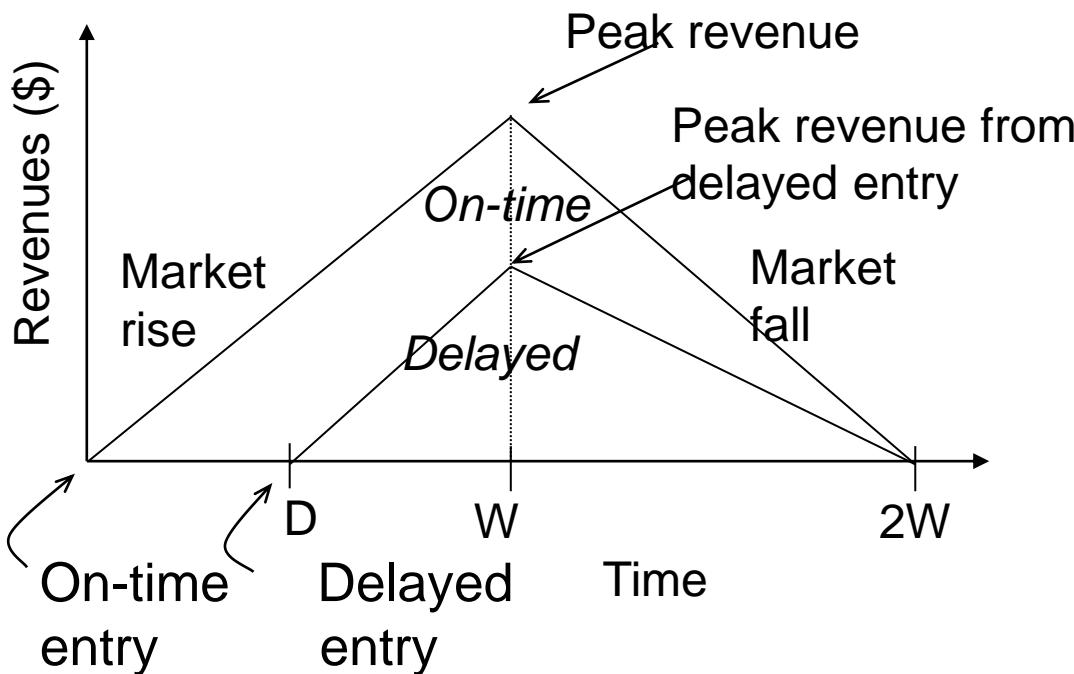
Losses due to delayed market entry

- Simplified revenue model
 - Product life = $2W$, peak at W
 - Time of market entry defines a triangle, representing market penetration
 - Triangle area equals revenue
- Loss
 - The difference between the on-time and delayed triangle areas



Losses due to delayed market entry (cont.)

- Area = $1/2 * \text{base} * \text{height}$
 - On-time = $1/2 * 2W * W$
 - Delayed = $1/2 * (W-D+W)*(W-D)$
- Percentage revenue loss = $(D(3W-D)/2W^2)*100\%$



- Try some examples
- Lifetime $2W=52$ wks, delay $D=4$ wks

$$(4*(3*26 - 4)/2*26^2) = 22\%$$
 - Lifetime $2W=52$ wks, delay $D=10$ wks

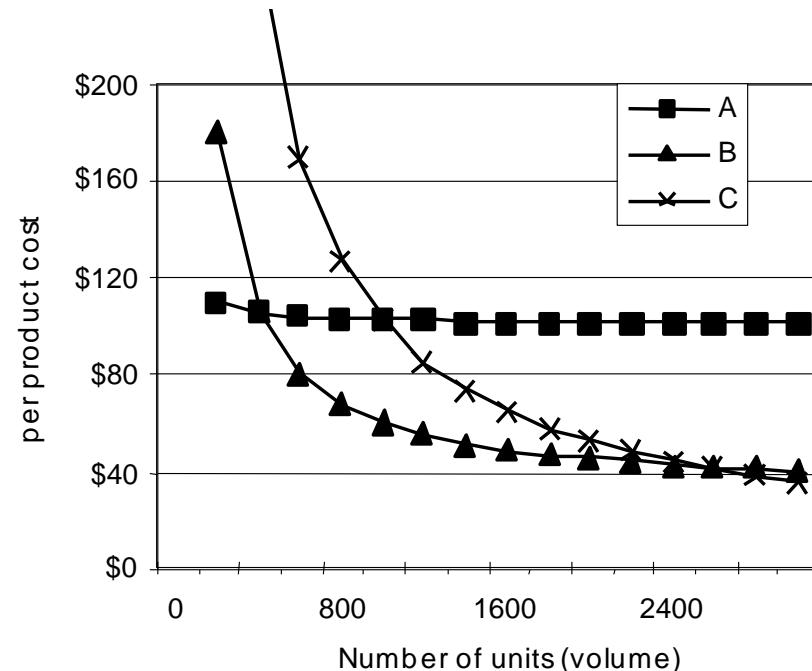
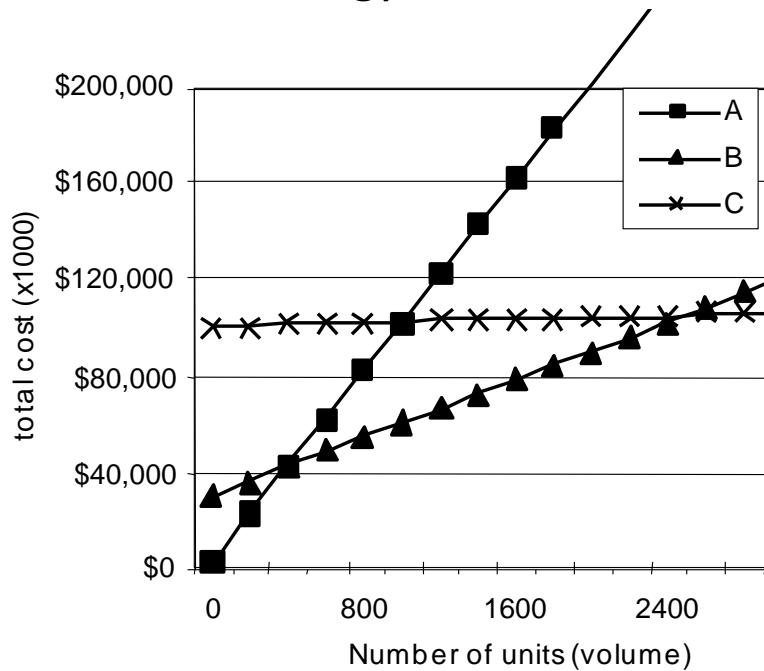
$$(10*(3*26 - 10)/2*26^2) = 50\%$$
 - Delays are costly!

NRE and unit cost metrics

- Costs:
 - Unit cost: the monetary cost of manufacturing each copy of the system, excluding NRE cost
 - NRE cost (Non-Recurring Engineering cost): The one-time monetary cost of designing the system
 - $\text{total cost} = \text{NRE cost} + \text{unit cost} * \# \text{ of units}$
 - $\text{per-product cost} = \text{total cost} / \# \text{ of units}$
 $= (\text{NRE cost} / \# \text{ of units}) + \text{unit cost}$
 - Example
 - NRE=\$2000, unit=\$100
 - For 10 units
 - $\text{total cost} = \$2000 + 10 * \$100 = \$3000$
 - $\text{per-product cost} = \underbrace{\$2000 / 10}_{\$200} + \$100 = \$300$
- Amortizing NRE cost over the units
results in an additional \$200 per unit*

NRE and unit cost metrics

- Compare technologies by costs -- best depends on quantity
 - Technology A: NRE=\$2,000, unit=\$100
 - Technology B: NRE=\$30,000, unit=\$30
 - Technology C: NRE=\$100,000, unit=\$2



- But, must also consider time-to-market

Class assignment

1. The design of a particular disk drive has an NRE cost of \$100,000 and a unit cost of \$20. How much will we have to add to the cost of the product to cover our NRE cost, assuming we sell: (a) 100 units, and (b) 10,000 units.
2. (a) Create a general equation for product cost as a function of unit cost, NRE cost, and number of units, assuming we distribute NRE cost equally among units.

(b) Create a graph with the x-axis the number of units and the y-axis the product cost, and then plot the product cost function for an NRE of \$50,000 and a unit cost of \$5.

2. Design Optimization

- Hardware / software partitioning
 - which functions should be performed in hardware, and which in software?
 - the more functions in software, the lower will be the product cost

2. Design Optimization

Design challenge – optimizing design metrics

- Common metrics
 1. **Unit cost**: the monetary cost of manufacturing each copy of the system, excluding NRE cost
 2. **NRE cost (Non-Recurring Engineering cost)**: The one-time monetary cost of designing the system
 3. **Size**: the physical space required by the system
 4. **Performance**: the execution time or throughput of the system
 5. **Power**: the amount of power consumed by the system
 6. **Flexibility**: the ability to change the functionality of the system without incurring heavy NRE cost

2. Design Optimization

- Common metrics (continued)

7. **Time-to-prototype**: the time needed to build a working version of the system
8. **Time-to-market**: the time required to develop a system to the point that it can be released and sold to customers
9. **Maintainability**: the ability to modify the system after its initial release
10. **Correctness**: check the functionality throughout the process of designing the system; insert test circuitry to check that manufacturing was correct
11. **Safety**: the probability that the system will not cause harm

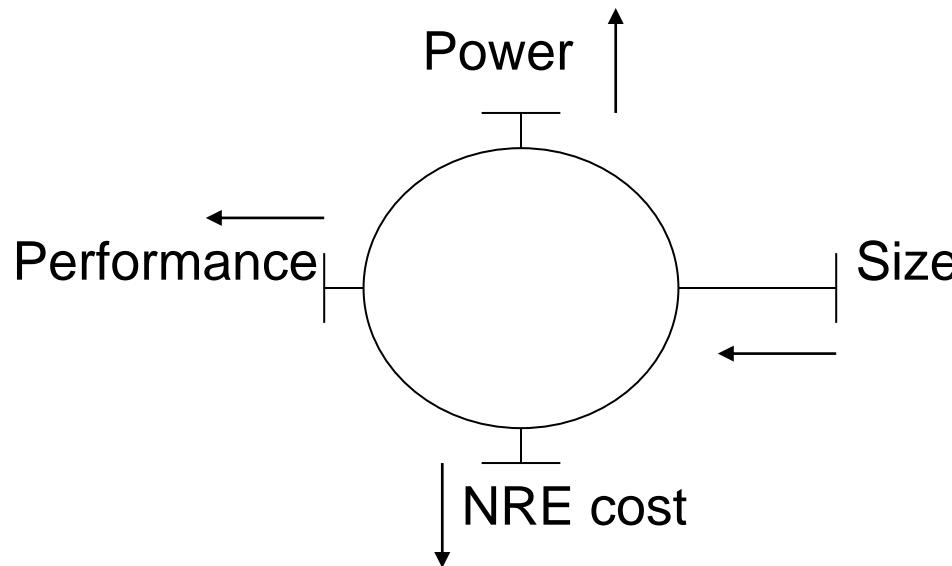
The performance design metric

- Widely-used measure of system, widely-abused
 - Clock frequency, instructions per second – not good measures
 - Digital camera example – a user cares about how fast it processes images, not clock speed or instructions per second
- Latency (response time)
 - Time between task start and end
 - e.g., Camera's A and B process images in 0.25 seconds
- Throughput
 - Tasks per second, e.g. Camera A processes 4 images per second
 - Throughput can be more than latency seems to imply due to concurrency, e.g. Camera B may process 8 images per second (by capturing a new image while previous image is being stored).
- *Speedup of B over S = B's performance / A's performance*
 - Throughput speedup = $8/4 = 2$

2. Design Optimization

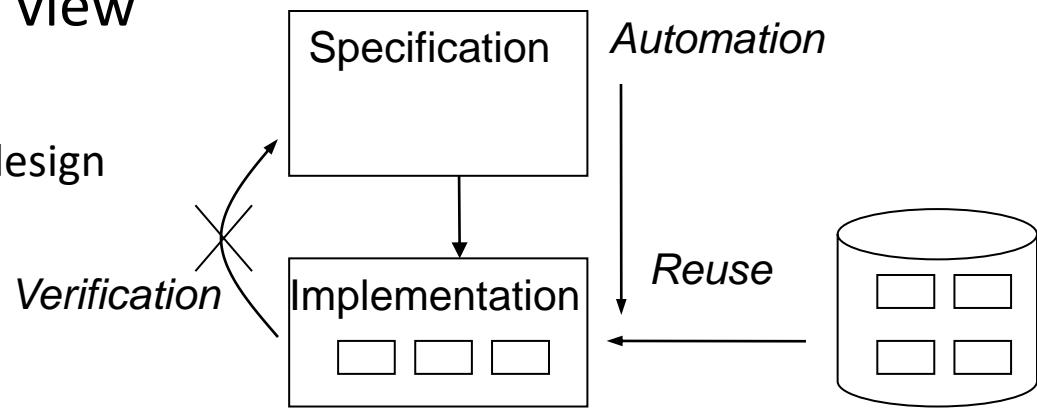
Design metric competition -- improving one may worsen others

- Expertise with both **software and hardware** is needed to optimize design metrics
 - Not just a hardware or software expert, as is common
 - A designer must be comfortable with various technologies in order to choose the best for a given application and constraints



Improving productivity

- Design technologies developed to improve productivity
- We focus on technologies advancing hardware/software unified view
 - **Automation**
 - Program replaces manual design
 - Synthesis
 - **Reuse**
 - Predesigned components
 - Cores
 - General-purpose and single-purpose processors on single IC
 - **Verification**
 - Ensuring correctness/completeness of each design step
 - Hardware/software co-simulation



Class assignment

1. Consider the following embedded systems: a pager, a computer printer, and an automobile cruise controller. Create a table with each example as a column, and each row one of the following design metrics: unit cost, performance, size, and power. For each table entry, explain whether the constraint on the design metric is very tight. Indicate in the performance entry whether the system is highly reactive or not.

2. List three pairs of design metrics that may compete, providing an intuitive explanation of the reason behind the competition.



3. Design Verification

- Process of ensuring system behaves as intended
- System is bug-free
- Functional Correctness
- Important Step
- Nearly 70% of development time



Bugs are costly

- Pentium bug
 - Intel Pentium chip, released in 1994 produced error in floating point division
 - Cost : \$475 million
- ARIANE Failure
 - In December 1996, the Ariane 5 rocket exploded 40 seconds after take off . A software components threw an exception
 - Cost : \$400 million payload.
- Therac-25 Accident :
 - A software failure caused wrong dosages of x-rays.
 - Cost: Human Loss.

Rigorous V&V Essential



Classification

- **Design Verification**
 - Design or Coding errors
 - Specification bugs
 - Simulation and Formal Verification
- **Implementation Testing**
 - Translation/Synthesis Errors
 - Fabrication defects



Traditional Design Verification

- Testing & Simulation
- Reviews & Walkthroughs
- Inadequate for safety-critical systems
- Detects presence of bugs not absence
- Late Detection of bugs
- When to stop testing
 - Coverage criteria
 - ~70% of time



Testing/Simulation

Run (model of the) system with test vectors

- Models at various levels
 - HLL, behavioral, State Machines, RTL, Gate level (HW)
- Functional and Delay models for timing verification

Various Issues

- What are the test vectors ?
 - Choose those that reveal the features or lack of them
 - Corner cases, boundary conditions

Issues in Simulation

- How to generate vectors ?
 - Based upon specification
 - designer's understanding
 - Random vectors
- How to evaluate simulation results ?
 - Use Golden Model
 - Use specifications
- When to stop simulating ?
 - Statement, condition and state coverage of the model
 - Feature coverage over specification
 - Time to market

Simulation

- Widely used validation technique
- Many powerful commercial simulators
- Many speed-up techniques – FPGA based hardware emulators
- Many simulations aids – assertion checkers and event monitors
- Many automatic tools for test bench creation
 - Specman (Verisity), Vera (synopsys)
 - automatic generation of tests, monitors, functional models, scoreboards
 - Coverage analysis tools



Problems in simulation

- Simulation is slow, requires billions of vectors for large designs
- Exhaustive simulation infeasible
- Coverage of design functionality unknown
- Correctness of golden model suspect
- Bugs lurk deep in designs that get revealed after complex input sequences



Formal Methods

- More rigorous approach
- Founded on Mathematical methods
- Proves correctness of Systems
- Increased confidence
- Early Detection of bugs
 - Design Verification
- Complementary to traditional techniques



Formal Verification

- Formally check a formal model of the system against a formal specification
- Formal : Mathematical, Precise, unambiguous
- Static analysis
- No test vector
- Exhaustive Verification
- Proves absence of bugs
- Complex and subtle bugs caught
- Early Detection



Three-step process

- Three steps are
 - Formal Specification
 - Formal Models
 - Verification
- Formal specification
 - Precise statement of properties
 - System requirements and environmental constraints
 - Logic - PL, FOL, temporal logic
 - Automata, labeled transition systems



Other steps

- **Models**
 - Flexible to model general to specific designs
 - Non-determinism, concurrency, fairness,
 - HLL, Transition systems, automata
- **Verification**
 - Checking that model satisfies specification
 - Static and exhaustive checking
 - Automatic or semi-automatic



An Example (Academic)

```
function gcd(x,y)
{
    /*assume: x > 0 & y >0 & x=a & y=b */
    while (x<>y)
        { if x>y then x=x-y else y=y-x
        }
    /*assert: gcd(x,y) = gcd(a,b) */
}
```



GCD Example (contd.)

- Specification
 - Assume and Assert Conditions
 - environment constraints and requirements
- Model
 - HLL code with precise semantics
- Verification
 - Symbolic Execution
 - Theorem Proving

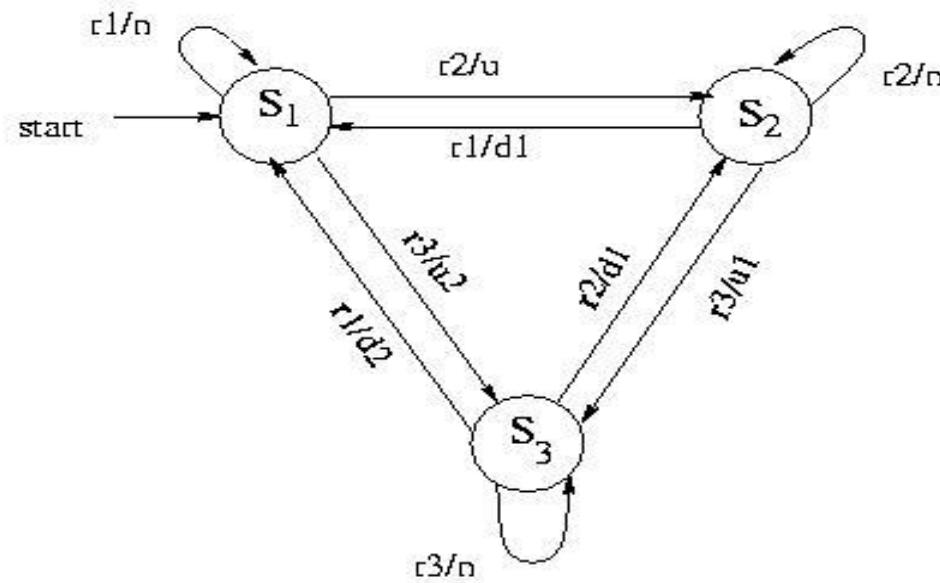


An Industrial Example

```
struct RCD3_data { double X, Y; };

void get_inputsXY(struct RCD3_data *final_data)
{   ret1 = read_from_reg( 1, &InputX );
    /*postfunc ( InputX >= 0 ∧ InputX <= 4095 ) end*/
    change_to_v(InputX, input_src, &tempX );
    /*assert !(tempX < 0 ∨ tempX > 5) end*/
    final_data->X= tempX;
    convert_to_d(1, tempX, final_data);
    /*post (#X final_data >= -180) ∧ (#X final_data <= 180)
    end*/
}
```

An abstract Design example



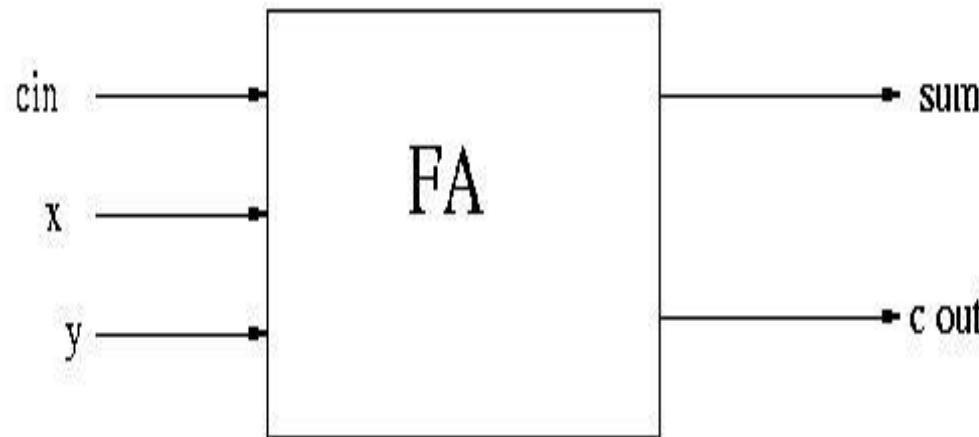
- 3- floor ele
- S_i - in floor
- r_i - request from floor I
- U,d – up or down movement



Lift Controller (contd.)

- Specification
 - When the lift is in motion door is closed
 - Every call is eventually attended
 - The lift is busy attending one of the requests
 - Safety and Liveness Properties
- Verification
 - State space exploration
 - Model Checking
 - Automatic

Third Example



Specification:

- $\text{sum} := (\text{x} \oplus \text{y}) \oplus \text{cin}$
- $\text{cout} := (\text{x} \wedge \text{y}) \vee ((\text{x} \oplus \text{y}) \wedge \text{cin})$



Full Adder (Contd.)

Model

- Boolean expression corr. implementation of full adder
- Verification
 - Comparision of boolean expressions
 - Equivalence Checking
 - Theorem Proving, in general



Some Observations

- Precise statement of requirements and constraints
 - Nothing in the head, one has to commit
 - No ambiguity, no escape
- Precise description of Designs
- Algorithmic or semi-automatic and rigorous techniques for verification



Observation (contd.)

- Verification fails
 - Specification can be wrong or incomplete
 - Design could be too abstract
 - Design is buggy and hence does not satisfy the specification
- Verification succeeds
 - if specification is trivial
 - if specification derived from code
 - if design meets specification



Formal Specification

- Thorough review of Specification essential
- Specification **independent** of design
- Additional Step
- Better done **before** the design starts



Further Observation

- FV complicates the problem
- Creates more work
- **No, not at all**
 - Traditional approaches ignores
 - FV unearths the inherent problems
- Definitely more work
 - **High quality does not come free!**



Formal Verification Techniques

- Three major techniques
 - Equivalence checking (HW)
 - Model checking(HW & SW)
 - Theorem proving(HW & SW)



Equivalence Checking

- Checking equivalence of two similar circuits
- Useful for validating optimizations, scan chain insertions
- Comparison of two boolean expressions – use of BDDs
- Highly automatic and efficient
- Most used formal Verification technique
- Commercial tools :
 - Design VERIFYer (Chrysalis Inc.)
 - Formality (Synopsis, million gates in less than an hour)
 - V-Formal



Model Checking

- Another promising automatic technique
- Checking design models against specification
- Specifications temporal properties and environment constraints; use of temporal logic or automata
- Design models are automata or HDL sub sets
- Checking is automatic and bug traces
- Very effective for control-intensive designs and Protocols
- Many Commercial and academic tools:
- Spin (Bell Labs.), Formal-Check (Cadence), VIS (UCB), SMV (CMU, Cadence)
- In-house tools: Rule Base (IBM), Intel, SUN, Bingo (Fujitsu), etc



Theorem Proving

- Most Powerful technique
- Specification and design are logical formulae
- Checking involves proving a theorem
- Semi-automatic
- High degree of human expertise required
- Mainly confined to academic
- Number of public domain tools
 - Nqthm, STeP, PVS, HOL



Formal verification (experiences)

- Very effective for small control-intensive designs-blocks of hundreds of latches or state bits
- Many subtle bugs have been caught in designs cleared by simulation or testing
- Strong theoretical foundation
- High degree of confidence
- Holds a lot of promise
- Requires a lot more effort and expertise
- Large designs need abstraction
- Many efforts are underway to improve



Systems verified

- Various microprocessors (instruction level verification):
 - DLX pipelined architectures, AAMP5 (avionics applications), FM9001 (32 bit processor), PowerPC
- Floating point units:
 - SRT division (Pentium), recent Intel ex-fpu, ADK IEEE multiplier, AMD division
- Multiprocessor coherence protocols
 - SGI, sun S3.Mp architectures, Gigamax, futurebus+
- Memory subsystems of PowerPC
- Fairisle ATM switch core



Challenges of formal verification

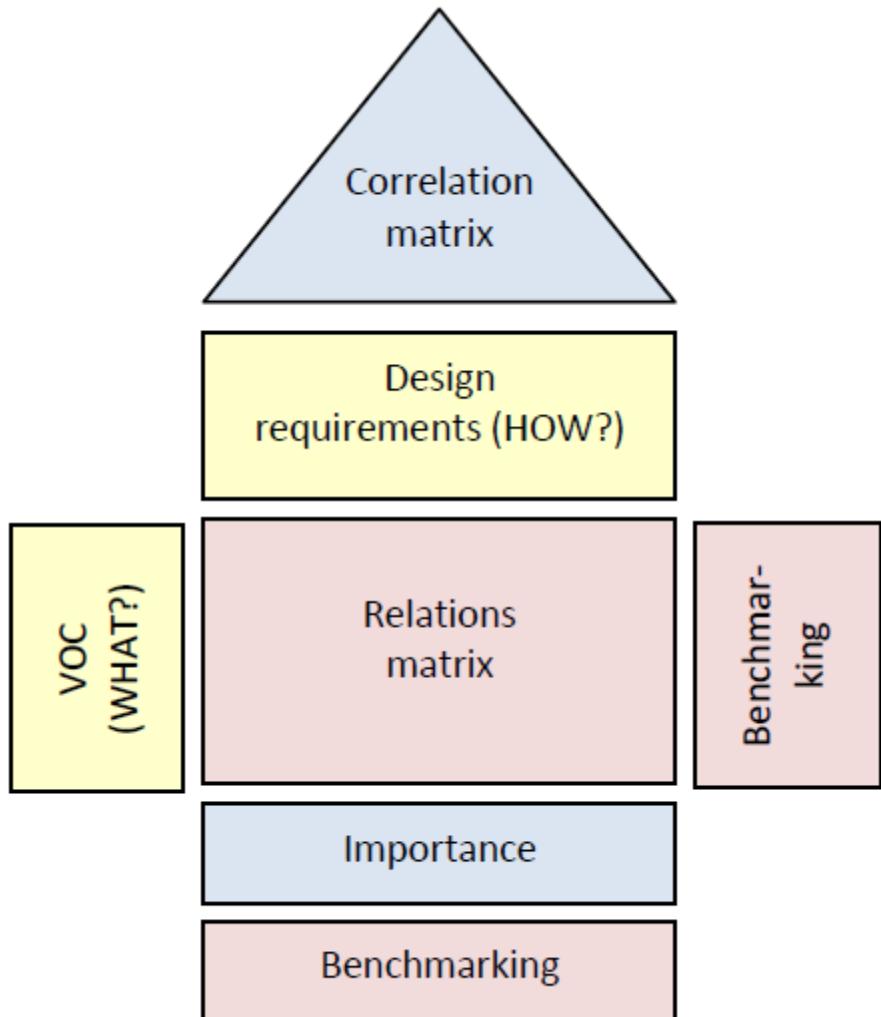
- Complexity of verification
 - Automatic for finite state systems (HW, protocols)
 - Semi-automatic in the general case of infinite state systems (software)
- State explosion problem
 - Symbolic model checking
 - Compositional reasoning
 - Localization Reduction (FormalCheck)
 - Partial Order Reduction (Spin)

Quality Function Deployment - QFD

- **Quality Function Deployment (QFD)** is a method:
 - proposed by Dr. Yoji Akao in 1996
 - to transform qualitative user demands into quantitative parameters
 - to deploy the functions forming quality
 - to deploy methods for achieving the design quality into subsystems and component parts,
 - to specific elements of the manufacturing process.

Quality Function Deployment - QFD

- House of Quality: is a techniques based on QFD
 - Appeared in 1972
 - The house can be divided in “rooms”

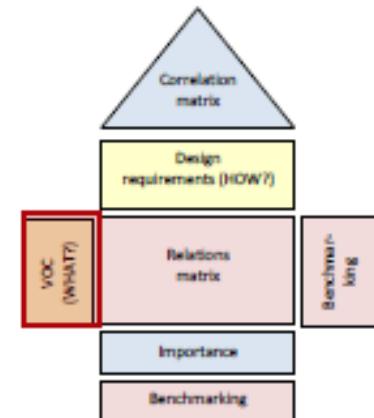


Quality Function Deployment - QFD

- House of Quality:

- **Table 1: What?**

- What is desired in order to reach the new service's development?



Example: Automatic light control system

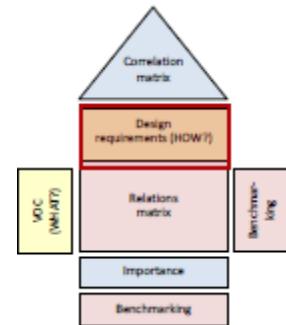
No.	Customer's requirements
1	Long life cycle
2	Easy to charge energy
3	Low power
4	Low cost

Quality Function Deployment - QFD

- House of Quality:

- **Table 2: How list**

- How are the design requirements of the product?



Example: Automatic light control system

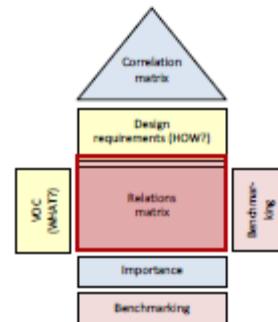
No.	Design requirements
1	Low power microcontroller
2	LED light bulb
3	Solar panel
4	Lithium-ion battery

Quality Function Deployment - QFD

- House of Quality:

- **Table 3: Relation matrix**

- shows the relationships between “What” and “How”
 - defined by three strength levels: weak, medium, and strong relation.



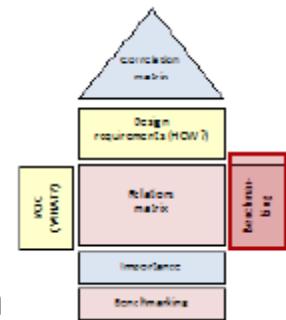
Customer's requirement	Design requirement	Low power micro-controller	LED light bulb	Solar panel	Lithium-ion battery
Long life cycle		S	S	S	S
Easy to charge energy		W	W	S	M
Low power		M	S	W	M
Low cost		M	M	M	S

Quality Function Deployment - QFD

- House of Quality:

- **Table 4: Benchmarking**

- Benchmarking is carried out for “What” and “How”.
 - Persons in charge of the product design make this evaluation

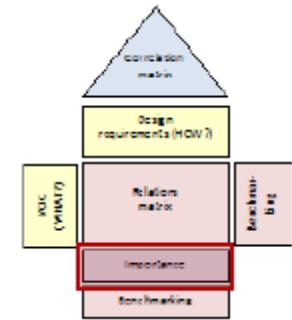


Customer's requirement	Design requirement					Evaluator 1	Evaluator 2	Evaluator 3
		Low power micro-controller	LED light bulb	Solar panel	Lithium-ion battery			
Long life cycle	S	S	S	S				X
Easy to charge energy	W	W	S	M		X		
Low power	M	S	W	M		X		
Low cost	M	M	M	S		X		

Quality Function Deployment - QFD

- House of Quality:
 - **Table 5: Importance level**

- Create a value for each relationship between client and design requirement
- The personnel in charge of the system design make this evaluation.



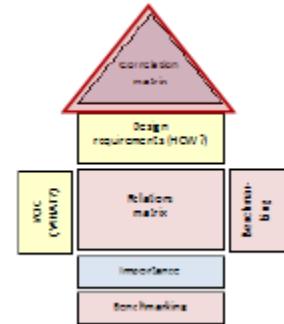
Quality Function Deployment - QFD

- House of Quality:
 - **Table 5: Importance level**

Customer's requirement		Design requirement	Low power micro-controller	LED light bulb	Solar panel	Lithium-ion battery	Evaluator 1	Evaluator 2	Evaluator 3
What	Importance						Bad	Average	Good
Long life cycle	4	S	S	S	S				X
Easy to charge energy	1	W	W	S	M		X		
Low power	2	M	S	W	M		X		
Low cost	3	M	M	M	S		X		
Importance		52	64						
Strong = 9			Medium = 3			Weak = 1			

Quality Function Deployment - QFD

- House of Quality:
 - **Table 6: Correlation matrix**
 - is a triangular table.
 - describes the strength of the relationships between the design requirements.
 - is to identify which requirements support each other and which ones do not.



Quality Function Deployment - QFD

- House of Quality:
 - **Table 6: Correlation matrix**

- ++ Strong positive
- + positive
- - negative

