Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE

innovate achieve lead

BITS Pilani
Pilani Campus

ESZG512/MELZG526/SEZG516
Embedded System Design
Contact Session 1

innovate achieve lead

BITS Pilani
Pilani|Dubai|Goa|Hyderabad

Introduction

innovate achieve lead

CS1 : (Introduction to Embedded Systems)

1	➢ Introduction to the course – evaluation components ➢ Introduction to various Embedded System Components

Welcome Note

- Welcome to the course Embedded System Design.
- This course covers both hardware and software aspects of the embedded technology in considerable depth.

Course Objectives

CO1	Introduce Hardware technologies of Embedded Systems and Hardware Design
CO2	Introduce Software technologies of Embedded Systems and Software Design
CO3	Introduce important practical aspects of Embedded System Design to the students.

Learning Outcomes

No.	Learning Outcomes for the course
LO1	Students should gain an understanding about the embedded system as a whole and its hardware and software components.
LO2	Students should gain a detailed understanding of popular CPU architectures used in embedded systems such as ARM.
LO3	Students should gain a detailed understanding about embedded software stack design, modelling and underlying real time operating system technologies.
LO4	With the acquired knowledge, students should be able to come up with the high-level design of an embedded system from both hardware and software perspectives.

In This Course

- **M1-Introduction to Embedded System Design**
- **M2-Embedded System Design Example**
- **M3-Modelling an Embedded System**
- **M4-Design using ARM Processors - Core V4, Cortex M4**
- **M5-Buses**
- **M6-I/O Interfaces - Normal**
- **M7-I/O Interfaces - Communication**
- **M8-I/O Interfaces - Advanced**
- **M9-Design Examples**
- **M10-Embedded Software Design**
- **M11-Embedded Software: Tasks, Task Management & RTS**
- **M12-Advanced Topics in Embedded Systems**

Evaluation Components

- EC1 - 30% (Assignment(s)/ Lab(s), Quiz(s)) - Online
- EC2 - 30% (Mid semester Test) – Closed Book
- EC3 - 40% (Comprehensive Examination) – Open Book

Detailed Plan for Lab Sessions: - Instructor : Prof. S.S. Kendre

Lab No	Lab Objective	Date and Time
1	ARMv4 Assembly Language Programming Example using Keil uVision on Simulator	3rd Feb 2024 (8:00 PM to 10:00 PM
2	ARM Cortex M4 Assembly Language Programming using Keil uVision on Simulator. Remote Lab Demo (Assignment 1 Release)	10th Feb 2024 (8:00 PM to 10:00 PM
3	Remote Lab implementation on LPC2378 using Keil uVision (Assignment 2 Release)	2nd March 2024 (8:00 PM to 10:00 PM
4	Remote Lab implementation on STM32 using Keil uVision	6th April 2024 (8:00 PM to 10:00 PM

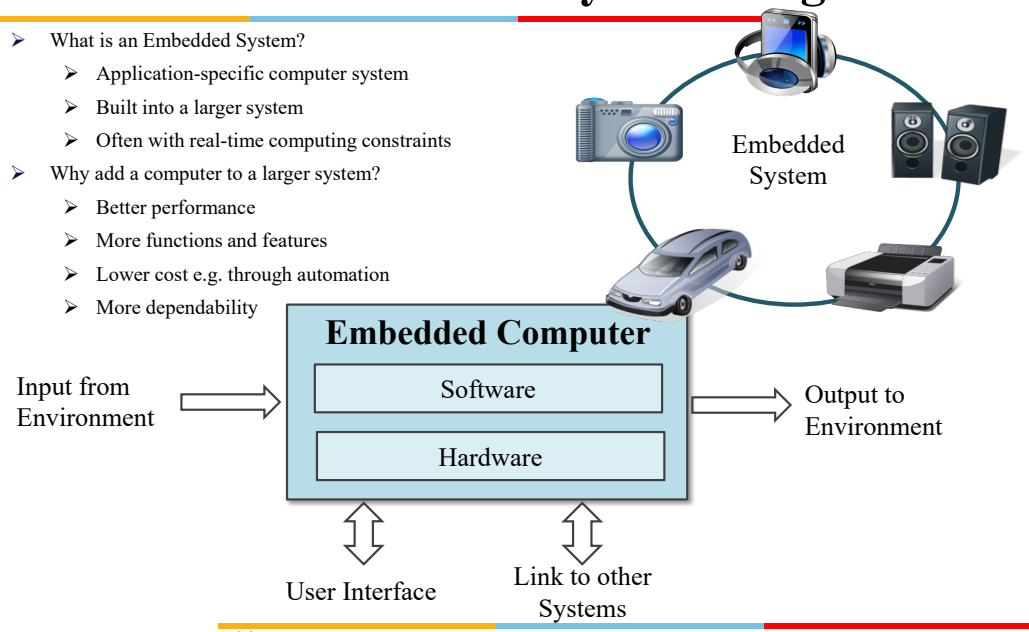
Introduction to Embedded System Design

- **System Definition**
- A way of working, organizing or performing one or many tasks according to a fixed set of rules, program or plan.
- Also, an arrangement in which all units assemble and work together according to a program or plan.

- **Examples of Systems:**
- Time display system – A watch
- Automatic cloth washing system – A washing machine

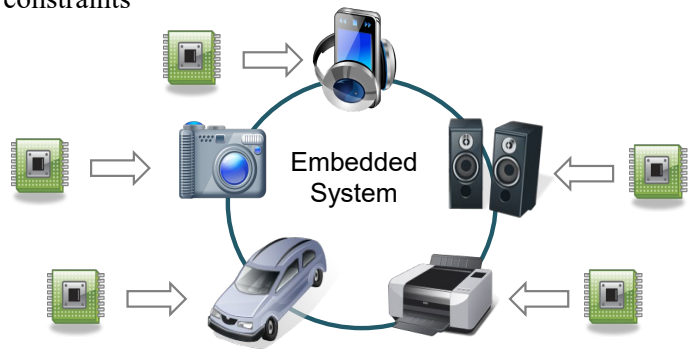
Introduction to Embedded System Design

- What is an Embedded System?
  - Application-specific computer system
  - Built into a larger system
  - Often with real-time computing constraints
- Why add a computer to a larger system?
  - Better performance
  - More functions and features
  - Lower cost e.g. through automation
  - More dependability



Embedded Systems

- **Embedded System**
- *An Electronic/Electro mechanical system which is designed to perform a specific function and is a combination of both hardware and firmware (Software)*
  - Typically implemented using MCUs
  - Often integrated into a larger mechanical or electrical system
  - Usually has real-time constraints



Example Embedded System: Bike Computer

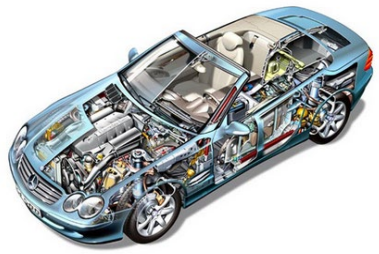
- **Functions**
  - Speed and distance measurement
- **Constraints**
  - Size
  - Cost
  - Power and Energy
  - Weight
- **Inputs**
  - Wheel rotation indicator
  - Mode key
- **Output**
  - Liquid Crystal Display
- Use Low Performance Microcontroller
  - 8-bit, 10 MIPS

Input:
Wheel rotation
Mode key

Output:
Display speed and distance

Gasoline Automobile Engine Control Unit

- Functions
  - Fuel injection
  - Air intake setting
  - Spark timing
  - Exhaust gas circulation
  - Electronic throttle control
  - Knock control
- Constraints
  - Reliability in harsh environment
  - Cost
  - Weight
- Many Inputs and Outputs
  - Discrete sensors & actuators
  - Network interface to rest of car
- Use High Performance Microcontroller
  - E.g. 32-bit, 3 MB flash memory, 150 - 300 MHz



Embedded Systems - Components

Components - Hardware

Embedded Systems - Components

Components - Software

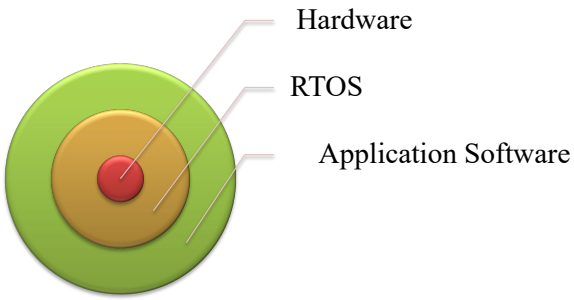
Application Specific Software – series of tasks

Constraints- available memory, available processing power, power dissipation limit

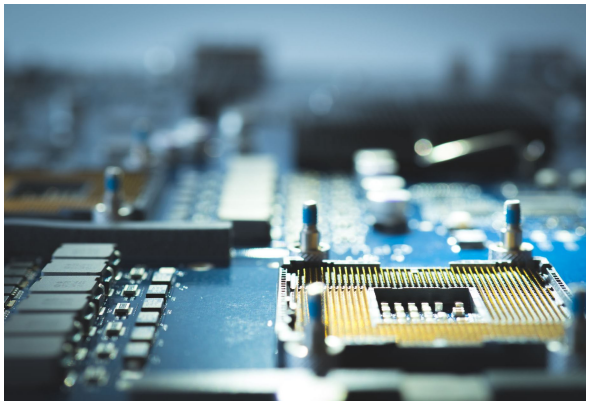
RTOS

- Supervise application s/w
- Allocate resources
- Context switching
- Latencies – meet the deadlines
- Scalable

Embedded Systems - Components

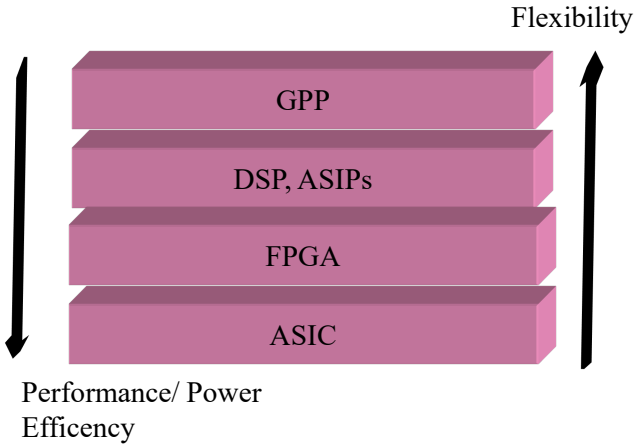


Embedded Systems - Components



Hardware Components

Processor



Processor Architectures

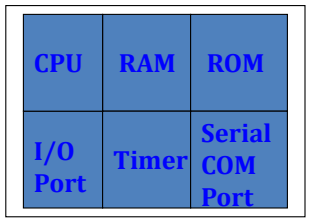
General-purpose microprocessor

- Microprocessor means General-purpose microprocessor such as Intel's x86(8086, 80286, 80386,80486, Pentium) or Motorola's 680x0(68000,68010,68020…)
- These microprocessors contain no RAM, ROM, or I/O on the CPU chip itself. The system designer must add RAM, ROM, I/O ports, and timers externally to make them functional.
- For this reason, they are commonly referred to as 'General-purpose microprocessor'.
- Have the advantage of versatility on the amount of RAM, ROM, and I/O ports.
- This is not in the case with microcontroller.

Processor Architectures

Microcontroller

- A microcontroller has a CPU (a microprocessor) in addition to a fixed amount of on-chip ROM, RAM, number of I/O ports & timers all on a single chip.
- Ideal for many applications in which cost and space are critical.
- In many applications, the space it takes, the power it consumes, and the price per unit are much more critical considerations than the computing power.
- Example：Motorola's 6811, Intel's 8051, Zilog's Z8 and PIC 16X, STM32, LPC2378



Microprocessor Vs Microcontroller

Microprocessor	Microcontroller
A silicon chip representing a Central Processing Unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of Instructions	A microcontroller is a highly integrated chip that contains a CPU, scratch pad RAM, Special and General-purpose Register Arrays, On Chip ROM/FLASH memory for program storage, Timer and Interrupt control units and dedicated I/O ports
It is a dependent unit. It requires the combination of other chips like Timers, Program and data memory chips, Interrupt controllers etc for functioning	It is a self-contained unit and it doesn't require external Interrupt Controller, Timer, UART etc for its functioning
Most of the time general purpose in design and operation	Mostly application oriented or domain specific
Doesn't contain a built in I/O port. The I/O Port functionality needs to be implemented with the help of external Programmable Peripheral Interface Chips like 8255	Most of the processors contain multiple built-in I/O ports which can be operated as a single 8 or 16 or 32 bit Port or as individual port pins
Targeted for high end market where performance is important	Targeted for embedded market where performance is not so critical (At present this demarcation is invalid)
Limited power saving options compared to microcontrollers	Includes lot of power saving features

Processor Architectures

- Based on number of memory & data buses used, there are 3 types of architectures for the processor.

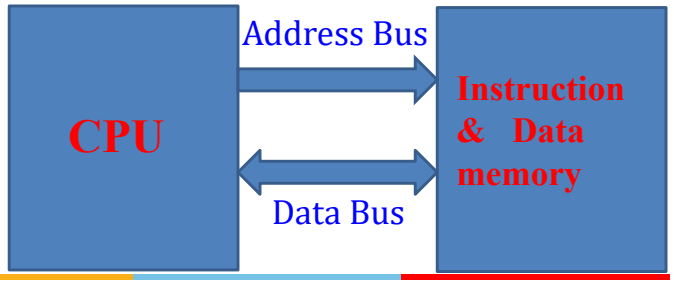
1. Von Neumann architecture.
2. Harvard architecture.
3. Super-Harvard architecture.

Von Neumann architecture (Princeton)

- One memory chip which stores both instruction & data.
- Processor interact with memory through address & data buses to fetch instructions & data.
- Single memory address for either program code or data, but not for both.
- No separation between data and program memory. Program & data fetches are done using time division multiplexing which affect the performance.

Example:

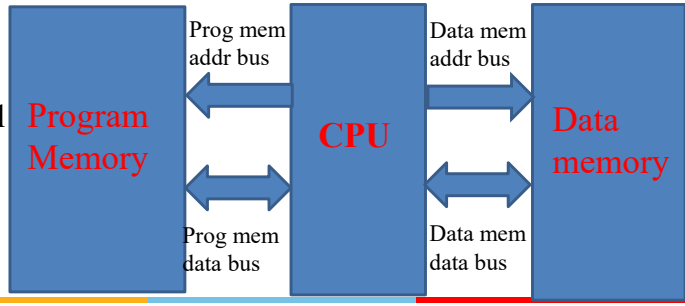
1. Motorola 68HC11 microcontroller
2. 8085 microprocessor.



Harvard architecture

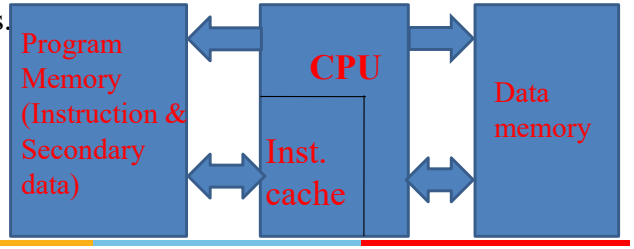
- Use two separate memories for program & data with their independent address & data buses.
- Because of two different streams of data & program, there is no need to have any time division multiplexing of address and data buses.
- This architecture is much more efficient because accessing of data & instruction simultaneously, so very fast.
- Uses same address in different memories for code & data.
- Example:

1.MCS-51 family of microcontrollers by Intel
2. PIC microcontrollers by Microchip
3.Digital signal processor



Super Harvard Architecture

- Cache used to store instructions, leaving both instruction bus and data bus free to fetch operands.
- Harvard Architecture + cache = Extended Harvard Architecture or Super Harvard Architecture
- In Harvard data memory is accessed more frequently than program memory. Provision has to made to store some secondary data in program memory to balance the load on both memory blocks.
- This architecture is proposed by Analog Devices, used extensively in Digital Signal Processors.
- Examples: SHARC DSP, Tiger SHARC



Harvard V/s Von-Neumann

Harvard Architecture	Von-Neumann Architecture
Separate buses for Instruction and Data fetching	Single shared bus for Instruction and Data fetching
Easier to Pipeline, so high performance can be achieved	Low performance Compared to Harvard Architecture
Comparatively high cost	Cheaper
No memory alignment problems	Allows self modifying codes
Since data memory and program memory are stored physically in different locations, no chances for accidental corruption of program memory	Since data memory and program memory are stored physically in same chip, chances for accidental corruption of program memory

CISC and RISC Architectures

- Complex Instruction Set Computer (CISC)
  - Large number of complex instructions
  - Low level
  - Facilitate the extensive manipulation of low-level computational elements and events such as **memory**, **binary arithmetic**, and addressing.
- Examples of CISC processors are the
  - System/360(excluding the 'scientific' Model 44),
  - VAX,
  - PDP-11,
  - Motorola 68000 family
  - Intel x86 architecture based processors.

CISC Architectures

- **Pro's**
- Emphasis on hardware
- Includes multi-clock complex instructions
- Memory-to-memory: "LOAD" and "STORE" incorporated in instructions
- Small code sizes, high cycles per second
- Transistors used for storing complex instructions
- **Con's**
- That is, the incorporation of older instruction sets into new generations of processors tended to force growing complexity.
- Many specialized CISC instructions were not used frequently enough to justify their existence.
- Because each CISC command must be translated by the processor into tens or even hundreds of lines of microcode, it tends to run slower than an equivalent series of simpler **commands** that do not require so much translation.

RISC Architectures

- Reduced Instruction Set Computer
  - Small number of instructions
  - instruction size constant
  - bans the indirect addressing mode
  - retains only those instructions that can be overlapped and made to execute in one machine cycle or less.
- RISC Examples
  - Apple iPods (custom ARM7TDMI SoC)
  - Apple iPhone (Samsung ARM1176JZF)
  - Palm and PocketPC PDAs and smartphones (Intel XScale family, Samsung SC32442 - ARM9)
  - Nintendo Game Boy Advance (ARM7)
  - Nintendo DS (ARM7, ARM9)
  - Sony Network Walkman (Sony in-house ARM based chip)

RISC Architectures

- **Pro's**
- Emphasis on software
- Single-clock, reduced instruction only
- Register to register: "LOAD" and "STORE" are independent instructions
- Low cycles per second, large code sizes
- Spends more transistors on memory registers

CISC and RISC Architectures

- **Performance**
- The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction.
- RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program.

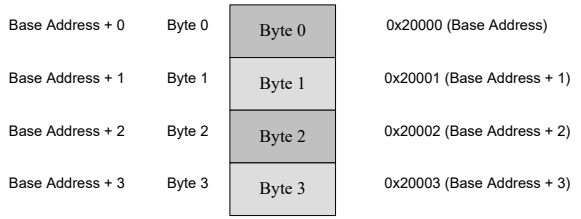
RISC V/s CISC Processors/Controllers

RISC	CISC
Lesser no. of instructions	Greater no. of Instructions
Instruction Pipelining and increased execution speed	Generally, no instruction pipelining feature
Orthogonal Instruction Set (Allows each instruction to operate on any register and use any addressing mode)	Non-Orthogonal Instruction Set (All instructions are not allowed to operate on any register and use any addressing mode. It is instruction specific)
Operations are performed on registers only, the only memory operations are load and store	Operations are performed on registers or memory depending on the instruction
Large number of registers are available	Limited no. of general purpose registers
Programmer needs to write more code to execute a task since the instructions are simpler ones	Instructions are like macros in C language. A programmer can achieve the desired functionality with a single instruction which in turn provides the effect of using more simpler single instructions in RISC
Single, Fixed length Instructions	Variable length Instructions
Less Silicon usage and pin count	More silicon usage since more additional decoder logic is required to implement the complex instruction decoding.
With Harvard Architecture	Can be Harvard or Von-Neumann Architecture

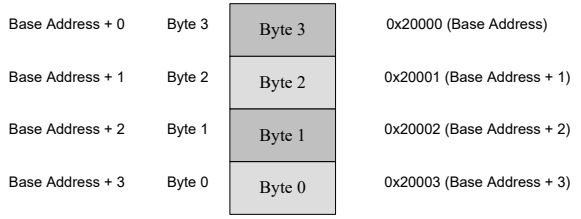
Big-endian V/s Little-endian processors

- Endianness specifies the order in which the data is stored in the memory by processor operations in a multi byte system (Processors whose word size is greater than one byte).
- Suppose the word length is two byte then data can be stored in memory in two different ways
  - Higher order of data byte at the higher memory and lower order of data byte at location just below the higher memory
  - Lower order of data byte at the higher memory and higher order of data byte at location just below the higher memory
- ***Little-endian*** means the lower-order byte of the data is stored in memory at the lowest address, and the higher-order byte at the highest address. (The little end comes first)
- ***Big-endian*** means the higher-order byte of the data is stored in memory at the lowest address, and the lower-order byte at the highest address. (The big end comes first.)

Big-endian V/s Little-endian processors



Little-endian Operation



Big-endian Operation

Memory

Int RAM
Ext RAM
Cache
Flash Memory/EEPROM – Ext/Int
System Ports
ROM/PROM

Memory

Specifications

Cache	Main Memory
• 64K x 32 • Organized as 128 0.5 K Blocks • Direct Mapped	• Memory address – 32 bits • 256 K x 32 • Organized as 4 pages – each page has 128 0.5 K blocks

Memory - Direct Mapped Cache

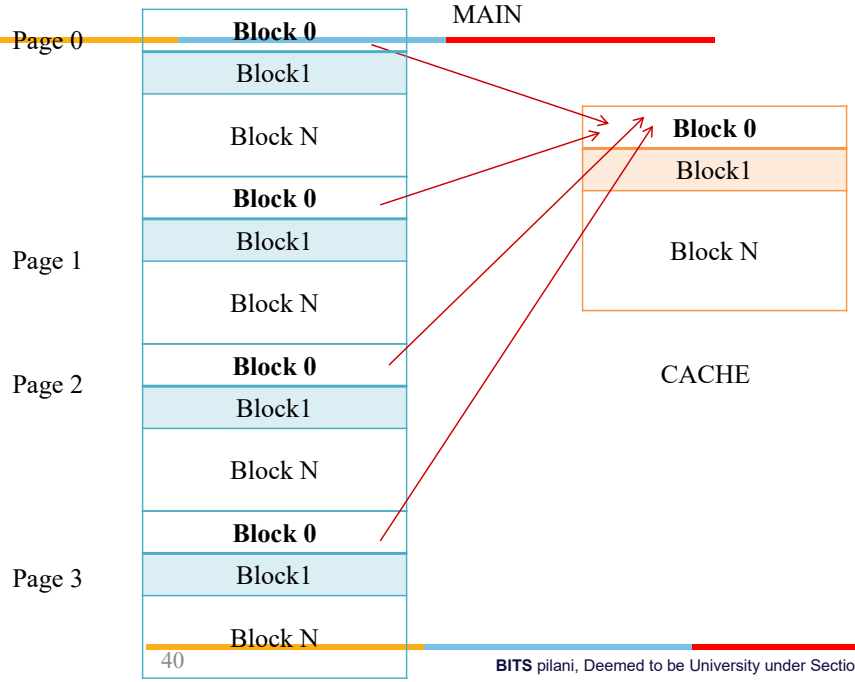
Page 0 … ……...Page 3

Block 0	Data 0 Data 511
Block 1	Data 0 Data 511
Block 127	Data 0 Data 511

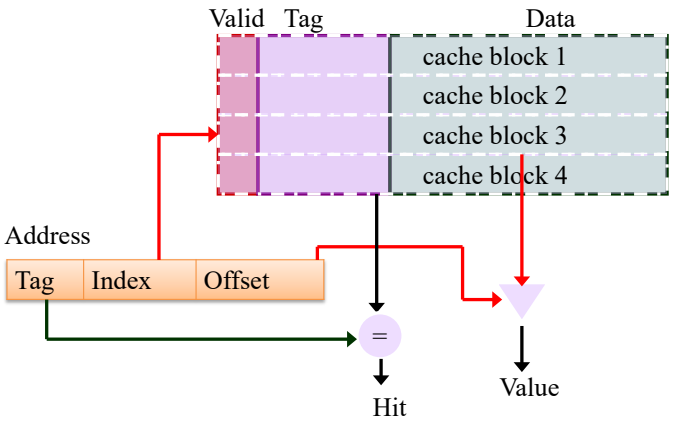
Block 0	Block 0	Block 0	Block 0	Data 0 Data 511
Block 1	Block 1	Block 1	Block 1	Data 0 Data 511
Block 127	Block 127	Block 127	Block 127	Data 0 Data 511

Cache **Main Memory**

Cache Mapping



Cache Mapping



Dirty Bit

Write Through

Write Back

Memory – Set Associative Mapped Cache

Cache Organization

Cache

64K x 32

128 0.5 K blocks

Two sets

Main Memory

256 K x 32

512 Blocks - m

64 Groups – n

Group No m mod n

Memory – Set Assosicative Mapped Cache

Main Memory Organization

Block						Group
0	64	128		384	448	0
1	65	129		385	449	1
2	66	130		386	450	2
63	127	191		447	511	63
T0	**T1**	**T2**		**T6**	**T7**	

Memory – Set Associative Mapped Cache

Mapping

0
1
63

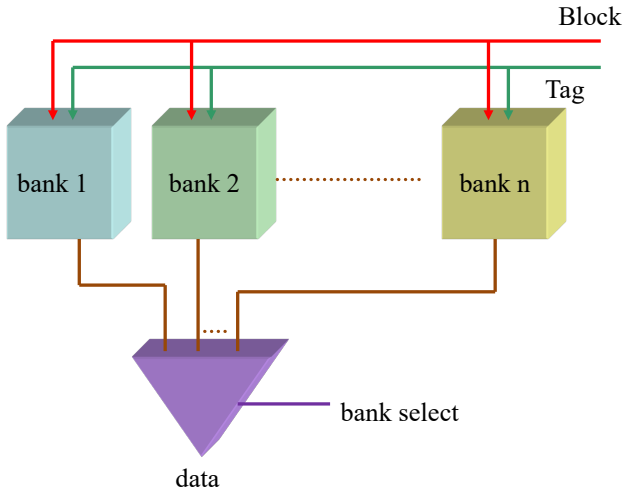
TAG

2	**7**
1	2
2	6

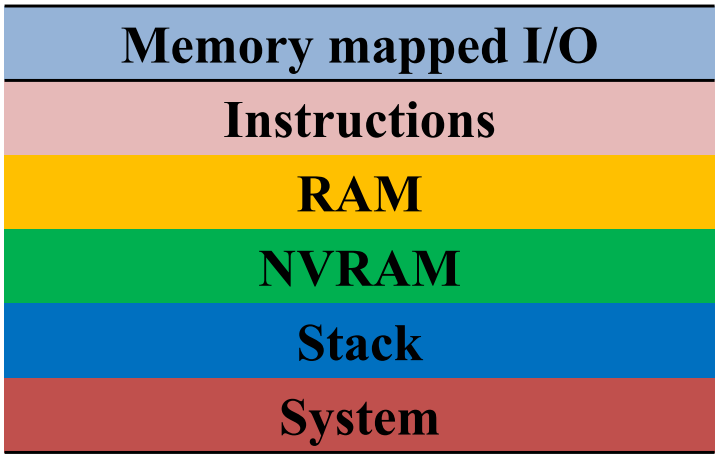
Cache

0	**128**	**448**
1	65	129
63	191	447
	S1	**S2**

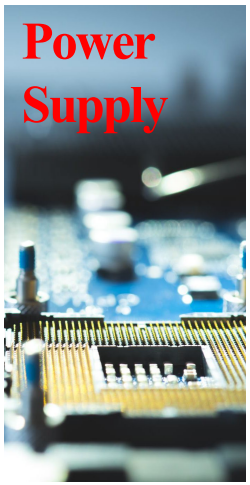
Memory – Set Associative Mapped Cache



Memory MAP



Other Hardware Components



Specific operation range

Range of voltages

- 5.0 ± 0.25 V
- 3.3 ± 0.3
- 2.0 ± 0.2
- 1.5 ± 0.2 V
- Additionally EEPROM/ RS232 C – 12 V

Usual pattern of Power Distribution

- 2 pins of V_{DD} + V_{SS}
- Distributes power to all sections and reduces interference

Other Hardware Components

Separation of Power lines

Ext I/O

Timers

Clock + Reset

Analog to Digital Conveter

- VDD, VSS, AGnd, AREF, AIP

Some Devices use charge pumps – no external power supply

Power & Performance

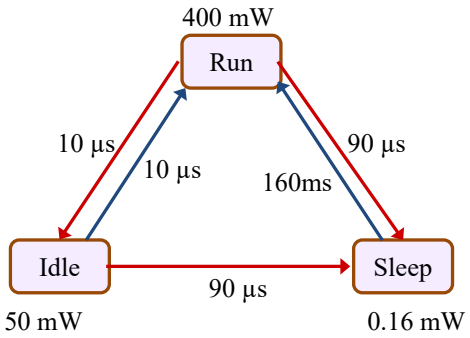
- Propagation delay in gates $\propto 1/V$
- Power consumed in a CMOS circuit $\propto V^2$
- An ES has to perform tasks continuously
- Power saving important
- Wait State/ Stop state
  - 2.5 mW /100 KHz

Power & Performance

SA 1100 – 2 Low Power Modes

- **Run**
  - Normal Consumes Max Power
- **Idle**
  - Saves power by stopping the CPU clock
- **Sleep**
  - Turns off all units except – Power -0
  - Keeps Power unit thro' I/O power supply line
  - Low speed clock available for power manager

Low Power modes



Clock Oscillator Circuits & Clocking Units

Clock source

- Crystal – highest stability – despite temp drifts
- Internal Ceramic Resonator – reasonable stability
- External IC based Oscillator – more driving power required

RTC/Timers/Counters

- At least one counter in every ES

Reset Circuit, Power-up Reset, Watchdog Reset

Activated only for a few clock cycles

System reset – CPU in synch with the reset of rest of the devices

Sources of reset

- Power on reset
- Reset
  - RC
  - IC
- S/w Instruction /Comp Operating Properly/Clk Monitor

Interrupt handler

Most Embedded Systems are Real-Time

No. of Interrupts

- h/w
- event driven

Priority

Latency

Default Priorities

- COP Watchdog
- External Interrupts
- Timer
- Serial I/f
- ADC

ADC,DAC

ADC – multi-channel Vref + , Vref-

DAC- PWM

Software in ES

S/w particular to an application

Processor of ES- handles inst/data

Final stage ⇒ ROM Image

ROM Image

- Boot-up program
- Stack/addr pointers
- Appln tasks
- ISR
- RTOS
- i/p data
- Vector address

Options for Building Embedded Systems

	Implementation	Design Cost	Unit Cost	Upgrades & Bug Fixes	Size	Weight	Power	System Speed
Dedicated Hardware	Discrete Logic	low	mid	hard	large	high	?	very fast
	ASIC	high ($500K / mask set)	very low	hard	tiny - 1 die	very low	low	extremely fast
	Programmable logic – FPGA, PLD	low to mid	mid	easy	small	low	medium to high	very fast
Software Running on Generic Hardware	Microprocessor + memory + peripherals	low to mid	mid	easy	small to med.	low to moderate	medium	moderate
	Microcontroller (int. memory & peripherals)	low	mid to low	easy	small	low	medium	slow to moderate
	Embedded PC	low	high	easy	medium	moderate to high	medium to high	fast

Major Application Areas of Embedded Systems

- **Consumer Electronics**: Camcorders, Cameras etc.
- **Household Appliances**: Television, DVD players, Washing machine, Fridge, Microwave Oven etc.
- **Home Automation and Security Systems**: Air conditioners, sprinklers, Intruder detection alarms, Closed Circuit Television Cameras, Fire alarms etc.
- **Automotive Industry**: Anti-lock breaking systems (ABS), Engine Control, Ignition Systems, Automatic Navigation Systems etc.
- **Telecom**: Cellular Telephones, Telephone switches, Handset Multimedia Applications etc.
- **Computer Peripherals**: Printers, Scanners, Fax machines etc.
- **Computer Networking Systems**: Network Routers, Switches, Hubs, Firewalls etc.
- **Health Care**: Different Kinds of Scanners, EEG, ECG Machines etc.
- **Measurement & Instrumentation**: Digital multimeters, Digital CROs, Logic Analyzers PLC systems etc.
- **Banking & Retail**: Automatic Teller Machines (ATM) and Currency counters, Point of Sales (POS)
- **Card Readers**: Barcode, Smart Card Readers, Handheld Devices etc.

Prof. Manoj S Kakade

Email id: manoj.kakade@pilani.bits-pilani.ac.in

THANK YOU!!!!!

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE

BITS Pilani
Pilani Campus

ESZG512/MELZG526/SEZG516
Embedded System Design
Contact Session 2

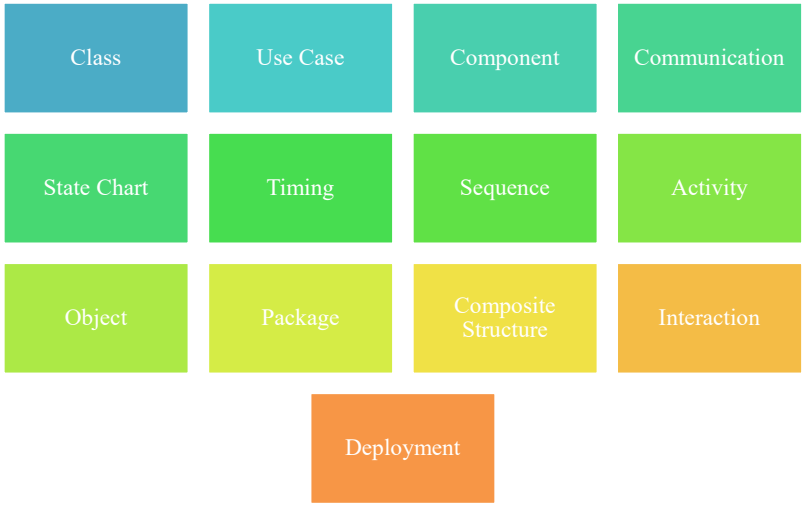
CS2 : (Overview of UML)

2	➢ Overview of UML

UML - Introduction

- Modeling System Behavior & Relationships
- Static Behavior
  - System Interaction with user – starting point
- Dynamic Behavior
  - System behavior whiles task execute
  - Interaction between threads

Common UML Diagram



Time, Performance and Quality of Service

- In 2002, Object Management Group (OMG) adopted the Unified Modeling Language™ (UML) profile
- Real-Time Profile (RTP): standardized means for specifying timeliness, performance, and schedulability aspects of systems and parts of systems
- Profiles provide a minutely specialized form of UML
- Modeling Actions and Concurrency
- Modeling Resources
- Modeling Time
- Modeling Schedulability
- Modeling Performance

Source:Bruce Powel Douglass, "Real-Time UML"

Overview of UML Diagram

Diagram Name	Used to...	Primary Phase
Structure Diagrams		
Class	Illustrate the relationships between classes modeled in the system	Analysis, Design
Object	Illustrate the relationships between objects modeled in the system; used when actual instances of the classes will better communicate the model	Analysis, Design
Package	Group other UML elements together to form higher-level constructs; implementation	Analysis, Design

Overview of UML Diagram

Diagram Name	Used to...	Primary Phase
Structure Diagrams		
Deployment	Show the physical architecture of the system; can also be used to show software components being deployed onto the physical architecture	Physical Design, Implementation
Component	Illustrate the physical relationships among the software components; implementation	Physical Design
Composite Structure Design	Illustrate the internal structure of a class, i.e., the relationships among the parts of a class	Analysis

Overview of UML Diagram

Diagram Name	Used to...	Primary Phase
Behavioral Diagrams		
Activity	Illustrate business workflows independent of classes, the flow of activities in a use case, or detailed design of a method	Analysis, Design
Sequence	Model the behavior of objects within a use case; focuses on the time-based ordering of an activity	Analysis, Design
Communication	Model the behavior of objects within a use case; focus on the communication among a set of collaborating objects of an activity	Analysis, Design

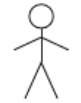
Overview of UML Diagram

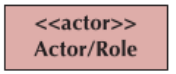
Diagram Name	Used to...	Primary Phase
Behavioral Diagrams		
Interaction Overview	Illustrate an overview of the flow of control of a process	Analysis, Design
Timing	Illustrate the interaction among a set of objects and the state changes they go through along a time axis	Analysis, Design
Behavioral State Machine	Examine the behavior of one class	Analysis, Design
Protocol State Machine	Illustrate the dependencies among the different interfaces of a class	Analysis, Design
Use-Case	Capture business requirements for the system and illustrate the interaction between the system and its environment.	Analysis

Syntax for Use-Case Diagram

- **A use case:**
- Represents a major piece of system functionality.
- Can extend another use case.
- Can include another use case.
- Is placed inside the system boundary.
- Is labeled with a descriptive verb–noun phrase.

Syntax for Use-Case Diagram

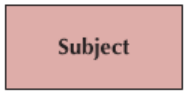




An actor:

- Is a person or system that derives benefit from and is external to the subject.
- Is depicted as either a stick figure (default) or, if a non human actor is involved, as a rectangle with <<actor>> in it (alternative).
- It is labelled with its role.
- Can be associated with other actors using a specialization/superclass association, denoted by an arrow with a hollow arrowhead.
- It is placed outside the subject boundary.

Syntax for Use-Case Diagram



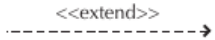
- **A subject boundary:**
- Includes the name of the subject inside or on top.
- Represents the scope of the subject, e.g., a system or an individual business process.

- **An association relationship:**
- Links an actor with the use case(s) with which it interacts.

Syntax for Use-Case Diagram

An include relationship:

- Represents the inclusion of the functionality of one use case within another.
- Has an arrow drawn from the base use case to the used use case.



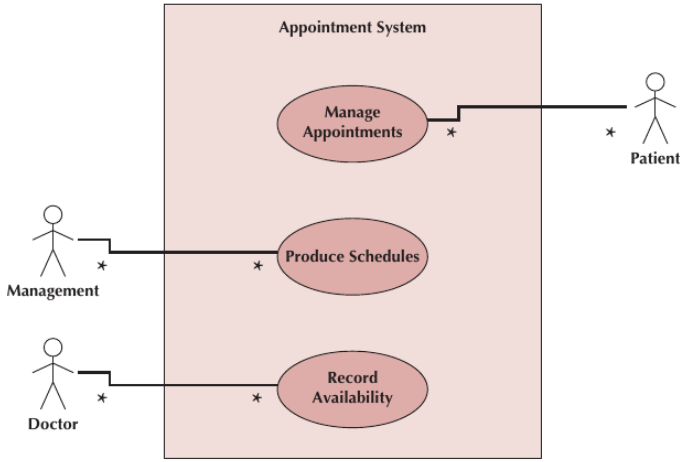
An extend relationship:

- Represents the extension of the use case to include optional behavior.
- Has an arrow drawn from the extension use case to the base use case.

A generalization relationship:

- Represents a specialized use case to a more generalized one.
- Has an arrow drawn from the specialized use case to the base use case.

Use-Case Diagram for the Appointment System



Syntax for an Activity Diagram

An action:

- Is a simple, non decomposable piece of behavior.
- Is labeled by its name.

An activity:

- Is used to represent a set of actions.
- Is labeled by its name.

An object node:

- Is used to represent an object that is connected to a set of object flows.
- Is labeled by its class name.

Syntax for an Activity Diagram

A control flow:

- Shows the sequence of execution.

An object flow:

- Shows the flow of an object from one activity (or action) to another activity (or action).

An initial node:

- Portrays the beginning of a set of actions or activities.

A final-activity node:

- Is used to stop all control flows and object flows in an activity (or action).

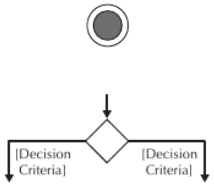
Syntax for an Activity Diagram

A final-flow node:

- Is used to stop a specific control flow or object flow.

A decision node:

- Is used to represent a test condition to ensure that the control flow or object flow only goes down one path.
- Is labeled with the decision criteria to continue down the specific path.



A merge node:

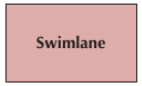
- Is used to bring back together different decision paths that were created using a decision node.

Syntax for an Activity Diagram

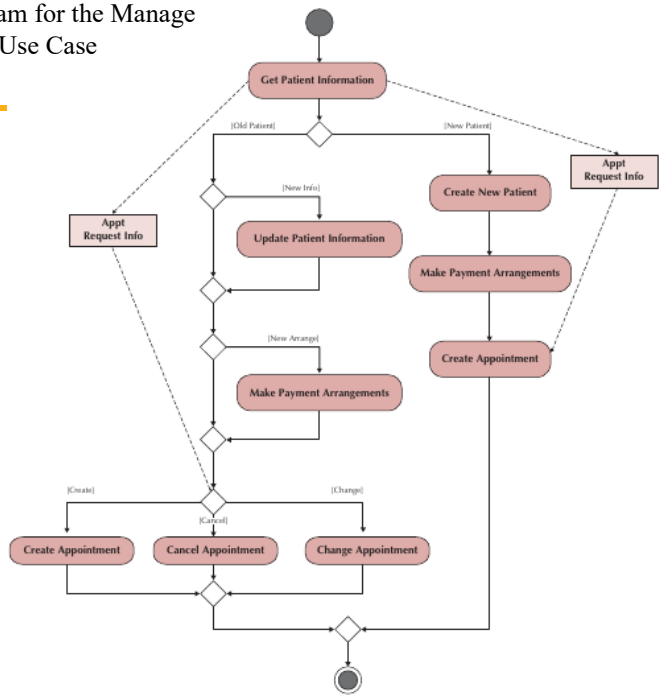
- **A fork node:**
- Is used to split behavior into a set of parallel or concurrent flows of activities (or actions)

- **A join node:**
- Is used to bring back together a set of parallel or concurrent flows of activities (or actions)

- **A swimlane:**
- Is used to break up an activity diagram into rows and columns to assign the individual activities (or actions) to the individuals or objects that are responsible for executing the activity (or action)
- Is labeled with the name of the individual or object responsible



Activity Diagram for the Manage Appointments Use Case



Behavioral State Machine Diagram Syntax

A state:

- Is shown as a rectangle with rounded corners.
- Has a name that represents the state of an object.

An initial state:

- Is shown as a small, filled-in circle.
- Represents the point at which an object begins to exist.

A final state:

- Is shown as a circle surrounding a small, filled-in circle (bull's-eye).
- Represents the completion of activity.

Behavioral State Machine Diagram Syntax

An event:

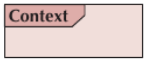
anEvent

- Is a noteworthy occurrence that triggers a change in state.
- Can be a designated condition becoming true, the receipt of an explicit signal from one object to another, or the passage of a designated period of time.
- Is used to label a transition.

A transition:

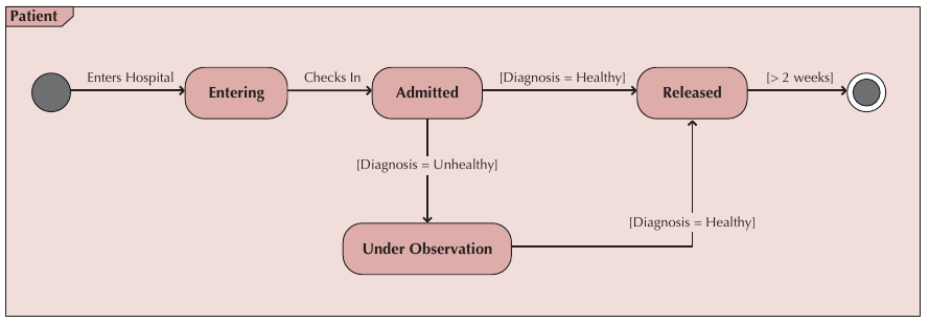
- Indicates that an object in the first state will enter the second state.
- Is triggered by the occurrence of the event labeling the transition.
- Is shown as a solid arrow from one state to another, labeled by the event name.

A frame:

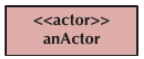


- Indicates the context of the behavioral state machine.

Behavioral State Machine Diagram



Sequence Diagram Syntax



An actor:

- Is a person or system that derives benefit from and is external to the system.
- Participates in a sequence by sending and/or receiving messages.
- Is placed across the top of the diagram.
- Is depicted either as a stick figure (default) or, if a nonhuman actor is involved, as a rectangle with <<actor>> in it (alternative).

An object:

- Participates in a sequence by sending and/or receiving messages.
- Is placed across the top of the diagram.

Sequence Diagram Syntax

A lifeline:

- Denotes the life of an object during a sequence.
- Contains an X at the point at which the class no longer interacts.

An execution occurrence:

- Is a long narrow rectangle placed atop a lifeline.
- Denotes when an object is sending or receiving messages.

A message:

- Conveys information from one object to another one.
- A operation call is labeled with the message being sent and a solid arrow, whereas a return is labeled with the value being returned and shown as a dashed arrow.

Sequence Diagram Syntax

A guard condition:

- Represents a test that must be met for the message to be sent.

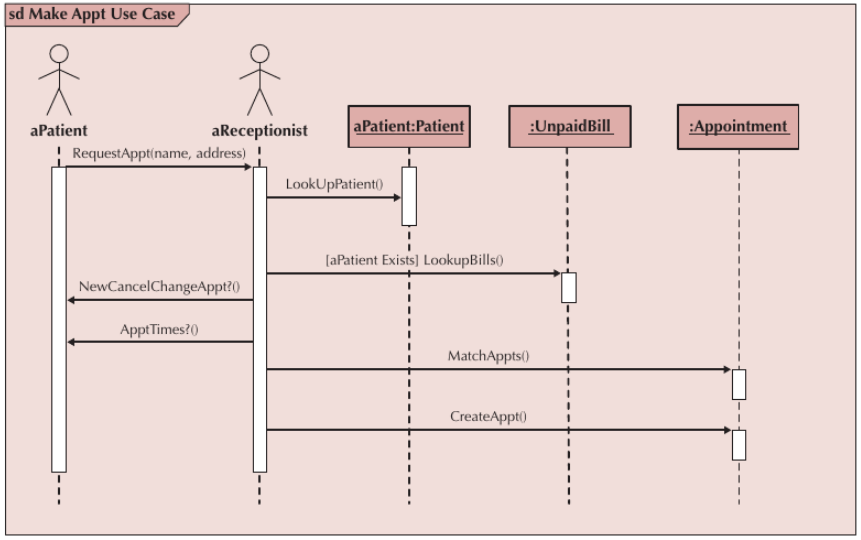
For object destruction:

- An X is placed at the end of an object's lifeline to show that it is going out of existence.

A frame:

- Indicates the context of the sequence diagram.

Sequence Diagram Example

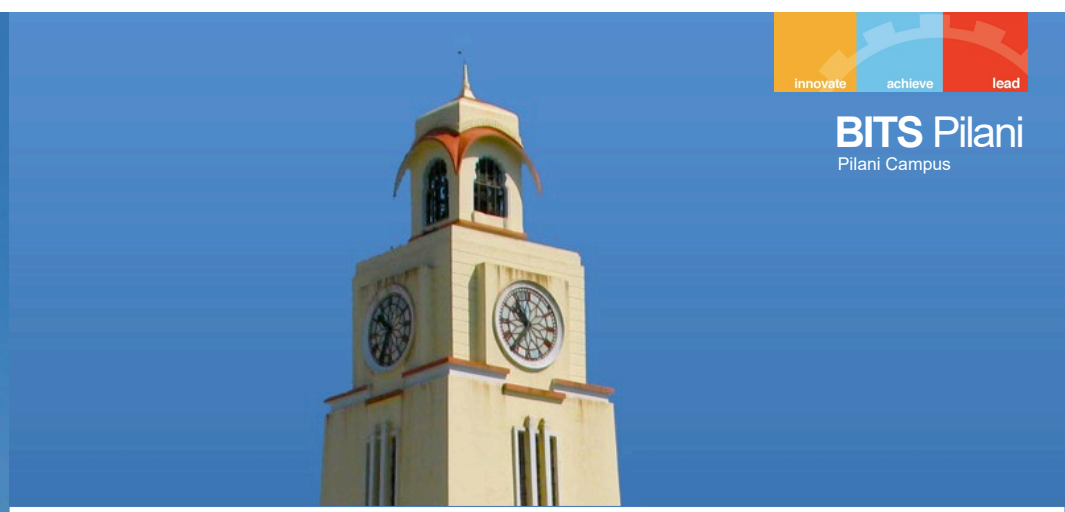


Thank you

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE



innovate achieve lead

BITS Pilani
Pilani Campus

ESZG512 / MELZG526 / SEZG516
Embedded System Design
Contact Session 3

innovate achieve lead

CS3 : (Introduction to ARMv4 Architecture)

1	➢ ARM CPU Architecture ➢ Programmers Model for ARM CPU ➢ Operating Modes , ISA ➢ ARM Exception Handling ➢ Pipelining

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

innovate achieve lead

ARM history

- 1983 developed by Acorn computers
  - To replace 6502 in BBC computers
  - 4-men VLSI design team
  - It simplicity came from the inexperience team
  - Match the needs for generalized SoC for reasonable power, performance and die size
- By 1985, design of first commercial RISC machine called Acron RISC Machine (ARM).
- In 1990, there were 12 engineers and 1 CEO, with no customers and a little money.
- 1990 ARM (Advanced RISC Machine), owned by Acorn, Apple and VLSI.
- In 1990's TI incorporated ARM for mobile phones

ARM history

- By 1998 there were 13 millionaires in company.
- Company headquarters in Cambridge, UK
  - Processor design centers in Cambridge, Austin, and Sophia Antipolis
  - Sales, support, and engineering offices all over the world
- Best known for its range of RISC processor cores designs
  - Other products – fabric IP, software tools, models, cell libraries - to help partners develop and ship ARM-based SoCs
- ARM does not manufacture silicon
- ARM's partners shipped around 15 billion ARM-based chips in 2015.
- In July 2016 SoftBank + ARM £24.4bn investment in future technology
  - Two companies with a shared vision of pushing the limits of technology coming together to accelerate ARM's leadership in technologies for the future

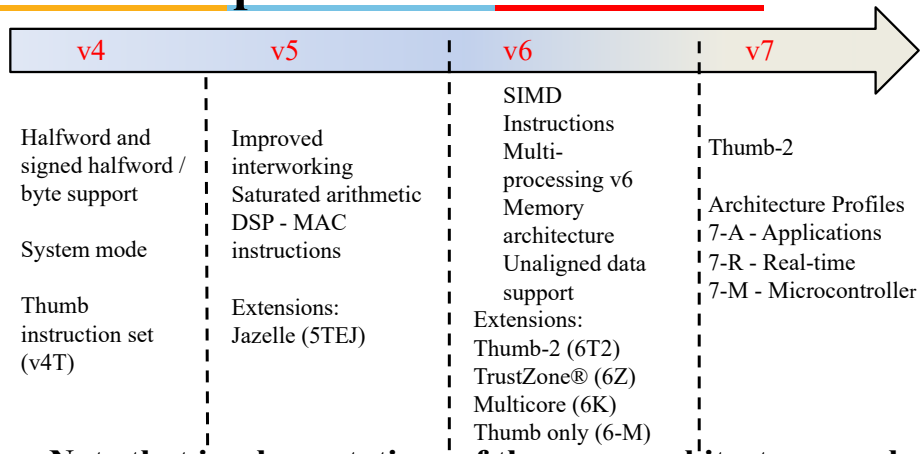
Why ARM?

- One of the most licensed and thus widespread processor cores in the world. 1,348 Cumulative licenses signed.
- ARM's partners shipped around 15 billion ARM-based chips in 2015.
- Today ARM-based application processors can be found in about an 85% share of mobile devices, including smartphones, tablets and laptops.
- ARM filed an additional 242 patents in 2015, taking the total number of patents owned or pending to more than 4,500.

ARM Processor Versions

Revision	Example core implementation	ISA enhancement
ARMv1	ARM1	First ARM processor 26-bit addressing
ARMv2	ARM2	32-bit multiplier 32-bit coprocessor support
ARMv2a	ARM3	On-chip cache Atomic swap instruction Coprocessor 15 for cache management
ARMv3	ARM6 and ARM7DI	32-bit addressing Separate *cpsr* and *spsr* New modes—*undefined instruction* and *abort* MMU support—virtual memory
ARMv3M	ARM7M	Signed and unsigned long multiply instructions
ARMv4	StrongARM	Load-store instructions for signed and unsigned halfwords/bytes New mode—*system* Reserve SWI space for architecturally defined operations 26-bit addressing mode no longer supported
ARMv4T	ARM7TDMI and ARM9T	Thumb
ARMv5TE	ARM9E and ARM10E	Superset of the ARMv4T Extra instructions added for changing state between ARM and Thumb Enhanced multiply instructions Extra DSP-type instructions Faster multiply accumulate
ARMv5TEJ	ARM7EJ and ARM926EJ	Java acceleration
ARMv6	ARM11	Improved multiprocessor instructions Unaligned and mixed endian data handling New multimedia instructions

Development of the ARM Architecture



- **Note that implementations of the same architecture can be different**
  - Cortex-A8 - architecture v7-A, with a 13-stage pipeline
  - Cortex-A9 - architecture v7-A, with an 8-stage pipeline

Arm Processor Families

- Cortex-A series (Application)
  - High performance processors capable of full operating system (OS) support
  - Applications include smartphones, digital TV, smart books

Cortex-A: Cortex-A5, Cortex-A7, Cortex-A8, Cortex-A9, Cortex-A15, Cortex-A17, Cortex-A32, Cortex-A34, Cortex-A35, Cortex-A53, Cortex-A55, Cortex-A57, Cortex-A65, Cortex-A65AE, Cortex-A72, Cortex-A73, Cortex-A75, Cortex-A76, Cortex-A75AE, Cortex-A77

- Cortex-R series (Real-time)
  - High performance and reliability for real-time applications
  - Applications include automotive braking systems, powertrains

Cortex-R: Cortex-R4, Cortex-R5, Cortex-R7, Cortex-R8, Cortex-R52

- Cortex-M series (Microcontroller)
  - Cost sensitive solutions for deterministic microcontroller applications
  - Applications include microcontrollers, smart sensors
  - SecurCore series for high security applications

Cortex-M: Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, Cortex-M4, Cortex-M7, Cortex-M23, Cortex-M33, Cortex-m35P

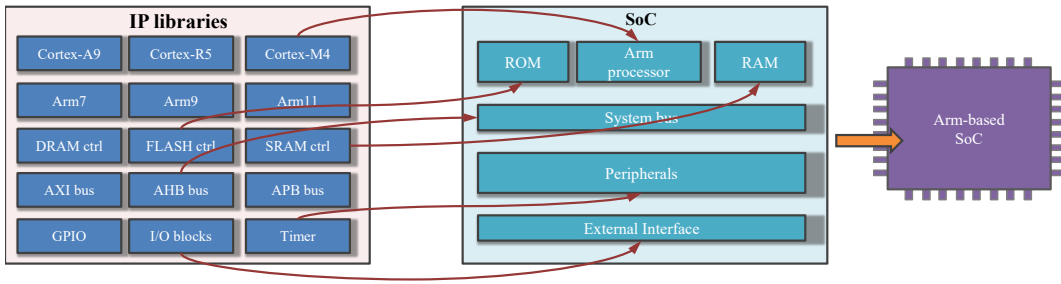
SecurCore: SC000 Processor, SC300 Processor

- Earlier classic processors including Arm7, Arm9, Arm11 families

Classic: Arm7, Arm9, Arm11

How to Design an Arm-based SoC

1. Select a set of IP cores from Arm and/or other third-party IP vendors
2. Integrate IP cores into a single-chip design
3. Give design to semiconductor foundries for chip fabrication



Licensable IPs | SoC Design | Chip Manufacture

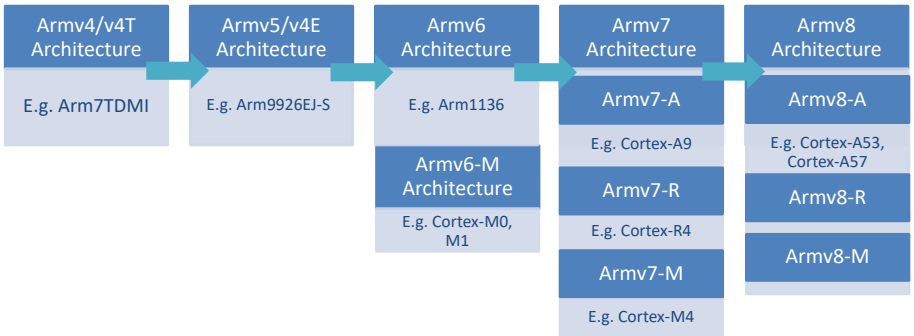
Arm Processors vs. Arm Architectures

Arm architecture

- Describes the details of instruction set, programmer's model, exception model, and memory map
- Documented in the 'Architecture Reference Manual'

Arm processor

- Developed using one of the Arm architectures
- More implementation details, such as timing information
- Documented in processor's 'Technical Reference Manual'



The ARM Processor

- Controls the embedded device.
- Different versions of the ARM processor are available to suit the desired operating characteristics.

- **An ARM Processor**

1. Core

(the execution engine that processes instructions and manipulates data)

+

2. Surrounding Components

(that interface it with a bus & include memory management and caches)

Programmer's Model

It contains the following :

- Registers
- Processor operating states
- Operating modes
- Memory formats
- Data types
- The program status registers
- Exceptions
- Interrupt latencies
- Reset

Data Sizes and Instruction Sets

- ARM is a 32-bit load / store RISC architecture
  - The only memory accesses allowed are loads and stores
  - Most internal registers are 32 bits wide
  - Most instructions execute in a single cycle
- When used in relation to ARM cores
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)
  - **Doubleword** means 64 bits (eight bytes)
- ARM cores implement two basic instruction sets
  - **ARM** instruction set – instructions are all 32 bits long
  - **Thumb** instruction set – instructions are a mix of 16 and 32 bits
    - Thumb-2 technology added many extra 32 and 16-bit instructions to the original 16-bit Thumb instruction set

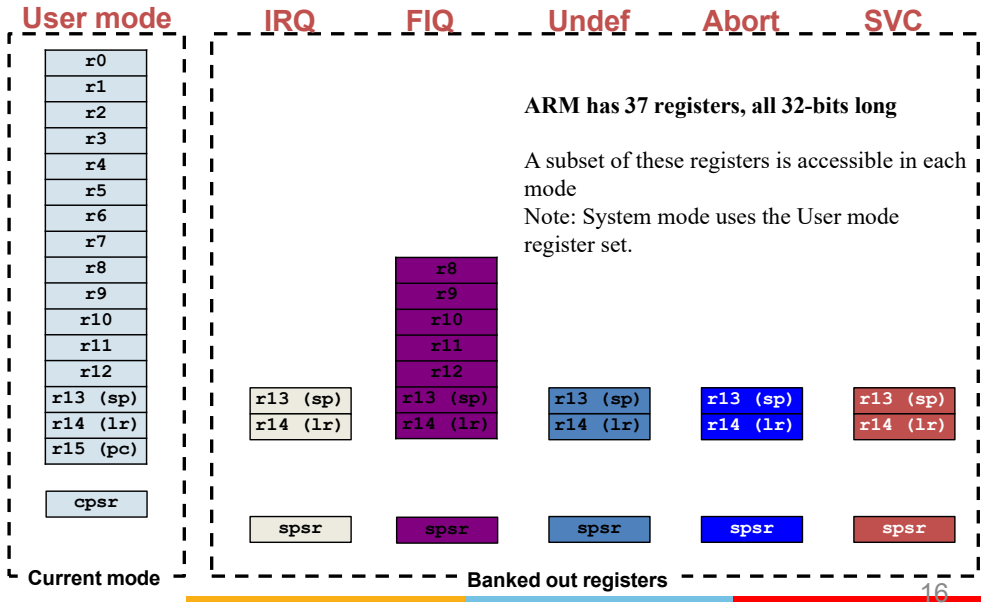
Processor Modes

- ARM has seven basic operating modes
  - Each mode has access to its own stack space and a different subset of registers
  - Some operations can only be carried out in a privileged mode

Mode	Description	
Supervisor (SVC)	Entered on reset and when a Supervisor call instruction (SVC) is executed	Privileged modes
FIQ	Entered when a high priority (fast) interrupt is raised	
IRQ	Entered when a normal priority interrupt is raised	
Abort	Used to handle memory access violations	
Undef	Used to handle undefined instructions	
System	Privileged mode using the same registers as User mode	
User	Mode under which most Applications / OS tasks run	Unprivileged mode

Exception modes

The ARM Register Set



The ARM Register Set

- They are of two types-
  - General purpose
  - Special Purpose
- The ARM7TDMI processor has a total of 37 registers:
  - 31 general-purpose 32-bit registers
  - 6 status registers
- These registers are not all accessible at the same time

The ARM Register Set

General Purpose Registers

1.Hold either data or an address.

2.Identified with the letter r prefixed to the register number. (register 4 = r4).

3.All the registers are 32 bits in size.

4.Up to 18 active registers: 16 data registers and 2 processor status registers.

5.Data registers are visible to the programmer (r0 to r15).

The ARM Register Set

6. Three special function registers : r13, r14, and r15. They are frequently given different labels to differentiate them from the other registers.

- Register r13 = Stack Pointer (SP) : stores the head of the stack in the current processor mode.
- Register r14 = Link register (LR) : whenever the core calls a subroutine, it puts the return address in this register.
- Register r15 = Program Counter (PC): contains the address of the next instruction to be fetched by the processor.

Registers r13 and r14 can also be used as general-purpose registers

The ARM Register Set

7. Registers r0 to r13 are orthogonal—any instruction that you can apply to r0 you can equally well apply to any of the other registers.
8. There are instructions that treat r14 and r15 in a special way.
9. The register file contains all the registers available to a programmer. Which registers are visible to the programmer depend upon the current mode of the processor.

Program Status Registers

'cpsr and spsr'

(the current and saved program status registers)

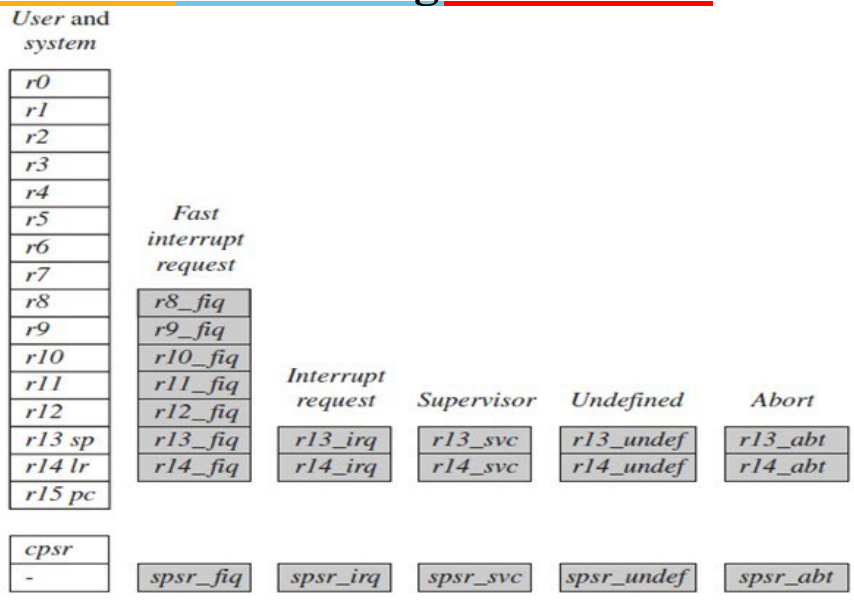
Unbanked Registers

- r0 to r7 are unbanked registers- Means in all processing modes they are representing same 32 bit physical register.
- They are completely general-purpose registers.
- No special use.

Banked Registers (r8 to r14)

- 20 registers are hidden from a program at different times. These registers are called banked registers
- They are available only when the processor is in a particular mode
- For example, abort mode has banked registers r13_abt, r14_abt and spsr_abt.

Banked Registers & States…



Banked Registers…

- All processor modes except system mode have a set of associated banked registers
- If you change processor mode, a banked register from the new mode will replace an existing register.
- For example, when the processor is in the interrupt request mode, the instructions you execute still access registers named r13 and r14. However, these registers are the banked registers r13_irq and r14_irq.
- The user mode registers r13_usr and r14_usr are not affected by the instruction referencing these registers.

Banked Registers…

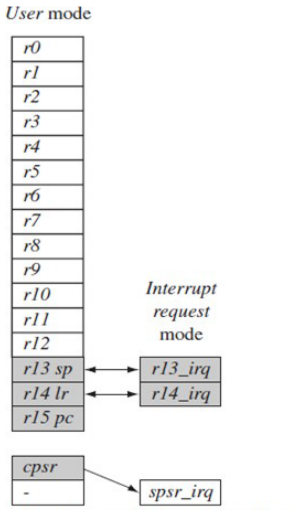
The processor mode can be changed by
1.A program that writes directly to the cpsr
2.Hardware when the core responds to an exception or interrupt

The following exceptions and interrupts cause a mode change:

- Reset
- Interrupt request
- Fast interrupt request
- Software interrupt
- Data abort
- Prefetch abort
- Undefined instruction

Banked Registers…

- Figure illustrates what happens when an interrupt forces a mode change



Changing mode on an exception.

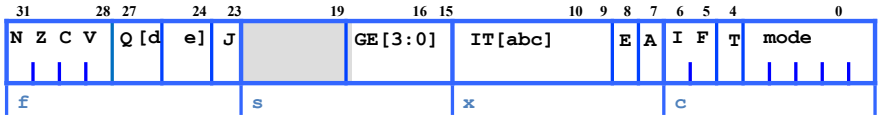
Banked Registers…

- User registers r13 and r14 to be banked
- The user registers are replaced with registers r13_irq and r14_irq
- Respectively r14_irq contains the return address and r13_irq contains the stack pointer for interrupt request mode.
- The saved program statusregister (spsr), stores the previous mode's cpsr.

Banked Registers…

- To return back to user mode, a special return instruction is used that instructs the core to restore the original cpsr from the spsr_irq and bank in the user registers r13 and r14.
- Another important feature to note is that the cpsr is not copied into the spsr when a mode change is forced due to a program writing directly to the cpsr.
- The saving of the cpsr only occurs when an exception or interrupt is raised.

Program Status Registers

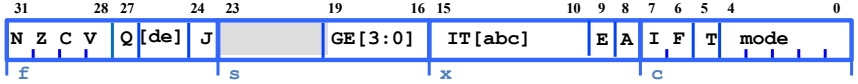


Four fields (each 8 bits wide)

1. ***Flags field :-*** **the condition flags**
2. ***Status field :-***
3. ***Extension field :-***
4. ***Control field :-*** **the processor mode, state, and interrupt mask bits.**

•Shaded parts: Reserved for future expansion.

Program Status Registers



- **Condition code flags**
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed
- **Sticky Overflow flag - Q flag**
  - Indicates if saturation has occurred
- **SIMD Condition code bits – GE[3:0]**
  - Used by some SIMD instructions
- **IF THEN status bits – IT[abcde]**
  - Controls conditional execution of Thumb instructions
- **T bit**
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state
- **J bit**
  - J = 1: Processor in Jazelle state
- **Mode bits**
  - Specify the processor mode
- **Interrupt Disable bits**
  - I = 1: Disables IRQ
  - F = 1: Disables FIQ
- **E bit**
  - E = 0: Data load/store is little endian
  - E = 1: Data load/store is bigendian
- **A bit**
  - A = 1: Disable imprecise data aborts

Current Program Status Register… (Bits 0-4)

Processor mode.

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast interrupt request	fiq	yes	10001
Interrupt request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

Processor Modes

- **Determines**
  1. Which registers are active
  2. The access rights to the ***CPSR*** register itself
- **Processor modes**
  - **Privileged**
  - **Non-privileged**

❖ **Privileged mode: Allows full read-write access to the *CPSR***

❖ **Non-privileged mode : *allows only read operation* access to the control field in the *CPSR* *but still allows read-write access to the condition flags.***

Processor Modes

- **It Determines** which registers are active and the access rights to the ***CPSR*** register itself .
- Total there are Seven Processor Modes
- **Six privileged modes**
  - **(*abort, fast interrupt request, interrupt request, supervisor, system, undefined*)**
- ***One Non-privileged mode***
  - **(*user*)**

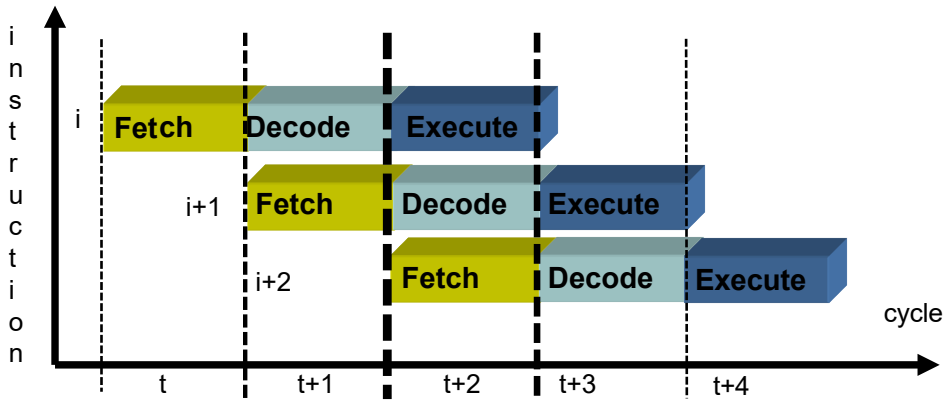
Processor Modes

The ARM has seven basic operating modes:-

- **User** : unprivileged mode under which most tasks run
- **FIQ** : entered when a high priority (fast) interrupt is raised
- **IRQ** : entered when a low priority (normal) interrupt is raised
- **Supervisor**: enteredon reset and when a Software Interrupt instruction is executed
- **Abort** : used to handle memory access violations
- **Undef** : used to handle undefined instructions
- **System**: privileged mode using the same registers as user mode

Pipeline

- 3-stage pipeline: Fetch – Decode - Execute
- Three-cycle latency, one instruction per cycle throughput



Pipeline

- **Fetch :** Loads an instruction from memory
- **Decode:** The instruction is decoded and the datapath control signals prepared for the next cycle
- **Execute :** The operands are read from the register bank, shifted, combined in the ALU and the result written back.
- As the pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency.
- System latency also increases because it takes more cycles to fill the pipeline before the core can execute an instruction.

Processor operating states

The ARM7TDMI processor has two operating states:

➢**ARM :-**32-bit, word-aligned ARM instructions are executed in this state.

➢**Thumb**:-16-bit, halfword-aligned Thumb instructions are executed in this state.

Processor operating states

Switching States :-

- The operating state of the ARM7TDMI core can be switched between ARM state and Thumb state using the BX instruction.
  - ➢All exception handling is entered in ARM state. If an exception occurs in Thumb state, the processor reverts to ARM state.
  - ➢The transition back to Thumb state occurs automatically on return.
  - ➢An exception handler can change to Thumb state but it must return to ARM state to allow the exception handler to terminate correctly.

The Thumb-state register set

The Thumb-state register set is a subset of the ARM-state set.
The programmer has access to:

- 8 general registers, r0–r7
- the PC
- the SP
- the LR
- the CPSR.

The Thumb-state register set

Thumb-state general registers and program counter

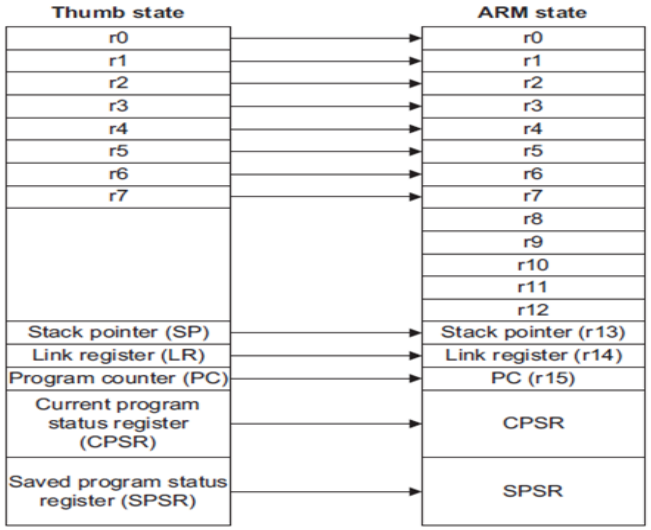
System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
SP	SP_fiq	SP_svc	SP_abt	SP_irq	SP_und
LR	LR_fiq	LR_svc	LR_abt	LR_irq	LR_und
PC	PC	PC	PC	PC	PC

Thumb-state program status registers

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

= banked register

The relationship between ARM-state and Thumb-state registers



Mapping of Thumb-state registers onto ARM-state registers

Memory formats

The ARM7TDMI processor views memory as a linear collection of bytes numbered in ascending order from zero.

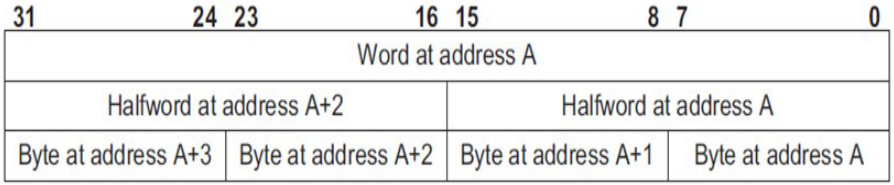
For example:

- bytes zero to three hold the first stored word
- bytes four to seven hold the second stored word.

The ARM7TDMI processor is bi-endian and can treat words in memory as being stored in either:

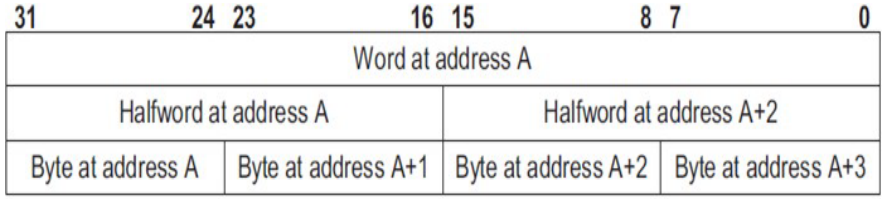
- Little-endian
- Big-Endian

Memory formats :- Little-endian



LIttle-endian addresses of bytes and halfwords within words

Memory formats :- Big-endian



Big-endian addresses of bytes and halfwords within words

Data types

The ARM7TDMI processor supports the following data types:

- words, 32-bit
- half words, 16-bit
- bytes, 8-bit.

You must align these as follows:

- word quantities must be aligned to four-byte boundaries
- half-word quantities must be aligned to two-byte boundaries
- byte quantities can be placed on any byte boundary.

Exceptions

- Exceptions arise whenever the normal flow of a program has to be halted temporarily, for example, to service an interrupt from a peripheral.
- Before attempting to handle an exception, the ARM7TDMI processor preserves the current processor state so that the original program can resume when the handler routine has finished.

Exception Handling

When an exception occurs, the core…

- Copies CPSR into SPSR_<mode>
- Sets appropriate CPSR bits
  - Change to ARM state (if appropriate)
  - Change to exception mode
  - Disable interrupts (if appropriate)
- Stores the return address in LR_<mode>
- Sets PC to vector address

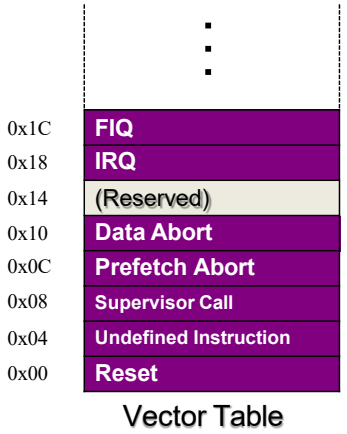
To return, exception handler needs to…

- Restore CPSR from SPSR_<mode>
- Restore PC from LR_<mode>

Cores can enter ARM state or Thumb state when taking an exception

- Controlled through settings in CP15

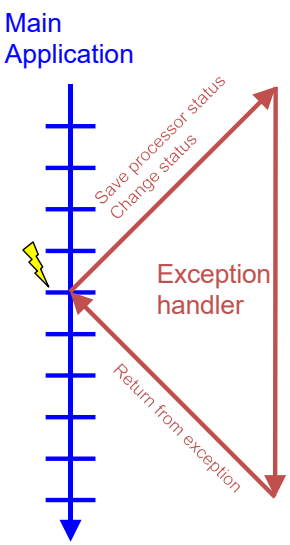
Note that v7-M and v6-M exception model is different



Vector Table

Vector table can also be at **0xFFFF0000** on most cores

Exception handling process



1. **Save processor status**
   - Copies CPSR into SPSR_<mode>
   - Stores the return address in LR_<mode>
   - Adjusts LR based on exception type
2. **Change processor status for exception**
   - Mode field bits
   - ARM or Thumb state
   - Interrupt disable bits (if appropriate)
   - Sets PC to vector address
3. **Execute exception handler**
   - <users code>
4. **Return to main application**
   - Restore CPSR from SPSR_<mode>
   - Restore PC from LR_<mode>

1 and 2 performed automatically by the core

3 and 4 responsibility of software

Priorities of Exceptions

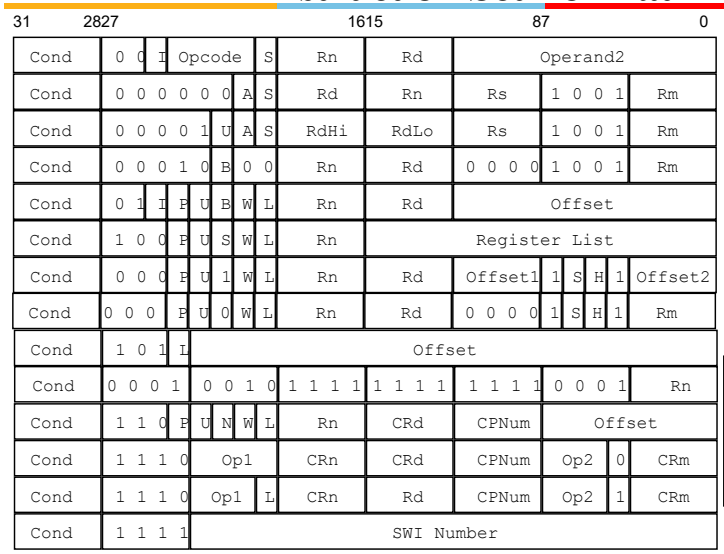
1. Reset (highest priority).
2. Data Abort.
3. FIQ.
4. IRQ.
5. Prefetch Abort.
6. Undefined instruction.
7. SWI (lowest priority) - Software Interrupt - is used to enter Supervisor mode, usually to request a particular supervisor function.

Vector Summary

Address	Exception	Mode on entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	-- reserved --	--
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

These are byte addresses, and will normally contain a branch instruction pointing to the relevant routine. The FIQ routine might reside at 0x1C onwards, and thereby avoid the need for (and execution time of) a branch instruction.

ARM Instruction Set Format



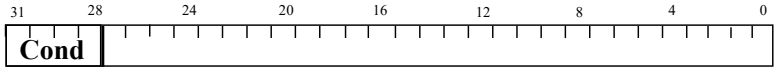
Instruction type

- Data processing / PSR Transfer
- Multiply
- Long Multiply (v3M / v4 only)
- Swap
- Load/Store Byte/Word
- Load/Store Multiple
- Halfword transfer : Immediate offset (v4 only)
- Halfword transfer: Register offset (v4 only)
- Branch
- Branch Exchange (v4T only)
- Coprocessor data transfer
- Coprocessor data operation
- Coprocessor register transfer
- Software interrupt

Conditional Execution

- Most instruction sets only allow branches to be executed conditionally.
- However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.
  - All instructions contain a condition field which determines whether the CPU will execute them.
  - Non-executed instructions soak up 1 cycle.
    - Still have to complete cycle so as to allow fetching and decoding of following instructions.
- This removes the need for many branches, which stall the pipeline (3 cycles to refill).
  - Allows very dense in-line code, without branches.
  - The Time penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed.

The Condition Field



- 0000 = EQ - Z set (equal)
- 0001 = NE - Z clear (not equal)
- 0010 = HS / CS - C set (unsigned higher or same)
- 0011 = LO / CC - C clear (unsigned lower)
- 0100 = MI -N set (negative)
- 0101 = PL - N clear (positive or zero)
- 0110 = VS - V set (overflow)
- 0111 = VC - V clear (no overflow)
- 1000 = HI - C set and Z clear (unsigned higher)
- 1001 = LS - C clear or Z (set unsigned lower or same)
- 1010 = GE - N set and V set, or N clear and V clear (>or =)
- 1011 = LT - N set and V clear, or N clear and V set (>)
- 1100 = GT - Z clear, and either N set and V set, or N clear and V set (>)
- 1101 = LE - Z set, or N set and V clear,or N clear and V set (<, or =)
- 1110 = AL - always
- 1111 = NV - reserved.

Prof. Manoj S Kakade

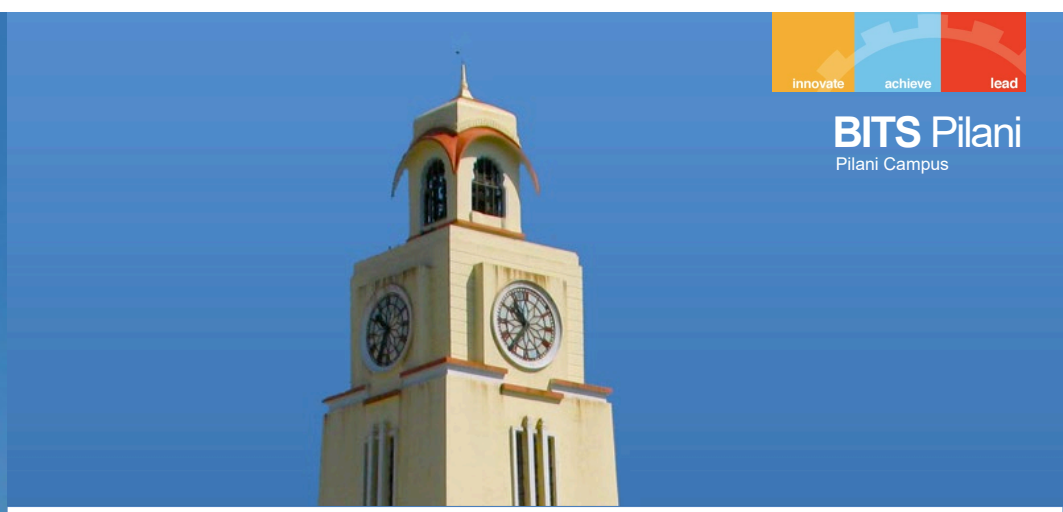
Email id: manoj.kakade@pilani.bits-pilani.ac.in

THANK YOU!!!!!

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE



innovate achieve lead

BITS Pilani
Pilani Campus

ESZG512/MELZG526/SEZG516
Embedded System Design
Contact Session 4

innovate achieve lead

CS5 : (Introduction to ARMv7M Architecture)

- **Introduction to Cortex M4 Architecture**

innovate achieve lead

Introduction to ARM Cortex M4 Architecture

- Cortex-M4 processors have:
- Three-stage pipeline design
- Harvard bus architecture with unified memory space: instructions and data use the same address space
- 32-bit addressing, supporting 4GB of memory space
- On-chip bus interfaces based on ARM AMBA (Advanced Microcontroller Bus Architecture) Technology, which allow pipelined bus operations for higher throughput
- An interrupt controller called NVIC (Nested Vectored Interrupt Controller) supporting up to 240 interrupt requests and from 8 to 256 interrupt priority levels (dependent on the actual device implementation)

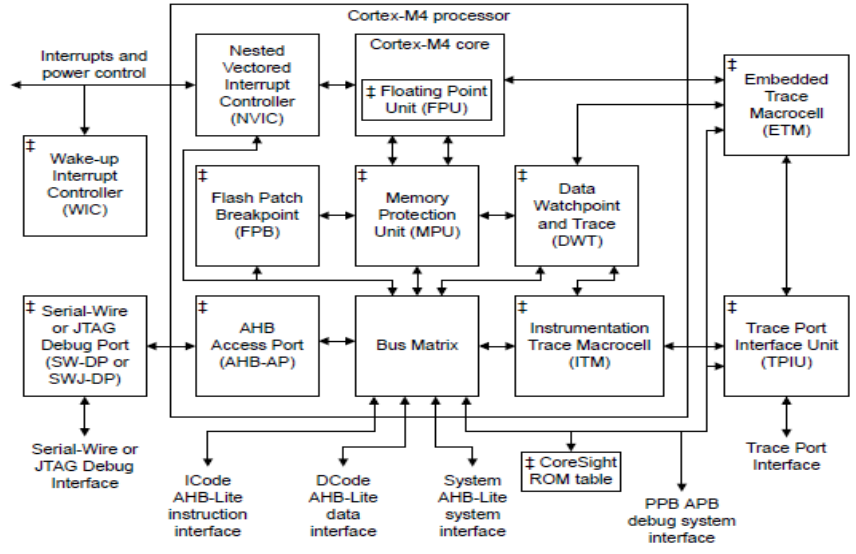
Introduction to ARM Cortex M4 Architecture

- Cortex-M4 processors have:
- Support for various features for OS (Operating System) implementation such as a system tick timer, shadowed stack pointer.
- Sleep mode support and various low power features.
- Support for an optional MPU (Memory Protection Unit) to provide memory protection features like programmable memory, or access permission control.
- The option of being used in single processor or multi-processor designs.

Introduction to ARM Cortex M4 Architecture

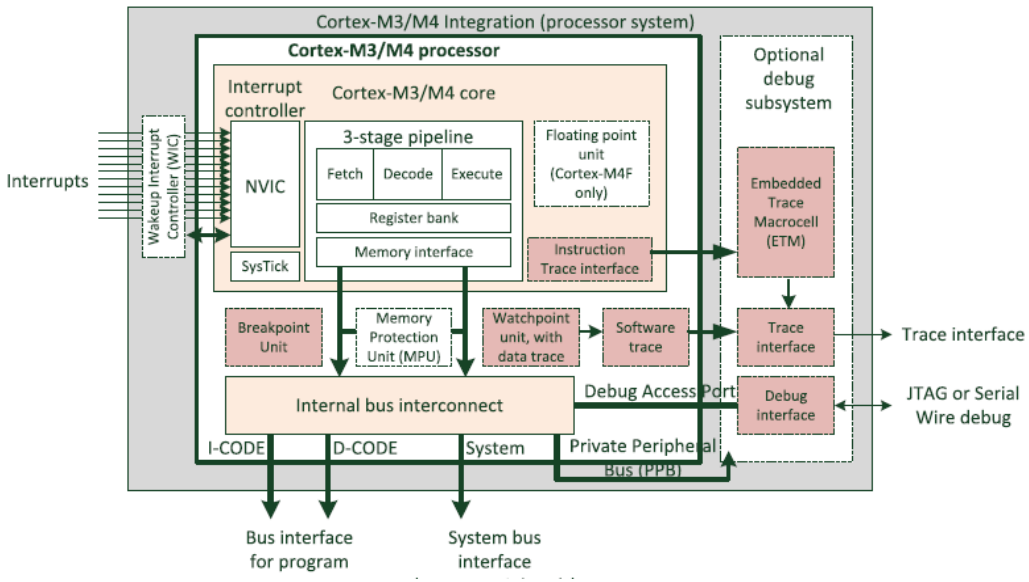
- In addition, the Cortex-M4 processor also supports:
- Single Instruction Multiple Data (SIMD) operations
- Additional fast MAC and multiply instructions
- Saturating arithmetic instructions
- Optional floating point instructions (single precision)

ARM Cortex M4 Architecture



Block diagram showing the structure of the Cortex-M4 processor.

ARM Cortex M4 Architecture



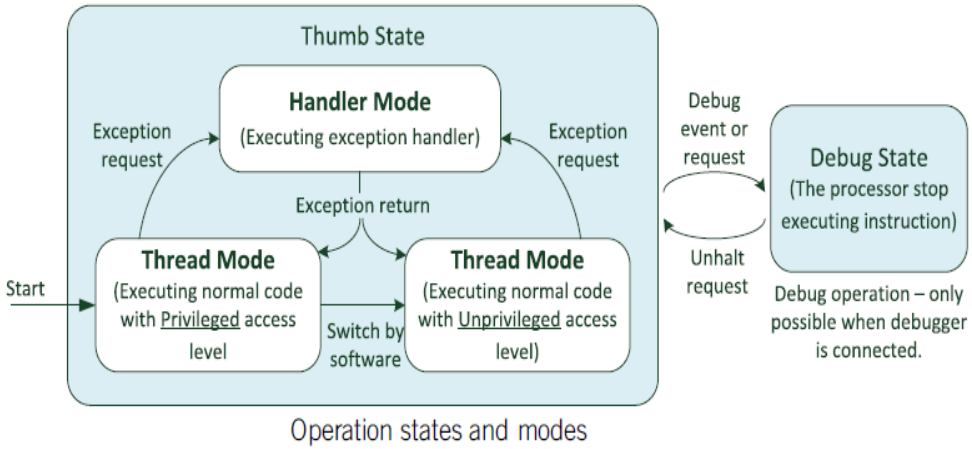
Programmer's model : Operation modes and states

- The Cortex-M3 and Cortex-M4 processors have two operation states and two modes.
- In addition, the processors can have privileged and unprivileged access levels.
- The privileged access level can access all resources in the processor, while the unprivileged access level means some memory regions are inaccessible, and a few operations cannot be used.
- The unprivileged access level might also be referred to as the "User" state, a term inherited from ARM7TDMI.

Programmer's model : Operation modes and states

- **Operation states :**
  - Debug state: When the processor is halted (e.g., by the debugger or after hitting a breakpoint), it enters debug state and stops executing instructions.
  - Thumb state: If the processor is running program code (Thumb instructions), it is in the Thumb state. Unlike classic ARM processors like ARM7TDMI, there is no ARM state because the Cortex-M processors do not support the ARM instruction set.

Programmer's model : Operation modes and states



Operation states and modes

Programmer's model : Operation modes and states

- **Operation modes**
  - **Handler mode:** When executing an exception handler such as an Interrupt Service Routine (ISR). When in handler mode, the processor always has a privileged access level.
  - **Thread mode:** When executing normal application code, the processor can be either in privileged access level or unprivileged access level. This is controlled by a special register called "CONTROL."

Programmer's model : Operation modes and states

- Software can switch the processor in privileged Thread mode to unprivileged Thread mode.
- However, it cannot switch itself back from unprivileged to privileged.
- If this is needed, the processor has to use the exception mechanism to handle the switch.
- The separation of privileged and unprivileged access levels allows system designers to develop robust embedded systems by providing a mechanism to safeguard memory accesses to critical regions and by providing a basic security model.

Programmer's model : Operation modes and states

- For example, a system can contain an embedded OS kernel that executes in privileged access level, and application tasks which execute in unprivileged access level.
- In this way, we can set up memory access permissions using the Memory Protection Unit (MPU) to prevent an application task from corrupting memory and peripherals used by the OS kernel and other tasks.
- If an application task crashes, the remaining application tasks and the OS kernel can still continue to run.

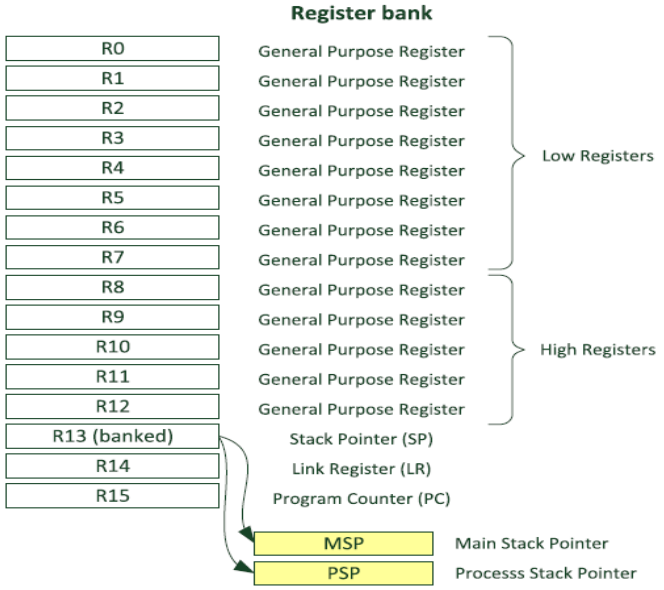
Programmer's model : Operation modes and states

- Thread mode and Handler mode have very similar programmer's models.
- However, Thread mode can switch to using a separate shadowed Stack Pointer (SP).
- Again, this allows the stack memory for application tasks to be separated from the stack used by the OS kernel, thus allowing better system reliability.
- By default, the Cortex-M processors start in privileged Thread mode and in Thumb state.
- In many simple applications, there is no need to use the unprivileged Thread model and the shadowed SP at all.

Programmer's model : Operation modes and states

- The debug state is used for debugging operations only.
- This state is entered by a halt request from the debugger or by debug events generated from debug components in the processor.
- This state allows the debugger to access or change the processor register values.
- The system memory, including peripherals inside and outside the processor, can be accessed by the debugger in either the Thumb state or debug state.

Programmer's model : Registers



- The register bank in the Cortex-M4 processors has 16 registers.
- Thirteen of them are general purpose 32-bit registers, and the other three have special uses.

Programmer's model : Registers

- **R0 - R12**
- Registers R0 to R12 are general purpose registers.
- The first eight (R0 - R7) are also called low registers.
- Due to the limited available space in the instruction set, many 16-bit instructions can only access the low registers.
- The high registers (R8 - R12) can be used with 32-bit instructions, and a few with 16-bit instructions, like MOV (move). The initial values of R0 to R12 are undefined.

Programmer's model : Registers

- **R13, stack pointer (SP)**
- It is used for accessing the stack memory via PUSH and POP operations.
- Physically there are two different Stack Pointers: the Main Stack Pointer (MSP, or SP_main in some ARM documentation) is the default Stack Pointer. It is selected after reset, or when the processor is in Handler Mode.
- The other Stack Pointer is called the Process Stack Pointer (PSP, or SP_process in some ARM documentation). The PSP can only be used in Thread Mode. The selection of Stack Pointer is determined by a special register called CONTROL.

Programmer's model : Registers

- Both MSP and PSP are 32-bit, but the lowest two bits of the Stack Pointers (either MSP or PSP) are always zero, and writes to these two bits are ignored.
- In ARM Cortex-M processors, PUSH and POP are always 32-bit, and the addresses of the transfers in stack operations must be aligned to 32-bit word boundaries.
- In most cases, it is not necessary to use the PSP if the application doesn't require an embedded OS.
- Many simple applications can rely on the MSP completely.
- The PSP is normally used when an embedded OS is involved, where the stack for the OS kernel and application tasks are separated.
- The initial value of PSP is undefined, and the initial value of MSP is taken from the first word of the memory during the reset sequence.

Programmer's model : Registers

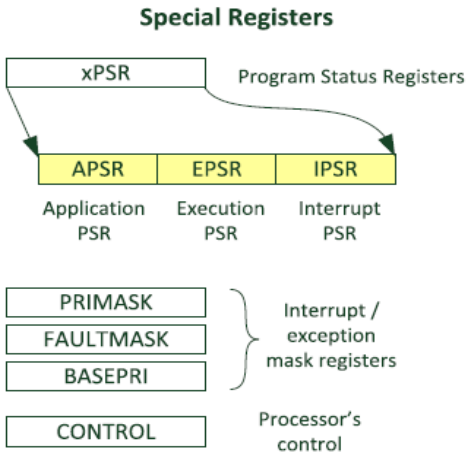
- **R14, link register (LR)**
- R14 is also called the Link Register (LR). This is used for holding the return address when calling a function or subroutine.
- At the end of the function or subroutine, the program control can return to the calling program and resume by loading the value of LR into the Program Counter (PC).
- When a function or subroutine call is made, the value of LR is updated automatically.
- If a function needs to call another function or subroutine, it needs to save the value of LR in the stack first.
- Otherwise, the current value in LR will be lost when the function call is made.

Programmer's model : Registers

- **R15, program counter (PC)**
- It is readable and writeable: a read returns the current instruction address plus 4 (this is due to the pipeline nature of the design, and compatibility requirement with the ARM7TDMI processor).
- Writing to PC (e.g., using data transfer/processing instructions) causes a branch operation.
- Since the instructions must be aligned to half-word or word addresses, the Least Significant Bit (LSB) of the PC is zero.

Programmer's model : Special registers

- **Program status registers**
- The Program Status Register is composed of three status registers:
- Application PSR (APSR)
- Execution PSR (EPSR)
- Interrupt PSR (IPSR)



Programmer's model : Special registers

Special Registers

	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				GE*							
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

*GE is available in ARMv7E-M processors such as the Cortex-M4. It is not available in the Cortex-M3 processor.

APSR, IPSR, and EPSR

	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		GE*	ICI/IT		Exception Number				

*GE is available in ARMv7E-M processors such as the Cortex-M4. It is not available in the Cortex-M3 processor.

Combined xPSR

Programmer's model : Special registers

Bit Fields in Program Status Registers

Bit	Description
N	Negative flag
Z	Zero flag
C	Carry (or NOT borrow) flag
V	Overflow flag
Q	Sticky saturation flag (not available in ARMv6-M)
GE[3:0]	Greater-Than or Equal flags for each byte lane (ARMv7E-M only; not available in ARMv6-M or Cortex®-M3).
ICI/IT	Interrupt-Continuable Instruction (ICI) bits, IF-THEN instruction status bit for conditional execution (not available in ARMv6-M).
T	Thumb state, always 1; trying to clear this bit will cause a fault exception.
Exception Number	Indicates which exception the processor is handling.

Programmer's model : Special registers

- **Program status registers**
- These three registers can be accessed as one combined register, referred to as xPSR
- The EPSR cannot be accessed by software code directly using MRS (read as zero) or MSR
- The IPSR is read only and can be read from combined PSR (xPSR).

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
ARM general (Cortex-A/R)	N	Z	C	V	Q	IT	J	Reserved	GE[3:0]	IT	E	A	I	F	T	M[4:0]
ARM7TDMI (ARMv4)	N	Z	C	V	Reserved								I	F	T	M[4:0]
ARMv7-M (Cortex-M3)	N	Z	C	V	Q	ICI/IT	T			ICI/IT		Exception Number				
ARMv7E-M (Cortex-M4)	N	Z	C	V	Q	ICI/IT	T		GE[3:0]	ICI/IT		Exception Number				
ARMv6-M (Cortex-M0)	N	Z	C	V			T									Exception Number

Comparing PSR of various ARM architectures

Programmer's model : Special registers

- **CONTROL register**
- The CONTROL register defines:
- The selection of stack pointer (Main Stack Point/Process Stack Pointer)
- Access level in Thread mode (Privileged/Unprivileged)
- In addition, for Cortex-M4 processor with a floating point unit, one bit of the CONTROL register indicates if the current context (currently executed code) uses the floating point unit or not.
- The CONTROL register can only be modified in the privileged access level and can be read in both privileged and unprivileged access levels.

Programmer's model : Special registers

		31:3	2	1	0
Cortex-M3 Cortex-M4	CONTROL			SPSEL	nPRIV
		31:3	2	1	0
Cortex-M4 with FPU	CONTROL		FPCA	SPSEL	nPRIV
		31:3	2	1	0
ARMv6-M (e.g. Cortex-M0)	CONTROL			SPSEL	nPRIV

CONTROL register in Cortex-M3, Cortex-M4, Cortex-M4 with FPU. The bit nPRIV is not available in the Cortex-M0 and is optional in the Cortex-M0+ processor

Programmer's model : Special registers

- **CONTROL register**
- After reset, the CONTROL register is 0. This means the Thread mode uses the Main Stack Pointer as Stack Pointer and Thread mode has privileged accesses.
- Programs in privileged Thread mode can switch the Stack Pointer selection or switch to unprivileged access level by writing to CONTROL .
- However, once nPRIV (CONTROL bit 0) is set, the program running in Thread can no longer access the CONTROL register.

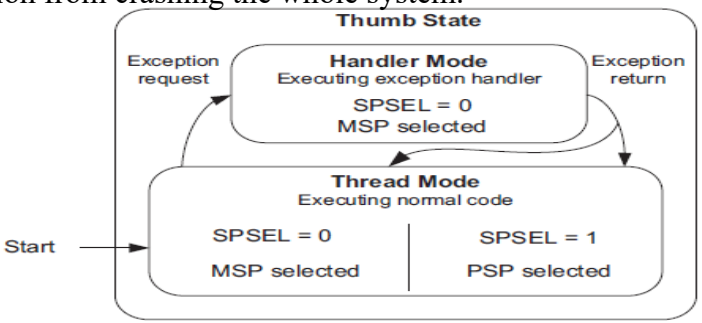
Programmer's model : Special registers

Bit Fields in CONTROL Register

Bit	Function
nPRIV (bit 0)	Defines the privileged level in Thread mode: When this bit is 0 (default), it is privileged level when in Thread mode. When this bit is 1, it is unprivileged when in Thread mode. In Handler mode, the processor is always in privileged access level.
SPSEL (bit 1)	Defines the Stack Pointer selection: When this bit is 0 (default), Thread mode uses Main Stack Pointer (MSP). When this bit is 1, Thread mode uses Process Stack Pointer (PSP). In Handler mode, this bit is always 0 and write to this bit is ignored.
FPCA (bit 2)	Floating Point Context Active – This bit is only available in Cortex-M4 with floating point unit implemented. The exception handling mechanism uses this bit to determine if registers in the floating point unit need to be saved when an exception has occurred. When this bit is 0 (default), the floating point unit has not been used in the current context and therefore there is no need to save floating point registers. When this bit is 1, the current context has used floating point instructions and therefore need to save floating point registers. The FPCA bit is set automatically when a floating point instruction is executed. This bit is clear by hardware on exception entry. There are several options for handling saving of floating point registers.

Programmer's model : Special registers

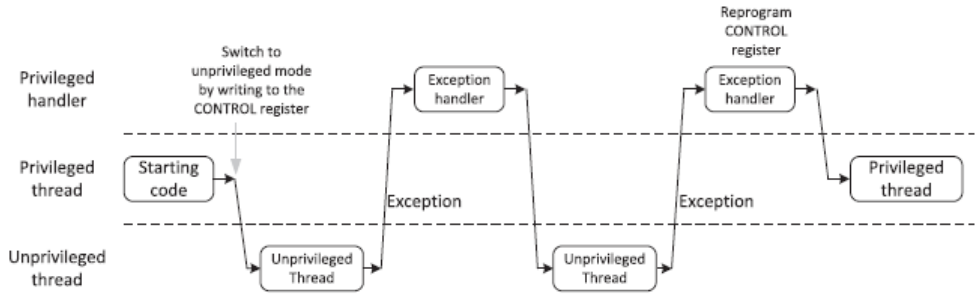
- A program in unprivileged access level cannot switch itself back to privileged access level.
- This is essential in order to provide a basic security usage model.
- For example, an embedded system might contain untrusted applications running in unprivileged access level and the access permission of these applications must be restricted to prevent security breaches or to prevent an unreliable application from crashing the whole system.



Stack Pointer selection

Programmer's model : Special registers

- If it is necessary to switch the processor back to using privileged access level in Thread mode, then the exception mechanism is needed.
- During exception handling, the exception handler can clear the nPRIV bit.
- When returning to Thread mode, the processor will be in privileged access level.



Switching between privileged thread mode and unprivileged thread mode

Programmer's model : Special registers

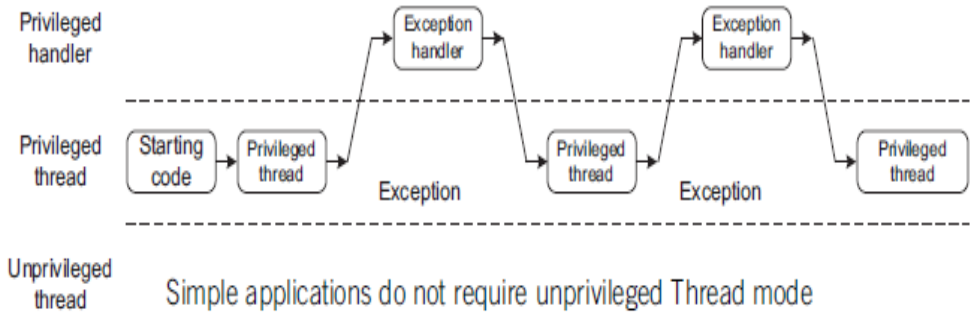
- When an embedded OS is used, the CONTROL register could be reprogrammed at each context switch to allow some application tasks to run with privileged access level and the others to run with unprivileged access level.
- The settings of nPRIV and SPSEL are orthogonal.
- Four different combinations of nPRIV and SPSEL are possible, although only three of them are commonly used in real world applications.
- In most simple applications without an embedded OS, there is no need to change the value of the CONTROL register.
- The whole application can run in privileged access level and use only the MSP.

Programmer's model : Special registers

Different Combinations of nPRIV and SPSEL

nPRIV	SPSEL	Usage Scenario
0	0	Simple applications – the whole application is running in privileged access level. Only one stack is used by the main program and interrupt handlers. Only the Main Stack Pointer (MSP) is used.
0	1	Applications with an embedded OS, with current executing task running in privileged Thread mode. The Process Stack Pointer (PSP) is selected in current task, and the MSP is used by OS Kernel and exception handlers.
1	1	Applications with an embedded OS, with current executing task running in unprivileged Thread mode. The Process Stack Pointer (PSP) is selected in current task, and the MSP is used by OS Kernel and exception handlers.
1	0	Thread mode tasks running with unprivileged access level and use MSP. This can be observed in Handler mode but is less likely to be used for user tasks because in most embedded OS, the stack for application tasks is separated from the stack used by OS kernel and exception handlers.

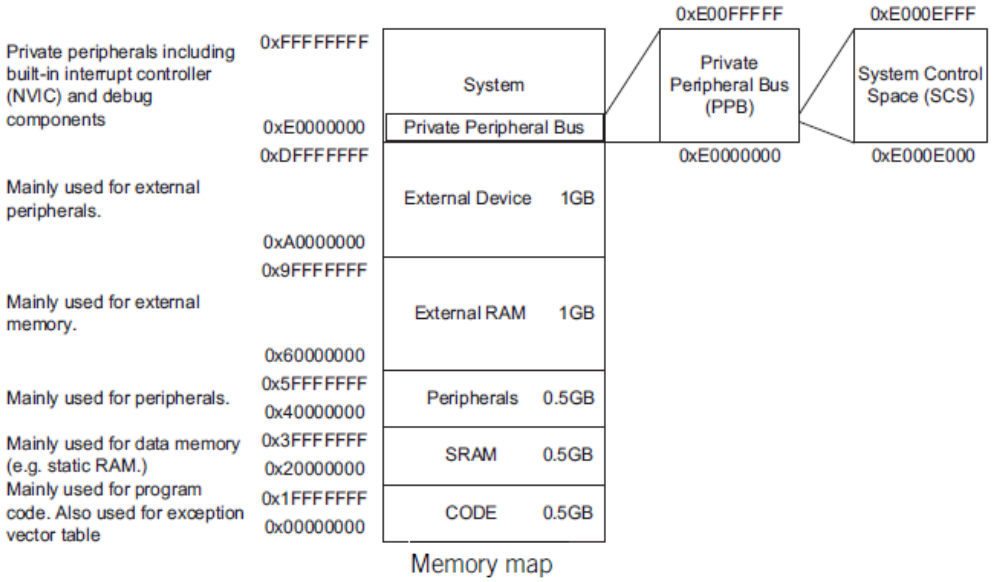
Programmer's model : Special registers



Memory map

- The 4GB address space of the Cortex-M processors is partitioned into a number of memory regions. The partitioning is based on typical usages so that different areas are designed to be used primarily for:
  - Program code accesses (e.g., CODE region)
  - Data accesses (e.g., SRAM region)
  - Peripherals (e.g., Peripheral region)
  - Processor's internal control and debug components (e.g., Private Peripheral Bus)
- The architecture also allows high flexibility to allow memory regions to be used for other purposes. For example, programs can be executed from the CODE as well as the SRAM region, and a microcontroller can also integrate SRAM blocks in CODE region.

Memory map



Memory map

Memory map : Stack memory

- Physically there are two stack pointers in the Cortex-M processors. They are the:
- **Main Stack Pointer (MSP)** - This is the default stack pointer used after reset, and is used for all exception handlers.
- **Process Stack Pointer (PSP)** - This is an alternate stack point that can only be used in Thread mode. It is usually used for application tasks in embedded systems running an embedded OS.

Exceptions and interrupts

- In Cortex-M processors, there are a number of exception sources:
- Exceptions are processed by the NVIC. The NVIC can handle a number of Interrupt Requests (IRQs) and a Non-Maskable Interrupt (NMI) request.
- Usually IRQs are generated by on-chip peripherals or from external interrupt inputs though I/O ports.
- The NMI could be used by a watchdog timer or brownout detector (a voltage monitoring unit that warns the processor when the supply voltage drops below a certain level).
- Inside the processor there is also a timer called SysTick, which can generate a periodic timer interrupt request, which can be used by embedded OSs for timekeeping, or for simple timing control in applications that don't require an OS.

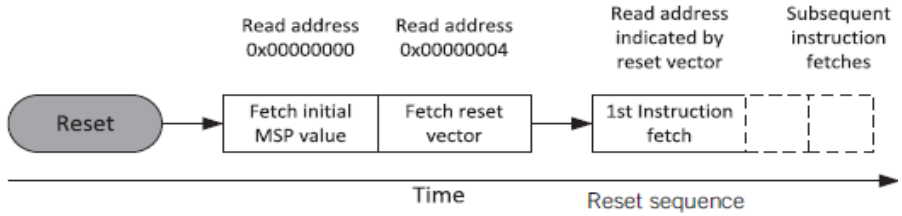
System control block (SCB)

- One part of the processor that is merged into the NVIC unit is the SCB.
- The SCB contains various registers for:
  - Controlling processor configurations (e.g., low power modes)
  - Providing fault status information (fault status registers)
  - Vector table relocation (VTOR)
- The SCB is memory-mapped. Similar to the NVIC registers, the SCB registers are accessible from the System Control Space (SCS).

Reset and reset sequence

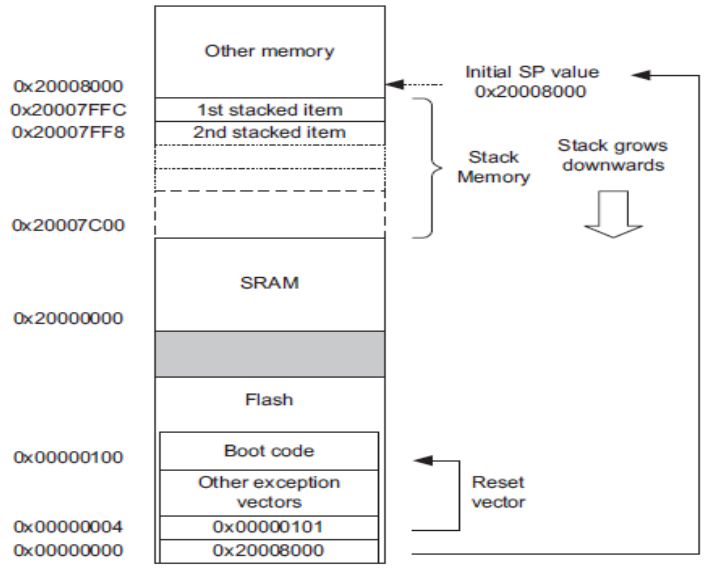
- In typical Cortex-M microcontrollers, there can be three types of reset:
- Power on reset - reset everything in the microcontroller. This includes the processor and its debug support component and peripherals.
- System reset - reset just the processor and peripherals, but not the debug support component of the processor.
- Processor reset - reset the processor only.
- The duration of Power on reset and System reset depends on the microcontroller design.
- In some cases the reset lasts a number of milli seconds as the reset controller needs to wait for a clock source such as a crystal oscillator to stabilize.

Reset and reset sequence



- The setup of the MSP is necessary because some exceptions such as the NMI or HardFault handler could potentially occur shortly after the reset, and the stack memory and hence the MSP will then be needed to push some of the processor status to the stack before exception handling.
- Because the stack operations in the Cortex-M3 or Cortex-M4 processors are based on full descending stack (SP decrement before store), the initial SP value should be set to the first memory after the top of the stack region. For example, if you have a stack memory range from 0x20007C00 to 0x20007FFF (1Kbytes), the initial stack value should be set to 0x20008000

Reset and reset sequence



Initial Stack Pointer value and Initial Program Counter value example

Prof. Manoj S Kakade

Email id: manoj.kakade@pilani.bits-pilani.ac.in

THANK YOU!!!!!

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE

innovate achieve lead

BITS Pilani
Pilani Campus

ESZG512/MELZG526/SEZG516
Embedded System Design
Contact Session 5

Contents

- Polling, interrupt, DMA

ARM Cortex-M4:

- Exceptions and Interrupts
- Exception handlers in C
- Exception entry behavior
- Exception return behavior

innovate achieve lead

Polling

- **Polling**
- The microcontroller continuously monitors the status of a given device.
- When the conditions are met, it performs the service.
- After that, it moves on to monitor the next device until everyone is serviced.
- The polling method is not efficient since it wastes much of the microcontroller's time by polling devices that do not need service.

Interrupt

- **Interrupt**
- Whenever any device needs its service, the device notifies the microcontroller by sending it an interrupt signal.
- Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device.
- The program which is associated with the interrupt is called the interrupt service routine (ISR) or interrupt handler.
- The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time)
  - Each device can get the attention of the microcontroller based on the assigned priority.
- The limiting factor for the data transfer is the time to recognize, process and return from the interrupt.

DMA

- **DMA**
- It is a device that can initiate and control, bus accesses between I/O devices and memory and between two memory areas.
- Using a DMA controller is reasonably simple, provided the programming defines exactly the data transfer operations that the processor expects.
- Low latency I/O data transfer method.

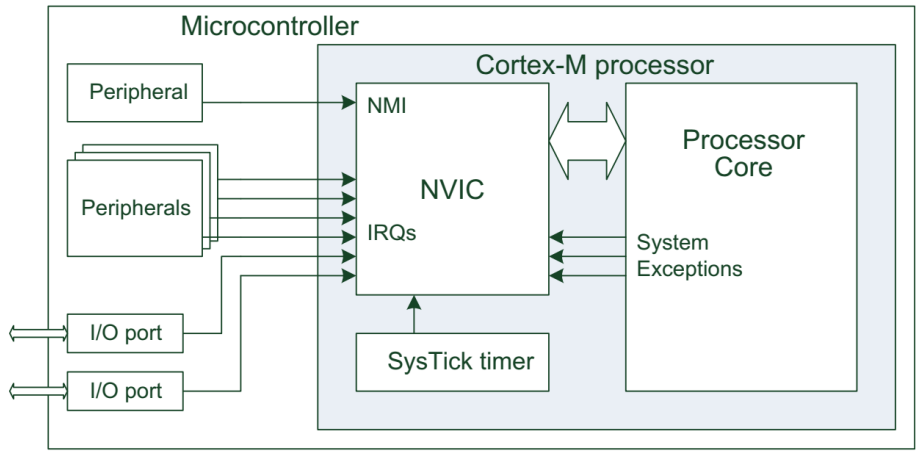
Overview of exceptions and interrupts

- Interrupts are events typically generated by hardware (e.g., peripherals or external input pins) that cause changes in program flow control outside a normal programmed sequence.
- In addition to interrupt requests, there are other events that need servicing, and we call them “exceptions.”
- In ARM terminology, an interrupt is one type of exception.
- All Cortex-M processors provide a Nested Vectored Interrupt Controller (NVIC) for exception handling.

Source: Joseph Yiu, “The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors”

Sources of Exceptions

Various sources of exceptions in a typical microcontroller



Source: Joseph Yiu, “The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors”

Sources of Exceptions

- **Cortex-M4 NVIC supports**:
- Up to 240 IRQs (Interrupt Requests)
- Non-Maskable Interrupt (NMI)
  - The NMI is usually generated from peripherals like a watchdog timer or Brown-Out Detector (BOD).
- SysTick (System Tick) timer interrupt.
- Number of system exceptions.
- Most of the IRQs are generated by peripherals such as timers, I/O ports, and communication interfaces (e.g., UART, I2C).
- The rest of the exceptions are from the processor core.
- Interrupts can also be generated using software.

Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

Exception types

- Exceptions are numbered 1 to 15 for system exceptions and 16 and above for interrupt inputs (inputs to the processor, but not necessarily accessible on the I/O pins of the package).
- Most of the exceptions, including all interrupts, have programmable priorities, and a few system exceptions have fixed priority.
- Different Cortex-M3 or Cortex-M4 microcontrollers can have different numbers of interrupt sources (from 1 to 240) and different numbers of priority levels.
  - This is because chip designers can configure the Cortex-M3 or Cortex-M4 design source code for different application requirements.

Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

Exception types

List of System Exceptions

Exception Number	Exception Type	Priority	Descriptions
1	Reset	–3 (Highest)	Reset
2	NMI	–2	Non-Maskable Interrupt (NMI), can be generated from on chip peripherals or from external sources.
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 program execution from address locations with XN (eXecute Never) memory attribute.

Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

Exception types

5	Bus Fault	Programmable	Bus error; usually occurs when AHB interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch or *data abort* if it is a data access). Can also be caused by other illegal accesses.
6	Usage Fault	Programmable	Exceptions due to program error or trying to access co-processor (the Cortex-M3 and Cortex-M4 processor do not support co-processors).
7–10	Reserved	NA	–
11	SVC	Programmable	SuperVisor Call; usually used in OS environment to allow application tasks to access system services.

Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

Exception types

12	Debug Monitor	Programmable	Debug monitor; exception for debug events like breakpoints, watchpoints when software based debug solution is used.
13	Reserved	NA	–
14	PendSV	Programmable	Pendable service call; An exception usually used by an OS in processes like context switching.
15	SYSTICK	Programmable	System Tick Timer; Exception generates by a timer peripheral which is included in the processor. This can be used by an OS or can be used as a simple timer peripheral.

Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

Exception types

16	Interrupt #0	Programmable	It can be generated from on chip peripherals or from external sources.
17	Interrupt #1	Programmable	
…	…	…	
255	Interrupt #239	Programmable	

Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

Exception handlers

Exception handlers:

The processor handles exceptions using:

Interrupt Service Routines (ISRs)

- Interrupts IRQ0 to IRQ239 are the exceptions handled by ISRs.

Fault handlers

- Hard fault, memory management fault, usage fault, bus fault are fault exceptions handled by the fault handlers.

System handlers

- NMI, PendSV, SVCall SysTick, and the fault exceptions are all system exceptions that are handled by system handlers.

Source: ARMv7M Architecture Reference Manual

Vector table

Exception number	IRQ number	Offset	Vector
255	239	0x03FC	IRQ239
.		.	.
.		.	.
.		.	.
		0x004C	
18	2	0x0048	IRQ2
17	1	0x0044	IRQ1
16	0	0x0040	IRQ0
15	-1	0x003C	Systick
14	-2	0x0038	PendSV
13			Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			
9			Reserved
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

Source: ARMv7M Architecture Reference Manual

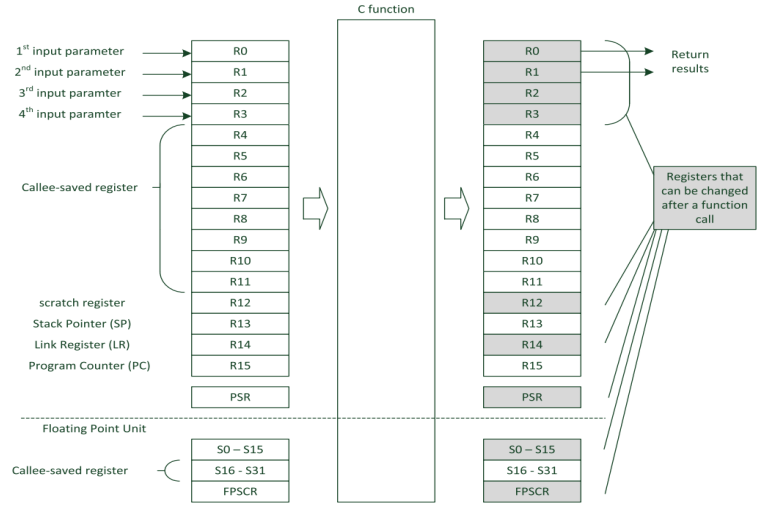
Exception handlers in C

- C compilers for ARM architecture follow a specification from ARM called the AAPCS (ARM Architecture Procedure Calling Standard), Procedure Call Standard for ARM Architecture.
- A C function can modify R0 to R3, R12, R14 (LR), and PSR.
- If the C function needs to use R4 to R11, it should save these registers on to the stack memory and restore them before the end of the function.
- R0 - R3, R12, LR, and PSR are called “caller saved registers.”
- R4 - R11 are called “callee-saved registers.”

Source: Joseph Yiu, “The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors”

Exception handlers in C

Register usage in a function call in AAPCS



Source: Joseph Yiu, “The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors”

Exception entry behavior

- On preemption of the instruction stream, the hardware saves context state onto a stack pointed to by one of the SP registers.
- The stack used depends on the mode of the processor at the time of the exception.
- The stacked context supports the ARM Architecture Procedure Calling Standard (AAPCS). This means the exception handler can be an AAPCS-compliant procedure.

Source: ARMv7M Architecture Reference Manual

The ARMv7-M architecture uses a full-descending stack, where:

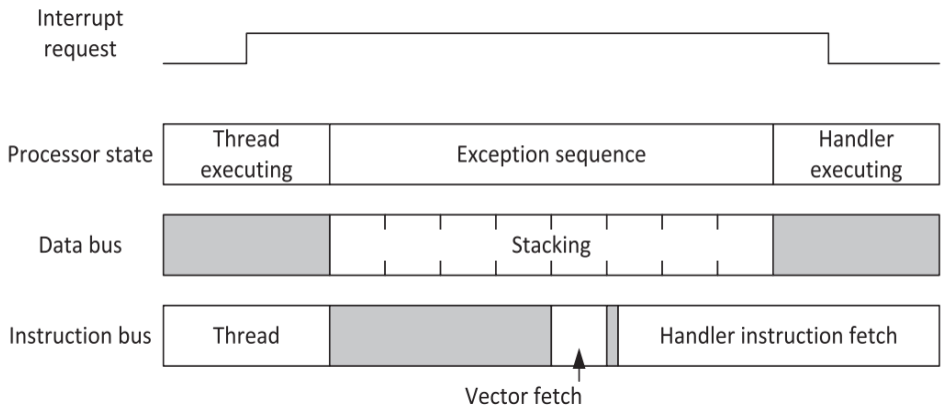
- When pushing context, the hardware decrements the stack pointer to the end of the new stack frame before it stores data onto the stack.
- When popping context, the hardware reads the data from the stack frame and then increments the stack pointer.

When pushing context to the stack, the hardware saves eight 32-bit words, comprising xPSR, Return Address, LR(R14), R12, R3, R2, R1, and R0.

Source: ARMv7M Architecture Reference Manual

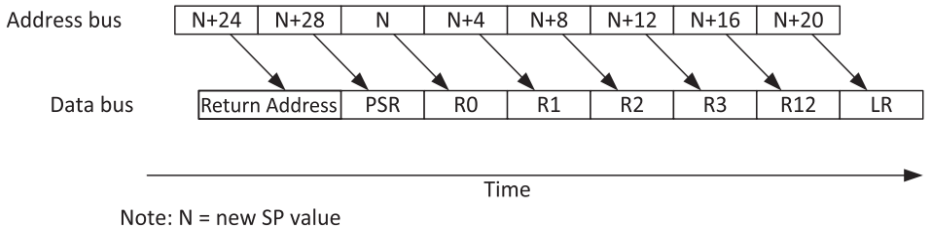
Stacking and vector fetch

Stacking and vector fetch



Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

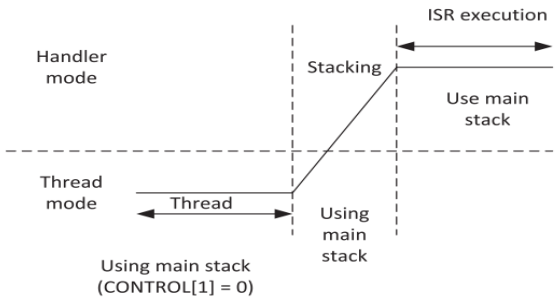
Stacking sequence in the Cortex-M3 processor on AHB lite interface



Note: N = new SP value

Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

- If the processor was in Thread mode, and was using MSP (bit 1 of the CONTROL register is 0, as in the default setting), the stacking operation is carried out in the main stack with MSP.

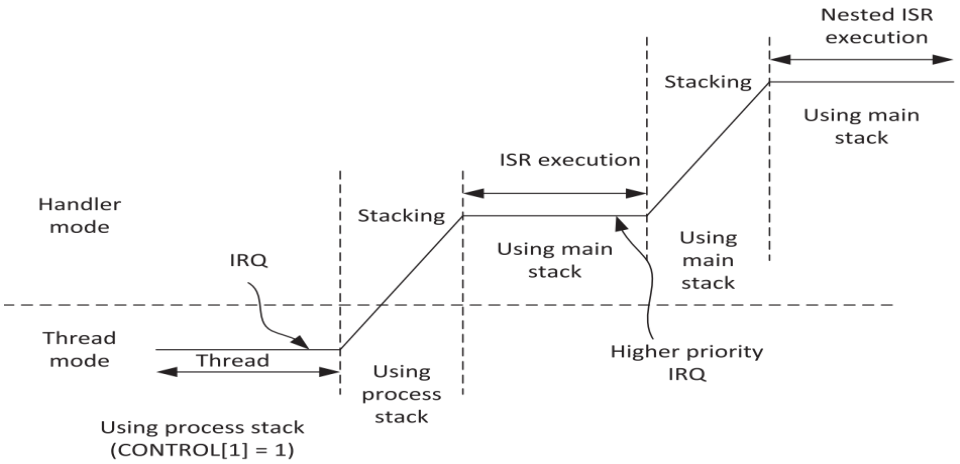


Exception stacking in thread mode using the main stack

Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

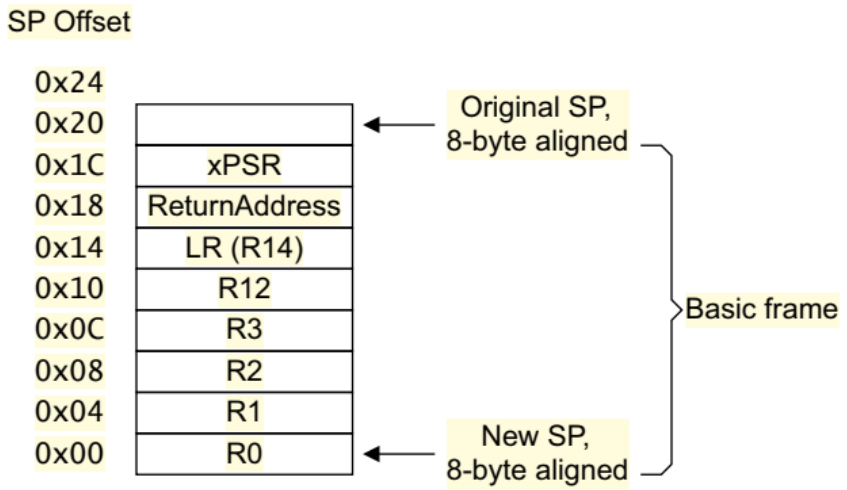
- If the processor was in Thread mode, and was using the process stack (bit 1 of the CONTROL register is 1), then the stacking operation is carried out in the process stack with PSP.
- After entering Handler mode, the processor must be using the MSP, so the stacking operation of all nested interrupts are carried out with the main stack with MSP.

Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"



Exception stacking in thread mode using the process stack, and nested interrupt stacking using the main stack

Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"



Source: ARMv7M Architecture Reference Manual

Saving context on process switch

- When switching between different processes, software must save all context for the old process, including its associated EXC_RETURN value, before switching to the new process, and restore that context before returning to the old process.

Source: ARMv7M Architecture Reference Manual

EXC_RETURN

- Since the value of the return address (PC) is not stored in LR as in normal C function calls (the exception mechanism puts an EXC_RETURN code in LR at exception entry, which is used in exception return), the value of the return address also needs to be saved by the exception sequence.
- So in total eight registers need to be saved during the exception handling sequence on the Cortex-M3 or Cortex-M4 processors without a floating point unit.

Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

EXC_RETURN

EXC_RETURN

- As the processor enters the exception handler or Interrupt Service Routine (ISR), the value of the Link Register (LR) is updated to a code called EXC_RETURN.
- The exception mechanism relies on this value to detect when the processor has completed an exception handler.
- The lowest five bits of this value provide information on the return stack and processor mode.

EXC_RETURN	Return to	Return stack
0xFFFFFFF1	Handler mode	Main
0xFFFFFFF9	Thread mode	Main
0xFFFFFFFD	Thread mode	Process

Source: ARMv7M Architecture Reference Manual

EXC_RETURN

Bit Fields of the EXC_RETURN

Bits	Descriptions	Values
31:28	EXC_RETURN indicator	0xF
27:5	Reserved (all 1)	0xEFFFFF (23 bits of 1)
4	Stack Frame type	1 (8 words) or 0 (26 words). Always 1 when the floating unit is unavailable. This value is set to the inverted value of FPCA bit in the CONTROL register when entering an exception handler.
3	Return mode	1 (Return to Thread) or 0 (Return to Handler)
2	Return stack	1 (Return with Process Stack) or 0 (Return with Main Stack)
1	Reserved	0
0	Reserved	1

Source: ARMv7M Architecture Reference Manual

Exception return behavior

Exception return behavior

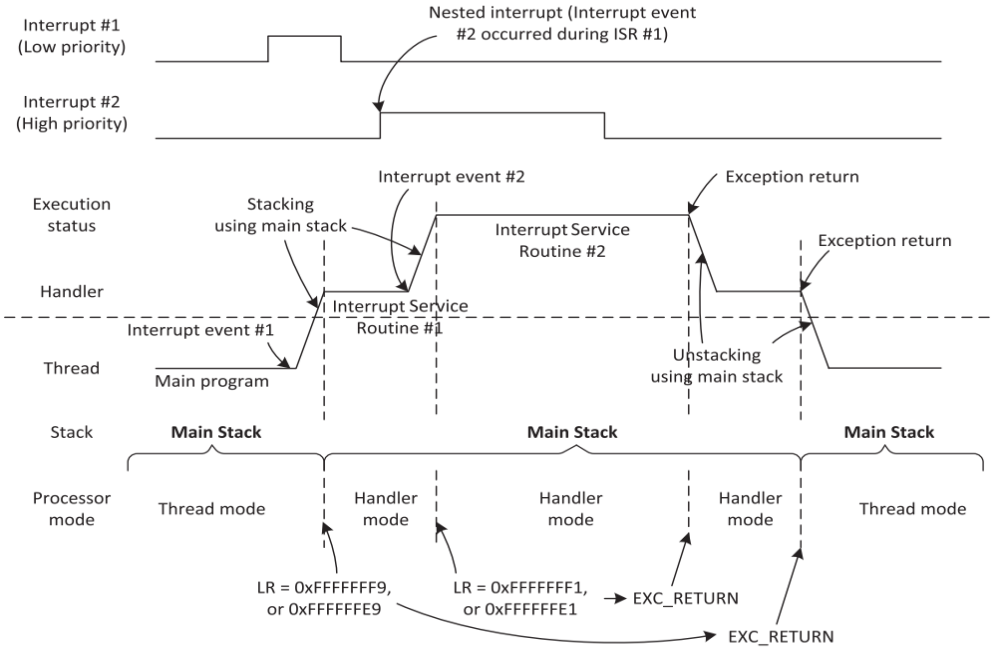
- An exception return occurs when the processor is in Handler mode and one of the following instructions loads a value of 0xFXXXXXXX into the PC:
- POP/LDM that includes loading the PC.
- LDR with PC as a destination.
- BX with any register.

When used in this way, the processor intercepts the value written to the PC. This value is the EXC_RETURN value.

Source: ARMv7M Architecture Reference Manual

- At the end of an exception handler, the bit 2 of the EXC_RETURN value generated at the exception entrance is used to decide which stack pointer should be used to extract the stack frame.
- If bit 2 is 0, the processor knows that the main stack was used for stacking, as shown in Figure.

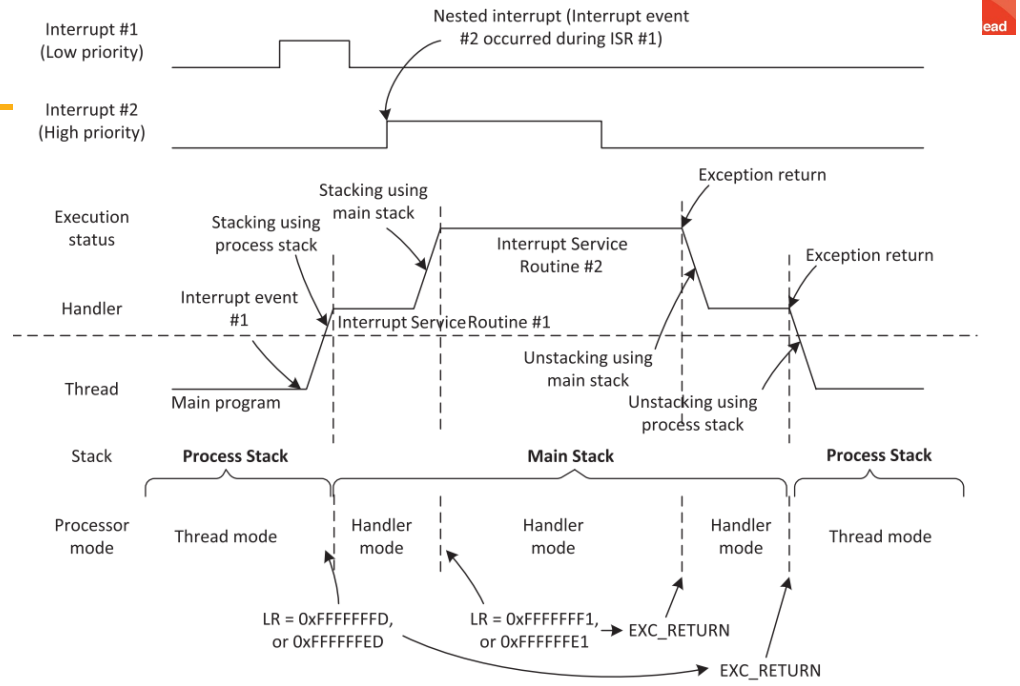
Source: ARMv7M Architecture Reference Manual



Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

- If bit 2 is 1, the processor knows that the process stack was used for stacking, as shown in the second unstacking operation in Figure

Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"



Source: Joseph Yiu, "The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors"

Lab Instructor :
Prof. S.S. Kendre
Email ID : kendres.shankarrao@wilp.bits-pilani.ac.in

Lab Access:
Mr. Vishal (08322580217/ 7720893522)
Email ID: vishal@wilp.bits-pilani.ac.in

Thank you

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE

innovate achieve lead

BITS Pilani
Pilani Campus

ESZG512/MELZG526/SEZG516 Embedded System Design Contact Session 6

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

innovate achieve lead

Small Scale Embedded System Design

- Understanding Complete Embedded System Design
- Procedure starting from System Requirements to Final Design

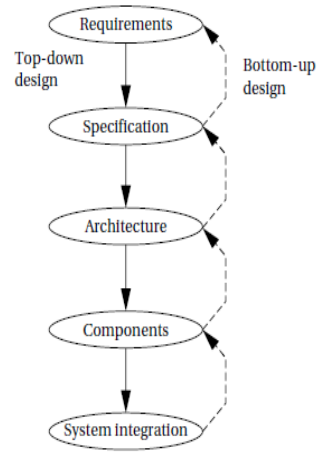
BITS Pilani
Pilani | Dubai | Goa | Hyderabad

innovate achieve lead

THE EMBEDDED SYSTEM DESIGN PROCESS

- First, it allows us to keep a scorecard on a design to ensure that we have done everything we need to do, such as optimizing ***performance*** or performing functional tests.
- Second, it allows us to develop computer-aided design tools.
- Developing a single program that takes in a concept for an embedded system and emits a completed design would be a daunting task, but by first breaking the process into manageable steps, we can work on automating (or at least semi automating) the steps one at a time.
- Third, a design methodology makes it much easier for members of a design team to communicate.
- By defining the overall process, team members can more easily understand what they are supposed to do, what they should receive from other team members at certain times, and what they are to hand off when they complete their assigned steps.

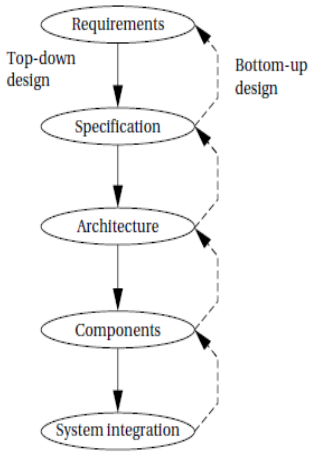
THE EMBEDDED SYSTEM DESIGN PROCESS



Major levels of abstraction in the design process.

- In **top–down** view , we start with the system ***requirements***.
- ***Specification***, we create a more detailed description of what we want. But the specification states only *how the system behaves, not how it is built.*
- The details of the system's internals begin to take shape when we develop the architecture, which gives the system structure in terms of large components.
- Once we know the components we need, we can design those components, including both software modules and any specialized hardware we need.
- Based on those components , we can finally build a complete system.

THE EMBEDDED SYSTEM DESIGN PROCESS



Major levels of abstraction in the design process.

- We need **bottom–up** design because we do not have perfect insight into how later stages of the design process will turn out.
- Decisions at one stage of design are based upon estimates of what will happen later:
- How fast can we make a particular function run?
- How much memory will we need?
- How much system bus capacity do we need?
- If our estimates are inadequate, we may have to backtrack and amend our original decisions to take the new facts into account.
- In general, the less experience we have with the design of similar systems, the more we will have to rely on bottom-up design information to help us refine the system.

THE EMBEDDED SYSTEM DESIGN PROCESS

- We also need to consider the major goals of the design:
  - manufacturing cost
  - performance (both overall speed and deadlines)
  - power consumption

- At each step in the design, we add detail:
  - We must *analyze* the design at each step to determine how we can meet the specifications.
  - We must then *refine* the design to add detail.
  - And we must verify the design to ensure that it still meets all system goals, such as cost, speed, and so on.

Requirements

We generally proceed in two phases:

- First, we gather an informal description from the customers known as requirements, and
- we refine the requirements into a specification that contains enough information to begin designing the system architecture.
- Requirements may be ***functional*** or ***non functional***.
- We must of course capture the basic functions of the embedded system, but functional description is often not sufficient.

Requirements

Typical non functional requirements include:-

- *Performance:* The speed of the system is often a major consideration both for the usability of the system and for its ultimate cost.
- *Cost:* The target cost or purchase price for the system is almost always a consideration. Cost typically has two major components:
  - ***manufacturing cost*** includes the cost of components and assembly;
  - ***nonrecurring engineering*** **(NRE)** costs include the personnel and other costs of designing the system

Requirements

Typical nonfunctional requirements include:-

- *Physical size and weight:* The physical aspects of the final system can vary greatly depending upon the application.
- *Power consumption:* Power, of course, is important in battery-powered systems and is often important in other applications as well. Power can be specified in the requirements stage in terms of battery life—the customer is unlikely to be able to describe the allowable wattage.

Requirements

Sample requirements form.:-

- Name
- Purpose
- Inputs
- Outputs
- Functions
- Performance
- Manufacturing cost
- Power
- Physical size and weight

After writing the requirements, you should check them for internal consistency:

- Did you forget to assign a function to an input or output?
- Did you consider all the modes in which you want the system to operate?
- Did you place an unrealistic number of features into a battery-powered, low-cost machine?

Requirements :- Example

Requirements analysis of a GPS moving map

- The moving map is a handheld device that displays for the user a map of the terrain around the user's current position; the map display changes as the user and the map device change position.
- ***Functionality:*** This system is designed for highway driving and similar uses, not nautical or aviation uses that require more specialized databases and functions. The system should show major roads and other landmarks available in standard topographic databases.
- ***User interface:*** The screen should have at least 400x600 pixel resolution. The device should be controlled by no more than three buttons. A menu system should pop up on the screen when buttons are pressed to allow the user to make selections to control the system.

Requirements :- Example

Requirements analysis of a GPS moving map

- ***Performance:*** The map should scroll smoothly. Upon power-up, a display should take no more than one second to appear, and the system should be able to verify its position and display the current map within 15 s.
- ***Cost:*** The selling cost (street price) of the unit should be no more than $100.
- ***Physical size and weight:*** The device should fit comfortably in the palm of the hand.
- ***Power consumption:*** The device should run for at least eight hours on four AA batteries.
- *Note that many of these requirements are not specified in engineering units—for example, physical size is measured relative to a hand, not in centimeters.*

Requirements :- Example

Requirements analysis of a GPS moving map

- This chart adds some requirements in engineering terms that will be of use to the designers
- Name:- GPS moving map
- Purpose :- Consumer-grade moving map for driving use
- Inputs :- Power button, two control buttons
- Outputs :- Back-lit LCD display 400 x 600
- Functions :- Uses 5-receiver GPS system; three user-selectable resolutions; always displays current latitude and longitude
- Performance:- Updates screen within 0.25 seconds upon movement
- Manufacturing cost:- $30
- Power:- 100mW
- Physical size and weight :- No more than 2” 6, ” 12 ounces

Specifications

- The specification is more precise—it serves as the contract between the customer and the architects. As such, the specification must be carefully written so that it accurately reflects the customer’s requirements and does so in a way that can be clearly followed during design.
- The specification should be understandable enough so that someone can verify that it meets system requirements and overall expectations of the customer.
- It should also be unambiguous enough that designers know what they need to build.

Specifications

- Designers can run into several different types of problems caused by unclear specifications.
- If the behavior of some feature in a particular situation is unclear from the specification, the designer may implement the wrong functionality.
- If global characteristics of the specification are wrong or incomplete, the overall system architecture derived from the specification may be inadequate to meet the needs of implementation.

Specifications

A specification of the GPS system would include several components:

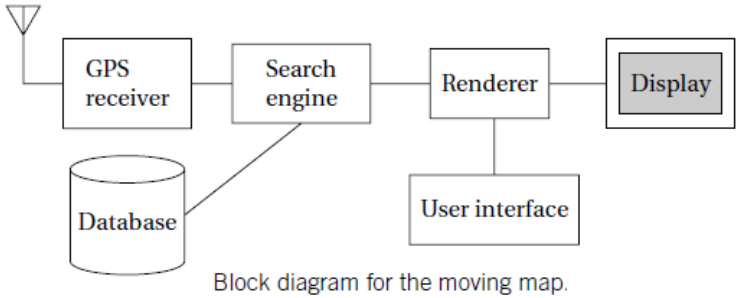
- Data received from the GPS satellite constellation.
- Map data.
- User interface.
- Operations that must be performed to satisfy customer requests.
- Background actions required to keep the system running, such as operating the GPS receiver.

Architecture Design

- The specification does not say how the system does things, only what the system does.
- Describing how the system implements those functions is the purpose of the architecture.
- The architecture is a plan for the overall structure of the system that will be used later to design the components that make up the architecture.
- The creation of the architecture is the first phase of what many designers think of as design.

Architecture Design

- System architecture in the form of a ***block diagram*** that shows major operations and data flows among them.



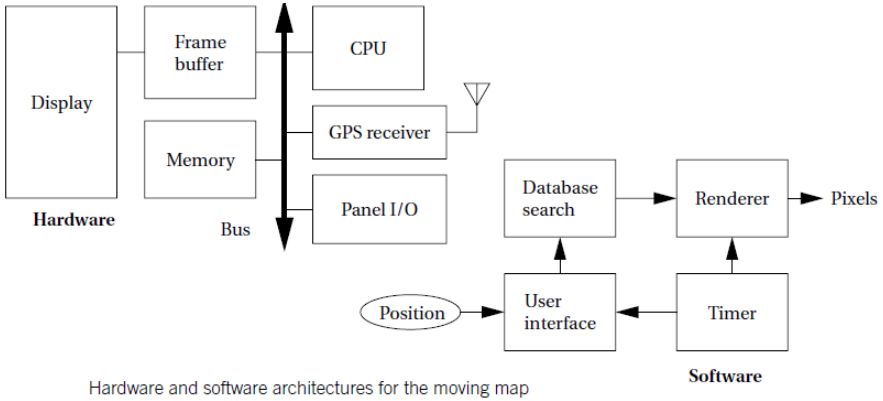
Block diagram for the moving map.

Architecture Design

- The diagram is still quite abstract,we have not yet specified which operations will be performed by software running on a CPU, what will be done by special-purpose hardware, and so on.
- The diagram does, however, go a long way toward describing how to implement the functions described in the specification.
- We clearly see, for example, that we need to search the topographic database and to render (i.e., draw) the results for the display.
- We have chosen to separate those functions so that we can potentially do them in parallel—performing rendering separately from searching the database may help us update the screen more fluidly.

Architecture Design

- Only after we have designed an initial architecture that is not biased toward too many implementation details should we refine that system block diagram into two block diagrams: one for hardware and another for software.



Hardware and software architectures for the moving map

Architecture Design

- The ***hardware block diagram*** clearly shows that we have one central CPU surrounded by memory and I/O devices. In particular, we have chosen to use two memories: a frame buffer for the pixels to be displayed and a separate program/data memory for general use by the CPU.
- The ***software block diagram*** fairly closely follows the system block diagram, but we have added a timer to control when we read the buttons on the user interface and render data onto the screen.
- To have a truly complete architectural description, we require more detail, such as where units in the software block diagram will be executed in the hardware block diagram and when operations will be performed in time.

Architecture Design

- Architectural descriptions must be designed to satisfy both functional and nonfunctional requirements.
- Not only must all the required functions be present, but we must meet cost, speed, power, and other nonfunctional constraints.
- Starting out with a system architecture and refining that to hardware and software architectures is one good way to ensure that we meet all specifications.
- We can concentrate on the functional elements in the system block diagram, and then consider the nonfunctional constraints when creating the hardware and software architectures.

Designing Hardware and Software Components

- The architectural description tells us what components we need.
- The component design effort builds those components in conformance to the architecture and specification.
- The components will in general include both hardware—FPGAs, boards and so on—and software modules.
- Some of the components will be ready-made. The CPU, will be a standard component in almost all cases, as will memory chips and many other components.
- The GPS receiver is a good example of a specialized component that will nonetheless be a predesigned, standard component.

Designing Hardware and Software Components

- We ca also make use of standard software modules.
- One good example is the topographic database.
- Standard topographic databases exist, and you probably want to use standard routines to access the database—not only is the data in a predefined format, but it is highly compressed to save storage.
- Using standard software for these access functions not only saves our design time, but it may give us a faster implementation for specialized functions such as the data decompression phase.
- You will have to design some components yourself.
- Even if you are using only standard integrated circuits, you may have to design the printed circuit board that connects them.
- You will probably have to do a lot of custom programming as well.

Designing Hardware and Software Components

- When creating embedded software modules, you must make use of your expertise to ensure that the system runs properly in real time and that it does not take up more memory space than is allowed.
- The power consumption of the GPS software example is particularly important.
- You may need to be very careful about how you read and write memory to minimize power—for example, since memory accesses are a major source of power consumption, memory transactions must be carefully planned to avoid reading the same data several times.

System Integration

- Only after the components are built do we have the satisfaction of putting them together and seeing a working system.
- Bugs are typically found during system integration, and good planning can help us find the bugs quickly.
- By building up the system in phases and running properly chosen tests, we can often find bugs more easily.
- If we debug only a few modules at a time, we are more likely to uncover the simple bugs and able to easily recognize them.
- We need to ensure during the architectural and component design phases that we make it as easy as possible to assemble the system in phases and test functions relatively independently.

Design challenge – optimizing design metrics

- Obvious design goal:
  - Construct an implementation with desired functionality
- Key design challenge:
  - Simultaneously optimize numerous design metrics
- Design metric
  - A measurable feature of a system's implementation
  - Optimizing design metrics is a key challenge

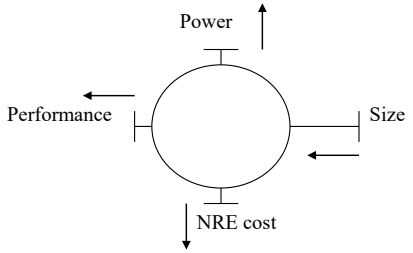
Design challenge – optimizing design metrics

- Common metrics
  - 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
  - Size: the physical space required by the system
  - Performance: the execution time or throughput of the system
  - Power: the amount of power consumed by the system
  - Flexibility: the ability to change the functionality of the system without incurring heavy NRE cost
  - Time-to-prototype: the time needed to build a working version of the system
  - Time-to-market: the time required to develop a system to the point that it can be released and sold to customers
  - Maintainability: the ability to modify the system after its initial release
  - Correctness, safety, many more

BITS Pilani
Pilani | Dubai | Goa | Hyderabad
innovate achieve lead

Design challenge – optimizing design metrics

- Design metric competition -- improving one may worsen others

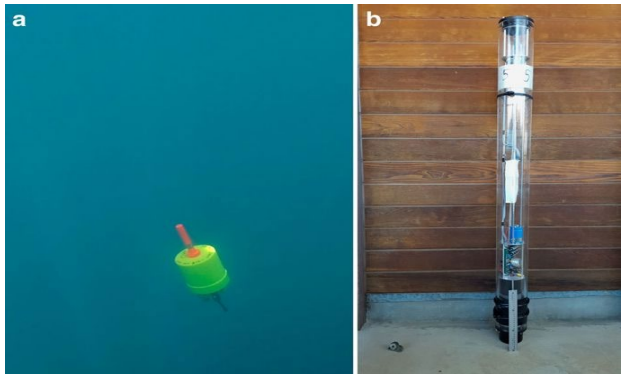


- Expertise with both software and hardware is needed to optimize design metrics
  - Not just a hardware or software expert, as is common
  - A designer must be comfortable with various technologies in order to choose the best for a given application and constraints

innovate achieve lead

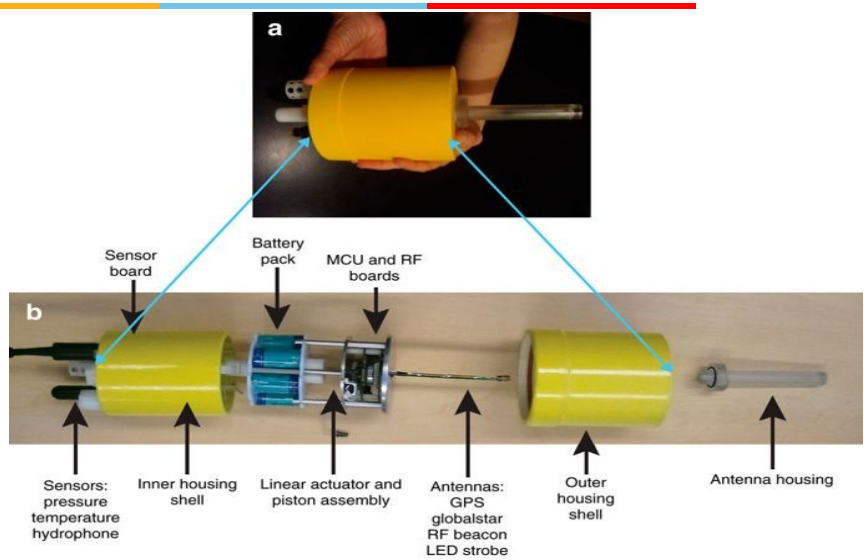
System Specifications

- Online Lake water quality monitoring system
- The system is made-up of a swarm of floating drogues
- Data collected is transferred to a base-station on land



Source: Anupama, K., et al. "Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks"
https://www.nature.com/articles/ncomms14189

innovate achieve lead



Source: https://www.nature.com/articles/ncomms14189

System Specifications

- Drogues are free-floating underwater devices that operate autonomously & collaborate through an acoustic underwater network
- Buoyancy controlled
- Acoustically tracked
- Equipped with sensors for data collection
- Part of an ad hoc network for relaying data to surface stations for analysis

Source: Anupama, K., et al. "Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks". https://www.nature.com/articles/ncomms14189

System Specifications

Specifications – Drogue Movement

- Buoyancy control - drogues to move to various depths
- Two concentric, syntactic foam cylinders forming inner and outer sleeves that can slide over each other to significantly increase or decrease the vehicle's volume
- Piston is used to create incremental changes in vehicle volume while fully submerged to produce the small changes in density necessary for the vehicle to regulate its depth.
- The total change in volume that is possible with a small piston is ±0.4% with 12,000 increments over the entire piston range.
- A geared motor whose shaft is equipped with an optical encoder rotates a threaded rod that controls the small piston.

Source: Anupama, K., et al. "Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks" https://www.nature.com/articles/ncomms14189

System Specifications

Specifications – Movement

- Maintaining a given depth requires an algorithm to change the vehicle's buoyancy when ambient currents or changes in density move it vertically.
- PID control algorithm
- These are used to actuate the piston to minimize the difference between the desired depth and the actual depth, with the goal of ultimately converging to the target depth.

Source: Anupama, K., et al. "Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks" https://www.nature.com/articles/ncomms14189

System Specifications

Specifications – Sensors

- Chlorophyll
- Dissolved O2
- pH
- Turbidity
- Temperature
- Salinity

Source: Anupama, K., et al. "Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks"

System Specifications

Specifications – Communications

- Data collected - transmitted acoustically using a self-organizing UAN to base station
- Formation& Maintenance of UAN - network protocol stack - resident in the flash memory- protocol stack is minimalistic

Source: Anupama, K., et al. "Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks"

Requirements

Requirements – Sensor Accuracy

- Chlorophyll – 0.5µg/l
- Dissolved O2 – 0.5 % 0 – 100 %
- pH – 1-14 resolution of 0.5
- Turbidity – 0- 50 Nephelometric Turbidity Units (NTU)
- Temperature - -50º C to 50º C accuracy 0.1º C
- Salinity 0 -2 Siemens/cm with accuracy of 0.5%
- TDS (mg/L) ≈Salinity (EC) x 0.55
- Output Preferably in Analog (0-5V) or (0-3.3V)

Source: Anupama, K., et al. "Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks"

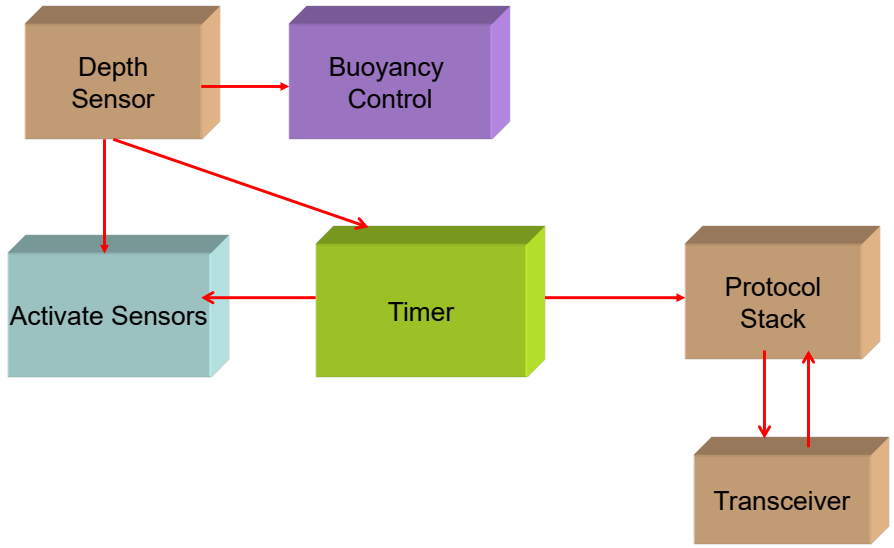
Requirements

Requirements Motors

- Lagrangian (current-following) vehicles
- Buoyancy Control
  - Bladder Mechanism
  - Servo Motor

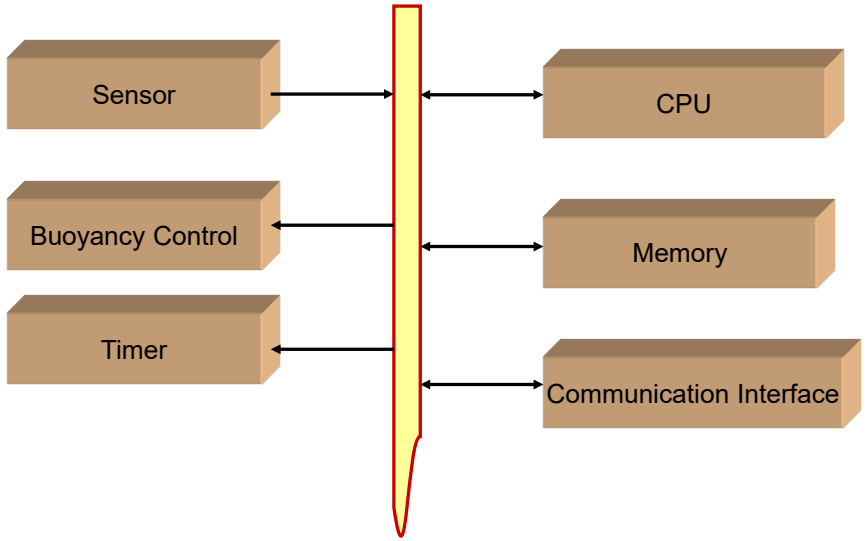
Source: Anupama, K., et al. "Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks"
https://www.nature.com/articles/ncomms14189

System Block Diagram



Source: Anupama, K., et al. "Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks"

Hardware Block Architecture



Source: Anupama, K., et al. "Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks"

Components: Sensors

Sensors

pH - WQ201 pH Sensors

- Submersible pH measurements
- Fully encapsulated electronics
- Marine grade cable with strain relief
- Stainless steel housing
- Replaceable pH element
- Output: 4-20 mA
- Range: 0-14 pH
- Accuracy: 2% of full scale
- Operating Voltage: 10-30 VDC

Source: http://www.globalw.com/downloads/WQ/WQ201B.pdf

Components: Sensors

Salinity - WQ-Cond

- 0.5% conductivity reading accuracy
- Dual 4-20 mA outputs, temperature and conductivity
- Measurement span from 0 to 2 siemens/cm
- Sensor armor available for harsh environments
- Measure conductivity up to 82 feet depth
- Fully encapsulated electronics

Source:http://www.globalw.com/downloads/WQ/WQ-cond.pdf

Components: Sensors

Salinity - WQ-Cond

- Conductivity Range:
  - 0 to 200 µS/cm
  - 200 to 2000 µS/cm
  - 2 to 20 mS/cm
  - 20 to 200 mS/cm
  - 200 to 2000 mS/cm
- Conductivity Resolution:
  - 0.1 µS/cm
  - 1 µS/cm
  - 0.01 mS/cm
  - 0.1 mS/cm
  - 1 mS/cm
- Output: Dual 4-20 mA (Conductivity and Temperature)

Source: http://www.globalw.com/downloads/WQ/WQ-cond.pdf

Components: Sensors

Dissolved O_2 - WQ401

- Fully encapsulated electronics
- Marine grade cable with strain relief
- Stainless steel housing
- Replaceable DO element
- Output: 4-20 mA
- Range: 0-100% Saturation, 0-8 ppm, temperature compensated to 77°F (25°C)
- Accuracy: ±0.5% of full scale

Source:http://www.globalw.com/products/wq401.html

Components: Sensors

Turbidity Sensors – WQ 730

- Simple and convenient to use
- Marine grade cable with strain relief
- Rugged stainless steel and Delrin sensor housing
- Range: Sensor=0-50 NTU and 0-1000 NTU; Meter=0-50 NTU or 0-1000 NTU selectable
- Accuracy: ± 1% of full scale
- Meter Resolution: 12 bit
- Output: 4-20mA

Source: http://www.globalw.com/products/turbidity.html

Components: Sensors

Temperature – WQ 101

- Fully encapsulated electronics
- Marine grade cable with strain relief
- Stainless steel housing
- Output: 4-20 mA
- Range: -58 to +122° F (-50 to +50°C)
- Accuracy: ±0.2°F or ±0.1°C

Source: http://www.globalw.com/products/wq101.html

Components: Sensors

Flourometers - for studies needing chlorophyll concentration data.

- Output Voltage 0-5.0 VDC
- Response Time 0.1 sec.
- Accuracy 0.02 µg/l

Source: Anupama, K., et al. “Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks”

- **Acoustic Modem**
- S2CM HS Modem – EvoLogics
- Data Rate - 62.5kbps
- Operating Depth – 200m
- BER- less than 10^{-10}
- Internal Data Buffer- 1 MB, configurable
- Host I/f - Ethernet or RS-232
- Power Consumption
  - Stand-by Mode 0.5 mW
  - Receive Mode 0.8 W
  - Transmit Mode 3.5 W, 200 m range, 10 W, max.
- Power Supply
  - External 24 VDC (12 VDC optional)

Source: https://www.evologics.de/files/DataSheets/EvoLogics_S2CM-HS-Product_Information.pdf

Sensors Connect

Sensors via ADC

- Current to Voltage Converter Module H8G6 – Converts 4- 20 mA to 0-3.3V
- Connected to ADC
- 6 Sensors – 6 Channels of ADC of LPC 2378 AD0- AD5

Source: Anupama, K., et al. "Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks"

Sensing Interval

Every 5 Minutes – Use Timer 0 - Match

Acoustic Modem

- Connected via Serial Communication interface

Serial Communication – LPC 23XX

- 16 byte Receive and Transmit FIFOs.
- Receiver FIFO trigger points at 1, 4, 8, and 14 bytes.
- Built-in baud rate generator.

Source: Anupama, K., et al. "Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks"

Movement Control

- Stepper Motors + Encoder – RS232C
- Depth Sensors – Bar 30 High resolution Depth upto 300m – with steps of 2mm - Connect via I^2C
- Stepper Driver DRV811
  - USM1 USM 0 – how much of the step
  - Step
  - Dir
  - A0
  - A1
  - B0
  - B1
  - $\frac{v\ x\ 320x\ 8}{60\ x\ angle/step}$

Source: http://www.ti.com/lit/ds/slvsc40g/slvsc40g.pdf

- The system is powered by a battery pack made from 6 Tenergy 1.2 V, 5000, mA hour NiMH cells in a series configuration.
- The battery pack is fused with a 2 A fuse and is recharged at an optimal rate of 600 mA.
- The system has a dedicated board for GPS, RF
- A u-blox6 GPS receiver is used with a Sarantel Helical GPS antenna (GeoHelix SL1300) to yield good localization near the water surface.
- Semtech SX1230 RF transmitter is programmed to send GPS coordinates from the unit over an RF link at 434 MHz with a usable range of 500–1,000 m.

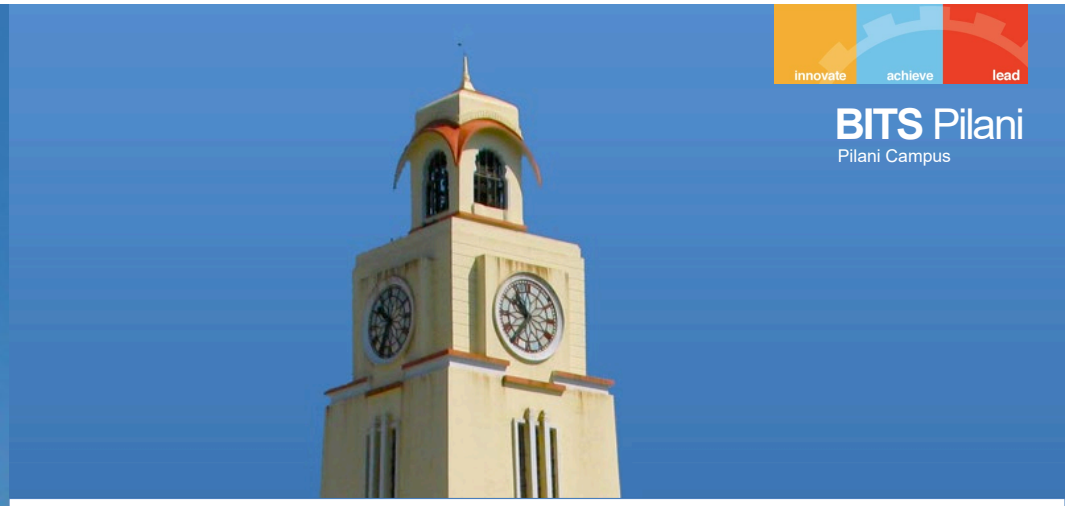
Source: Anupama, K., et al. “Multi-parameter ocean monitoring system using Underwater Wireless Sensor Networks”

Thank You.

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE

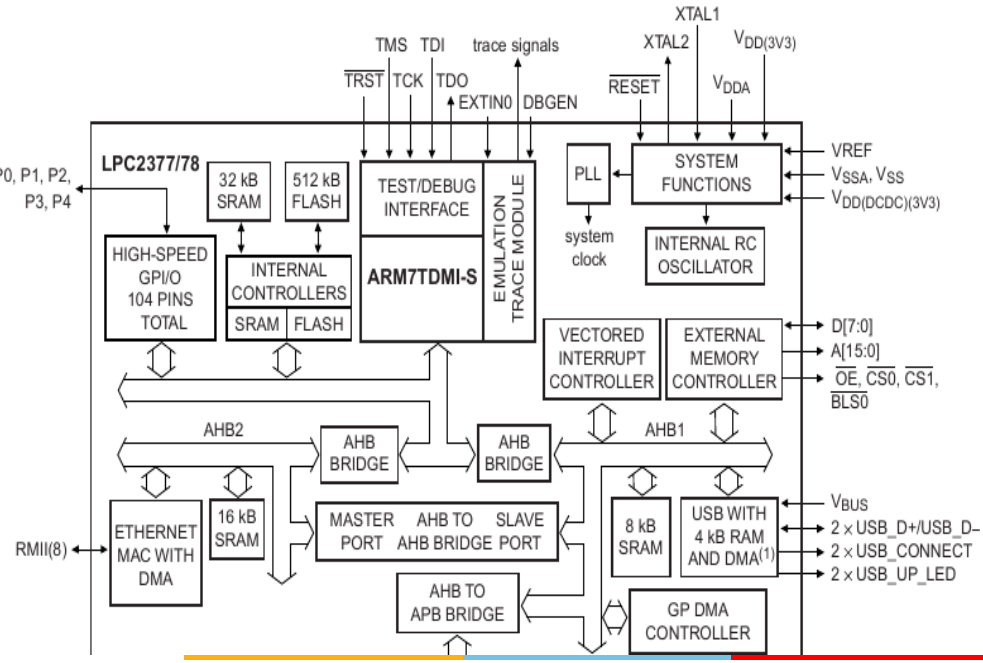


BITS Pilani
Pilani Campus

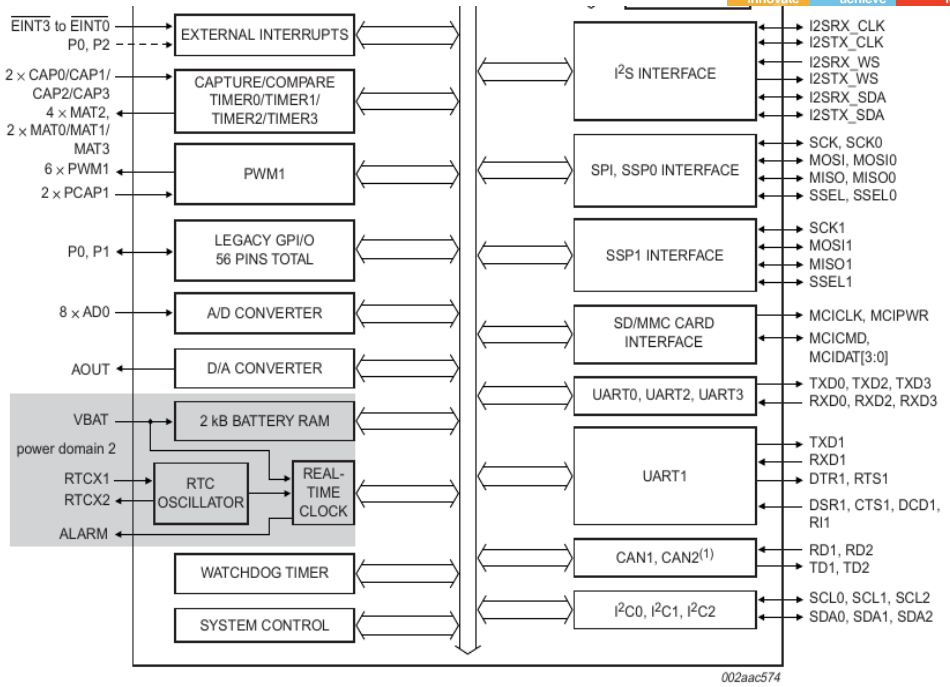
ESZG512 / MELZG526 / SEZG516
Embedded System Design
Contact Session 7

Contents

LPC2378 Block Diagram



LPC2378 Block Diagram



LPC23XX General Purpose Input / Output ports

Features:

- Digital I/O ports
  - ✓ GPIO PORT 0 and PORT 1 are ports accessible via either the group of registers providing enhanced features and accelerated port access or the legacy group of registers.
  - ✓ PORT 2 / 3 / 4 are accessed as fast ports only.
  - ✓ Accelerated GPIO functions:
    - GPIO registers are relocated to the ARM local bus so that the fastest possible I/O timing can be achieved.
    - Mask registers allow treating sets of port bits as a group, leaving other bits unchanged.
    - All GPIO registers are byte and half-word addressable.
    - The entire port value can be written in one instruction.

Source: LPC23XX User manual

LPC23XX General Purpose Input / Output ports

- Bit-level set and clear registers allow a single instruction set or clear of any number of bits in one port.
- Direction control of individual bits.
- All I/O default to inputs after reset.
- Backward compatibility with other earlier devices is maintained with legacy registers appearing at the original addresses on the APB bus.

- Interrupt generating digital ports
  - PORT0 and PORT2 provide an interrupt for each port pin.
  - Each interrupt can be programmed to generate an interrupt on a rising edge, a falling edge, or both.
  - Edge detection is asynchronous, so it may operate when clocks are not present, such as during Power-down mode.
  - With this feature, level-triggered interrupts are not needed.

Source: LPC23XX User manual

LPC23XX General Purpose Input / Output ports

- Each enabled interrupt contributes to a Wakeup signal that can be used to bring the part out of Power-down mode.
- Registers provide software a view of pending rising edge interrupts, pending falling edge interrupts, and overall pending GPIO interrupts.
- GPIO0 and GPIO2 interrupts share the same VIC slot with the External Interrupt 3 event.

- Applications
  - General purpose I/O.
  - Driving LEDs, or other indicators.
  - Controlling off-chip devices.
  - Sensing digital inputs, detecting edges.
  - Bringing the part out of Power Down mode.

Source: LPC23XX User manual

Total pins -112

- Port 2
  - Pin 14-31 NA
- Port 3
  - Pin 8-22 NA
  - Pin 27-31 NA
- Port 4
  - Pin 16-23 NA
  - Pin 26-27 NA

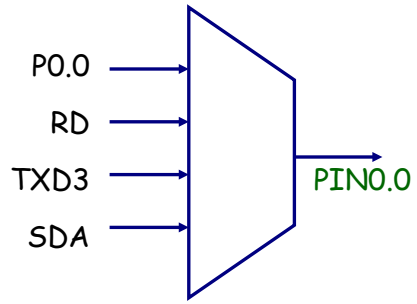
Port 0 & 1 How do they function ?

Use as GPIO/ Fast I/O - SCS Reg Bit 0 – GPIOM

Pin Select Registers

Every pin has alternate functions

00 – GPIO



Pin Mode Registers – Select Pull-up / Pull down

How to use GPIO?

Regs – Legacy – Only for Port 0/1: IOxPIN, IOxSET, IOxCLR, IOxDIR

Table 132. GPIO register map (legacy APB accessible registers)

Generic Name	Description	Access	Reset value[1]	PORTn Register Address & Name
IOPIN	GPIO Port Pin value register. The current state of the GPIO configured port pins can always be read from this register, regardless of pin direction. By writing to this register port's pins will be set to the desired level instantaneously.	R/W	NA	IO0PIN - 0xE002 8000 IO1PIN - 0xE002 8010
IOSET	GPIO Port Output Set register. This register controls the state of output pins in conjunction with the IOCLR register. Writing ones produces highs at the corresponding port pins. Writing zeroes has no effect.	R/W	0x0	IO0SET - 0xE002 8004 IO1SET - 0xE002 8014
IODIR	GPIO Port Direction control register. This register individually controls the direction of each port pin.	R/W	0x0	IO0DIR - 0xE002 8008 IO1DIR - 0xE002 8018
IOCLR	GPIO Port Output Clear register. This register controls the state of output pins. Writing ones produces lows at the corresponding port pins and clears the corresponding bits in the IOSET register. Writing zeroes has no effect.	WO	0x0	IO0CLR - 0xE002 800C IO1CLR - 0xE002 801C

Source: LPC23XX User manual

How to use FGPIO?

FIOxPIN
- FIOxPIN0-3
- FIOxPINL / FIOxPINU

FIOxSET
- FIOxSET0-3
- FIOxSETL / FIOxSETU

FIOxCLR
- FIOxCLR0-3
- FIOxCLR / FIOxCLRU

FIOxDIR
- FIOxDIR0-3
- FIOxDIRL / FIOxDIRU

FIOxMASK
- FIOxMASK0-3
- FIOxMASKL / FIOxMASKU

How to use FGPIO?

Table 133. GPIO register map (local bus accessible registers - enhanced GPIO features)

Generic Name	Description	Access	Reset value[1]	PORTn Register Address & Name
FIODIR	Fast GPIO Port Direction control register. This register individually controls the direction of each port pin.	R/W	0x0	FIO0DIR - 0x3FFF C000 FIO1DIR - 0x3FFF C020 FIO2DIR - 0x3FFF C040 FIO3DIR - 0x3FFF C060 FIO4DIR - 0x3FFF C080
FIOMASK	Fast Mask register for port. Writes, sets, clears, and reads to port (done via writes to FIOPIN, FIOSET, and FIOCLR, and reads of FIOPIN) alter or return only the bits enabled by zeros in this register.	R/W	0x0	FIO0MASK - 0x3FFF C010 FIO1MASK - 0x3FFF C030 FIO2MASK - 0x3FFF C050 FIO3MASK - 0x3FFF C070 FIO4MASK - 0x3FFF C090
FIOPIN	Fast Port Pin value register using FIOMASK. The current state of digital port pins can be read from this register, regardless of pin direction or alternate function selection (as long as pins are not configured as an input to ADC). The value read is masked by ANDing with inverted FIOMASK. Writing to this register places corresponding values in all bits enabled by zeros in FIOMASK. **Important:** if a FIOPIN register is read, its bit(s) masked with 1 in the FIOMASK register will be set to 0 regardless of the physical pin state.	R/W	0x0	FIO0PIN - 0x3FFF C014 FIO1PIN - 0x3FFF C034 FIO2PIN - 0x3FFF C054 FIO3PIN - 0x3FFF C074 FIO4PIN - 0x3FFF C094
FIOSET	Fast Port Output Set register using FIOMASK. This register controls the state of output pins. Writing 1s produces highs at the corresponding port pins. Writing 0s has no effect. Reading this register returns the current contents of the port output register. Only bits enabled by 0 in FIOMASK can be altered.	R/W	0x0	FIO0SET - 0x3FFF C018 FIO1SET - 0x3FFF C038 FIO2SET - 0x3FFF C058 FIO3SET - 0x3FFF C078 FIO4SET - 0x3FFF C098
FIOCLR	Fast Port Output Clear register using FIOMASK0. This register controls the state of output pins. Writing 1s produces lows at the corresponding port pins. Writing 0s has no effect. Only bits enabled by 0 in FIOMASK0 can be altered.	WO	0x0	FIO0CLR - 0x3FFF C01C FIO1CLR - 0x3FFF C03C FIO2CLR - 0x3FFF C05C FIO3CLR - 0x3FFF C07C FIO4CLR - 0x3FFF C09C

Source: LPC23XX User manual

Example 1:

sequential accesses to IOSET and IOCLR affecting the same GPIO pin/bit

In the example code:

IO0DIR = 0x0000 0080 ;pin P0.7 configured as output

IO0CLR = 0x0000 0080 ;P0.7 goes LOW

IO0SET = 0x0000 0080 ;P0.7 goes HIGH

IO0CLR = 0x0000 0080 ;P0.7 goes LOW

pin P0.7 is configured as an output (write to IO0DIR register).

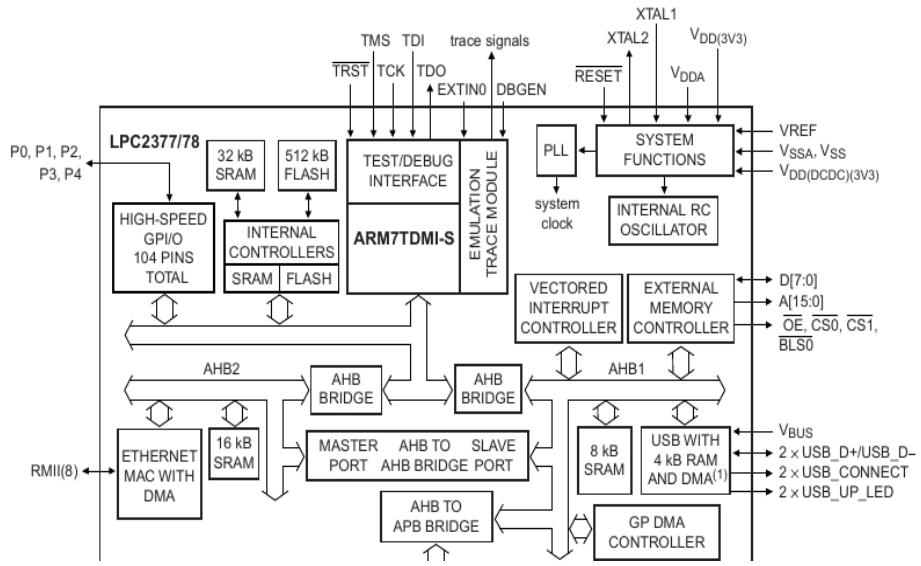
After this, P0.7 output is set to low (first write to IO0CLR register).

Short high pulse follows on P0.7 (write access to IO0SET), and

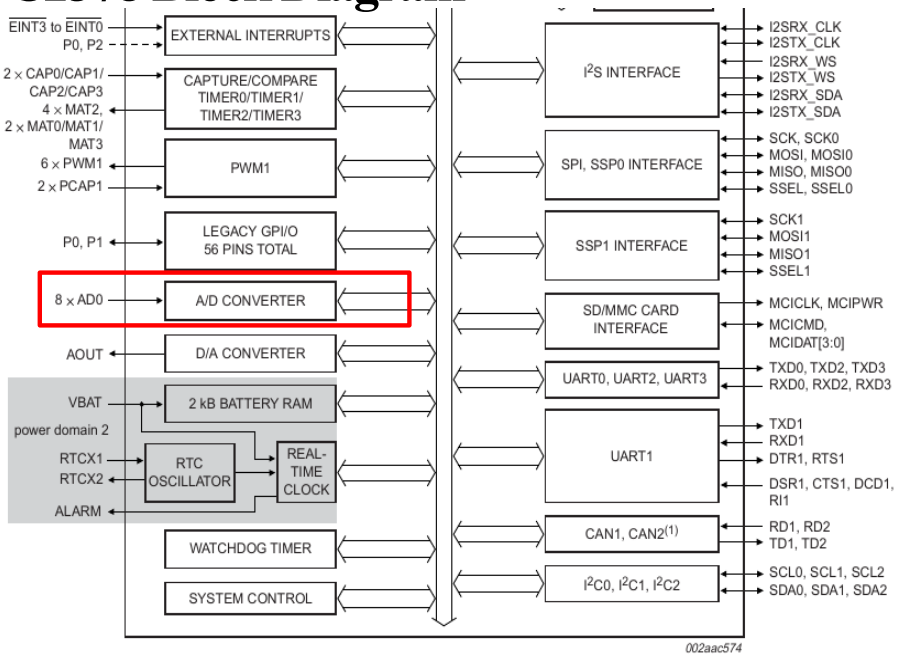
the final write to IO0CLR register sets pin P0.7 back to low level.

Source: LPC23XX User manual

LPC2378 Block Diagram



LPC2378 Block Diagram



LPC23XX Analog-to-Digital Converter (ADC)

Features:

- 10 bit successive approximation analog to digital converter.
- Input multiplexing among 6 pins or 8 pins.
- Power down mode.
- Measurement range 0 to 3 V.
- 10 bit conversion time ≥ 2.44 µs.
- Burst conversion mode for single or multiple inputs.
- Optional conversion on transition on input pin or Timer Match signal.
- Individual result registers for each A/D channel to reduce interrupt overhead.

Source: LPC23XX User manual

LPC23XX Analog-to-Digital Converter (ADC)

Pin	Type	Description
AD0[5:0] or AD0[7:0]	Input	**Analog Inputs.** The A/D converter cell can measure the voltage on any of these input signals. Note that these analog inputs are always connected to their pins, even if the Pin Multiplexing Register assigns them to port pins. A simple self-test of the A/D Converter can be done by driving these pins as port outputs. **Note:** while the ADC pins are specified as 5 V tolerant (see Section 8–1), the analog multiplexing in the ADC block is not. More than $V_{DD(3V3)}$/VREF/3.3 V (V_{DDA}) should not be applied to any pin that is selected as an ADC input, or the ADC reading will be incorrect. If for example AD0.0 and AD0.1 are used as the ADC0 inputs and voltage on AD0.0 = 4.5 V while AD0.1 = 2.5 V, an excessive voltage on the AD0.0 can cause an incorrect reading of the AD0.1, although the AD0.1 input voltage is within the right range. If the A/D converter is not used in an application then the pins associated with A/D inputs can be used as 5V tolerant digital IO pins
VREF	Reference	**Voltage Reference.** This pin provides a voltage reference level for the A/D converter.
V_{DDA}, V_{SSA}	Power	**Analog Power and Ground.** These should be nominally the same voltages as $V_{DD(3V3)}$ and V_{SS} respectively but should be isolated to minimize noise and error.

Remark: When the ADC is not used, the V_{DDA} and VREF pins must be connected to the power supply, and pin V_{SSA} must be grounded. These pins should **not** be left floating.

Basic clocking for the A/D converters is provided by the APB peripheral clock PCLK_ADC. A programmable divider is included in each converter to scale this clock to the 4.5 MHz (max) clock needed by the successive approximation process. A fully accurate conversion requires 11 of these clocks.

Source: LPC23XX User manual

Basic configuration

The ADC is configured using the following registers:

1. Power: In the PCONP register, set bits PCADC.
   Remark: On reset, the ADC is disabled. To enable the ADC, first set the PCADC bit, and then enable the ADC in the AD0CR register (bit PDN) Table 27–520. To disable the ADC, first clear the PDN bit, and then clear the PCADC bit.
2. Clock: In the PCLK_SEL0 register, select PCLK_ADC. To scale the clock for the ADC, see Table 27–520 bits CLKDIV.
3. Pins: Select ADC pins and pin modes in registers PINSELn and PINMODEn.
4. Interrupts: To enable interrupts in the ADC, see Table 27–523. Interrupts are enabled in the VIC using the VICIntEnable register.

Programming for ADC: Step.1.

Enable ADC by setting bit 12 in PCONP register

Table 56. Power Control for Peripherals register (PCONP - address 0xE01F C0C4) bit description

Bit	Symbol	Description	Reset value
0	-	Unused, always 0.	0
1	PCTIM0	Timer/Counter 0 power/clock control bit.	1
2	PCTIM1	Timer/Counter 1 power/clock control bit.	1
3	PCUART0	UART0 power/clock control bit.	1
4	PCUART1	UART1 power/clock control bit.	1
5	-	Unused, always 0.	1
6	PCPWM1	PWM1 power/clock control bit.	1
7	PCI2C0	The I^2C0 interface power/clock control bit.	1
8	PCSPI	The SPI interface power/clock control bit.	1
9	PCRTC	The RTC power/clock control bit.	1
10	PCSSP1	The SSP1 interface power/clock control bit.	1
11	PCEMC	External Memory Controller	1
12	PCAD	A/D converter (ADC) power/clock control bit. **Note:** Clear the PDN bit in the AD0CR (see Section 27–6.1) before clearing this bit, and set this bit before setting PDN.	0
13	PCAN1	CAN Controller 1 power/clock control bit.	0
14	PCAN2	CAN Controller 2 power/clock control bit.	0
18:15	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
19	PCI2C1	The I^2C1 interface power/clock control bit.	1

Programming for ADC: Step.2.

Configure the PINSEL registers to choose the ADC inputs

Table 108. Pin function select register 1 (PINSEL1 - address 0xE002 C004) bit description (LPC2377/78 and LPC2388)

PINSEL1	Pin name	Function when 00	Function when 01	Function when 10	Function when 11	Reset value
1:0	P0.16	GPIO Port 0.16	RXD1	SSEL0	SSEL	00
3:2	P0.17	GPIO Port 0.17	CTS1	MISO0	MISO	00
5:4	P0.18	GPIO Port 0.18	DCD1	MOSI0	MOSI	00
7:6	P0.19	GPIO Port 0.19	DSR1	MCICLK	SDA1	00
9:8	P0.20	GPIO Port 0.20	DTR1	MCICMD	SCL1	00
11:10	P0.21	GPIO Port 0.21	RI1	MCIPWR	RD1[1]	00
13:12	P0.22	GPIO Port 0.22	RTS1	MCIDAT0	TD1[1]	00
15:14	P0.23	GPIO Port 0.23	AD0.0	I2SRX_CLK	CAP3.0	00
17:16	P0.24	GPIO Port 0.24	AD0.1	I2SRX_WS	CAP3.1	00
19:18	P0.25	GPIO Port 0.25	AD0.2	I2SRX_SDA	TXD3	00
21:20	P0.26	GPIO Port 0.26	AD0.3	AOUT	RXD3	00
23:22	P0.27[2]	GPIO Port 0.27	SDA0	Reserved	Reserved	00
25:24	P0.28[2]	GPIO Port 0.28	SCL0	Reserved	Reserved	00

Programming for ADC: Step.3.

Configure the clock in PCLKSEL0 register

Table 49. Peripheral Clock Selection register 0 (PCLKSEL0 - address 0xE01F C1A8) bit description

Bit	Symbol	Description	Reset value
1:0	PCLK_WDT	Peripheral clock selection for WDT.	00
3:2	PCLK_TIMER0	Peripheral clock selection for TIMER0.	00
5:4	PCLK_TIMER1	Peripheral clock selection for TIMER1.	00
7:6	PCLK_UART0	Peripheral clock selection for UART0.	00
9:8	PCLK_UART1	Peripheral clock selection for UART1.	00
11:10	-	Unused, always read as 0.	00
13:12	PCLK_PWM1	Peripheral clock selection for PWM1.	00
15:14	PCLK_I2C0	Peripheral clock selection for I2C0.	00
17:16	PCLK_SPI	Peripheral clock selection for SPI.	00
19:18	PCLK_RTC[1]	Peripheral clock selection for RTC.	00
21:20	PCLK_SSP1	Peripheral clock selection for SSP1.	00
23:22	PCLK_DAC	Peripheral clock selection for DAC.	00
25:24	PCLK_ADC	Peripheral clock selection for ADC.	00
27:26	PCLK_CAN1	Peripheral clock selection for CAN1.	00
29:28	PCLK_CAN2	Peripheral clock selection for CAN2.	00
31:30	PCLK_ACF	Peripheral clock selection for CAN filtering.	00

Table 51. Peripheral Clock Selection register bit values

PCLKSEL0 and PCLKSEL1 individual peripheral's clock select options	Function	Reset value
00	PCLK_xyz = CCLK/4	00
01	PCLK_xyz = CCLK[1]	
10	PCLK_xyz = CCLK/2	
11	Peripheral's clock is selected to PCLK_xyz = CCLK/8 except for CAN1, CAN2, and CAN filtering when '11' selects PCLK_xyz = CCLK/6.	

Programming for ADC: Step.4.

Configure the Interrupt Enable Register

Table 523: A/D Interrupt Enable Register (AD0INTEN - address 0xE003 400C) bit description

Bit	Symbol	Description	Reset Value
7:0	ADINTEN 7:0	These bits allow control over which A/D channels generate interrupts for conversion completion. When bit 0 is one, completion of a conversion on A/D channel 0 will generate an interrupt, when bit 1 is one, completion of a conversion on A/D channel 1 will generate an interrupt, etc.	0x00
8	ADGINTEN	When 1, enables the global DONE flag in ADDR to generate an interrupt. When 0, only the individual A/D channels enabled by ADINTEN 7:0 will generate interrupts.	1
31:9	Unused	Unused, always 0.	0

Programming for ADC: Step.5.

Configure A/D control register

Select AD0.1 channel, clock for A/D is 4 MHz, burst mode, 10 bits, A/D converter is operational

Table 520: A/D Control Register (AD0CR - address 0xE003 4000) bit description

Bit	Symbol	Value	Description	Reset Value
7:0	SEL		Selects which of the AD0.7:0 pins is (are) to be sampled and converted. For AD0, bit 0 selects Pin AD0.0, and bit 7 selects pin AD0.7. In software-controlled mode, only one of these bits should be 1. In hardware scan mode, any value containing 1 to 8 ones. All zeroes is equivalent to 0x01.	0x01
15:8	CLKDIV		The APB clock (PCLK) is divided by (this value plus one) to produce the clock for the A/D converter, which should be less than or equal to 4.5 MHz. Typically, software should program the smallest value in this field that yields a clock of 4.5 MHz or slightly less, but in certain cases (such as a high-impedance analog source) a slower clock may be desirable.	0
16	BURST	0	Conversions are software controlled and require 11 clocks.	0
		1	The AD converter does repeated conversions at the rate selected by the CLKS field, scanning (if necessary) through the pins selected by 1s in the SEL field. The first conversion after the start corresponds to the least-significant 1 in the SEL field, then higher numbered 1 bits (pins) if applicable. Repeated conversions can be terminated by clearing this bit, but the conversion that's in progress when this bit is cleared will be completed. **Important:** START bits must be 000 when BURST = 1 or conversions will not start.	

Table 520: A/D Control Register (AD0CR - address 0xE003 4000) bit description

Bit	Symbol	Value	Description	Reset Value
19:17	CLKS		This field selects the number of clocks used for each conversion in Burst mode, and the number of bits of accuracy of the result in the LS bits of ADDR, between 11 clocks (10 bits) and 4 clocks (3 bits).	000
		000	11 clocks / 10 bits	
		001	10 clocks / 9 bits	
		010	9 clocks / 8 bits	
		011	8 clocks / 7 bits	
		100	7 clocks / 6 bits	
		101	6 clocks / 5 bits	
		110	5 clocks / 4 bits	
		111	4 clocks / 3 bits	
20			Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
21	PDN	1	The A/D converter is operational.	0
		0	The A/D converter is in power-down mode.	
23:22	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
26:24	START		When the BURST bit is 0, these bits control whether and when an A/D conversion is started:	0
		000	No start (this value should be used when clearing PDN to 0).	
		001	Start conversion now.	
		010	Start conversion when the edge selected by bit 27 occurs on P2.10/EINT0.	
		011	Start conversion when the edge selected by bit 27 occurs on P1.27/CAP0.1.	
		100	Start conversion when the edge selected by bit 27 occurs on MAT0.1.	
		101	Start conversion when the edge selected by bit 27 occurs on MAT0.3[1].	
		110	Start conversion when the edge selected by bit 27 occurs on MAT1.0.	
		111	Start conversion when the edge selected by bit 27 occurs on MAT1.1.	
27	EDGE		This bit is significant only when the START field contains 010-111. In these cases:	0
		1	Start conversion on a falling edge on the selected CAP/MAT signal.	
		0	Start conversion on a rising edge on the selected CAP/MAT signal.	
31:28	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Programming for ADC: Step.6.

In the ISR, when an interrupt is received, check the ADC status register.

Table 522: A/D Status Register (AD0STAT - address 0xE003 4030) bit description

Bit	Symbol	Description	Reset Value
7:0	Done7:0	These bits mirror the DONE status flags that appear in the result register for each A/D channel.	0
15:8	Overrun7:0	These bits mirror the OVERRRUN status flags that appear in the result register for each A/D channel. Reading ADSTAT allows checking the status of all A/D channels simultaneously.	0
16	ADINT	This bit is the A/D interrupt flag. It is one when any of the individual A/D channel Done flags is asserted and enabled to contribute to the A/D interrupt via the ADINTEN register.	0
31:17	Unused	Unused, always 0.	0

Programming for ADC: Step.7.

- After finding which channel(s) completed conversion, read the ADC data register. Each channel has a separate data register.

Table 524: A/D Data Registers (AD0DR0 to AD0DR7 - addresses 0xE003 4010 to 0xE003 402C) bit description

Bit	Symbol	Description	Reset Value
5:0	Unused	Unused, always 0. These bits always read as zeroes. They provide compatible expansion room for future, higher-resolution ADCs.	0
15:6	V/V_{REF}	When DONE is 1, this field contains a binary fraction representing the voltage on the Ain pin, divided by the voltage on the Vref pin. Zero in the field indicates that the voltage on the Ain pin was less than, equal to, or close to that on V_{REF}, while 0x3FF indicates that the voltage on Ain was close to, equal to, or greater than that on Vref.	NA
29:16	Unused	These bits always read as zeroes. They allow accumulation of successive A/D values without AND-masking, for at least 256 values without overflow into the CHN field.	0
30	OVERRUN	This bit is 1 in burst mode if the results of one or more conversions was (were) lost and overwritten before the conversion that produced the result in the LS bits.This bit is cleared by reading this register.	0
31	DONE	This bit is set to 1 when an A/D conversion completes. It is cleared when this register is read.	0

Interfacing of LPC 2XXX with 16x2 LCD

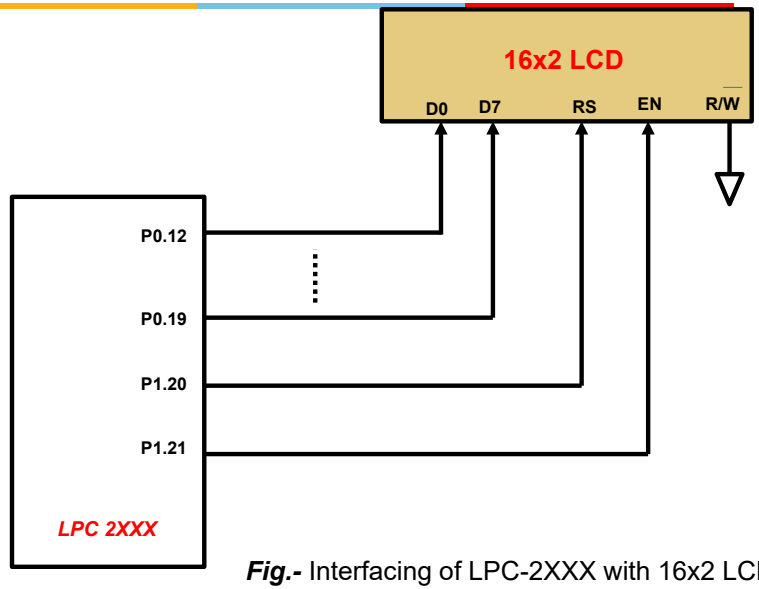


Fig.- Interfacing of LPC-2XXX with 16x2 LCD

Source: LPC23XX User manual

Interfacing of LPC 2XXX with 16x2 LCD

How to set a Register

0=Input
1=Output

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0				0				0				F				F				0				0				0			

Data Pins P0.12 to Port P0.19 as a Output Pin to LCD D0-D7

IODIR0= 0x000FF000

Source: LPC23XX User manual

Interfacing of LPC 2XXX with 16x2 LCD

How to set a Register

0=Input
1=Output

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0				0				3				0				0				0				0				0			

Port P1.20 and P1.21 as RS and EN and as a O/P pin

IODIR1= 0x00300000

Interfacing of LPC 2XXX with 16x2 LCD

Pin Number	Symbol	Pin Function
1	VSS	Ground
2	VCC	+5v
3	VEE	Contrast adjustment (VO)
4	RS	Register Select. 0:Command, 1: Data
5	R/W	Read/Write, R/W=0: Write & R/W=1: Read
6	EN	Enable. Falling edge triggered
7	D0	Data Bit 0 (Not used in 4-bit operation)
8	D1	Data Bit 1 (Not used in 4-bit operation)
9	D2	Data Bit 2 (Not used in 4-bit operation)
10	D3	Data Bit 3 (Not used in 4-bit operation)
11	D4	Data Bit 4
12	D5	Data Bit 5
13	D6	Data Bit 6
14	D7	Data Bit 7/Busy Flag
15	A/LED+	Back-light Anode(+)
16	K/LED-	Back-Light Cathode(-)

Interfacing of LPC 2XXX with 16x2 LCD

- **Data Bus:** As shown in the above figure and table, an alpha numeric lcd has a 8-bit data bus referenced as D0-D7. As it is a 8-bit data bus, we can send the data/cmd to LCD in bytes. It also provides the provision to send the the data/cmd in chunks of 4-bit, which is used when there are limited number of GPIO lines on the microcontroller.

- **Register Select(RS):** The LCD has two register namely a Data register and Command register. Any data that needs to be displayed on the LCD has to be written to the data register of LCD. Command can be issued to LCD by writing it to Command register of LCD. This signal is used to differentiate the data/cmd received by the LCD.
- If the RS signal is **LOW** then the LCD interprets the 8-bit info as **Command** and writes it **Command register** and performs the action as per the command. If the RS signal is **HIGH** then the LCD interprets the 8-bit info as **data** and copies it to **data register**. After that the LCD decodes the data for generating the 5x7 pattern and finally displays on the LCD.

Interfacing of LPC 2XXX with 16x2 LCD

- **Read/Write(RW):** This signal is used to write the data/cmd to LCD and reads the busy flag of LCD. For write operation the RW should be **LOW** and for read operation the R/W should be **HIGH**.
- **Enable(EN):** This pin is used to send the enable trigger to LCD. After sending the data/cmd, Selecting the data/cmd register, Selecting the Write operation. A HIGH-to-LOW pulse has to be send on this enable pin which will latch the info into the LCD register and triggers the LCD to act accordingly.

Interfacing of LPC 2XXX with 16x2 LCD

LCD Interfacing:

The RS pin is used to select one of these registers.

- RS (Register Select): ***If '1', data register is selected.***
  If '0', command register is selected.

The R/w input allows the user to read or write information to LCD.

- R/W (Read/write): ***while reading ,R/W='1'.***
  while writing, R/W='0'.

Interfacing of LPC 2XXX with 16x2 LCD

LCD Interfacing:

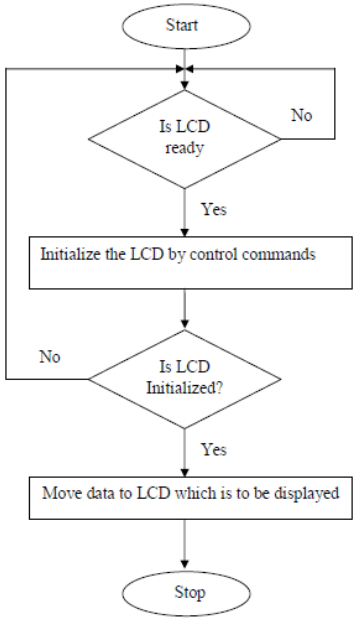
- ✓ The (E) Enable pin is used by the LCD to latch the information presented to its data pins.
- ✓ A high to low pulse should be applied to this pin in order for the LCD to latch in the data presented at data pins.
- ✓ The 8-bit data pins, D0-D7, are used to send information to LCD or read the contents of the LCD's internal registers.

Interfacing of LPC 2XXX with 16x2 LCD

LCD Commands

Hex Value	Functions
1	Clear the display
2	Return home
4	Decrement cursor(shift cursor to left)
6	Increment cursor(shift cursor to right)
5	Shift display right
7	Shift display left
8	Display off, Cursor off
0A	Display off, Cursor on
0C	Display on, Cursor off
0E	Display on, Cursor blinking
10	Shift cursor position to left
14	Shift cursor position to right
18	Shift entire display to left
1C	Shift entire display to right
80	Force cursor to beginning of 1st line
C0	Force cursor to beginning of 2nd line
38	2 lines and 5X7 matrix

Interfacing of LPC 2XXX with 16x2 LCD



Interfacing of LPC 2XXX with 16x2 LCD

LCD Interfacing Details

Description	Signal
LCD	LPC2xxx
LCD D0	P0.12
LCD D1	P0.13
LCD D2	P0.14
LCD D3	P0.15
LCD D4	P0.16
LCD D5	P0.17
LCD D6	P0.18
LCD D7	P0.19
LCD RS	P1.20
LCD EN	P1.21

Interfacing of LPC 2XXX with 16x2 LCD

IODIR0=0x000FF000; //Port 0.0 from pin 12-19

IODIR1=0x00300000; // port 1 pin 20=rs,21=en

Command	Data Mode
• RS=0	• RS=1
• Command	• Data
• En=1	• En=1
• DELAY	• DELAY
• En=0	• En=0

Interfacing of LPC 2XXX with 16x2 LCD

Command Mode:

IOCLR0=0X000FF000;

RS=0; ***IOCLR1=0X00100000;***

cmddata=cmddata<<12;

IOSET0=cmddata;

Send Command

EN=High (1) ***IOSET1=0x00200000;***

DELAY

EN=Low(0)

IOCLR1=0x00200000;

Interfacing of LPC 2XXX with 16x2 LCD

Data Mode:

IOCLR0=0X000FF000;

RS=1; *IOSET1=0X00100000;*

outdata=outdata<<12;

IOSET0=outdata;

Send Data

EN=High (1) *IOSET1=0x00200000;*

DELAY

EN=Low(0) *IOCLR1=0x00200000;*

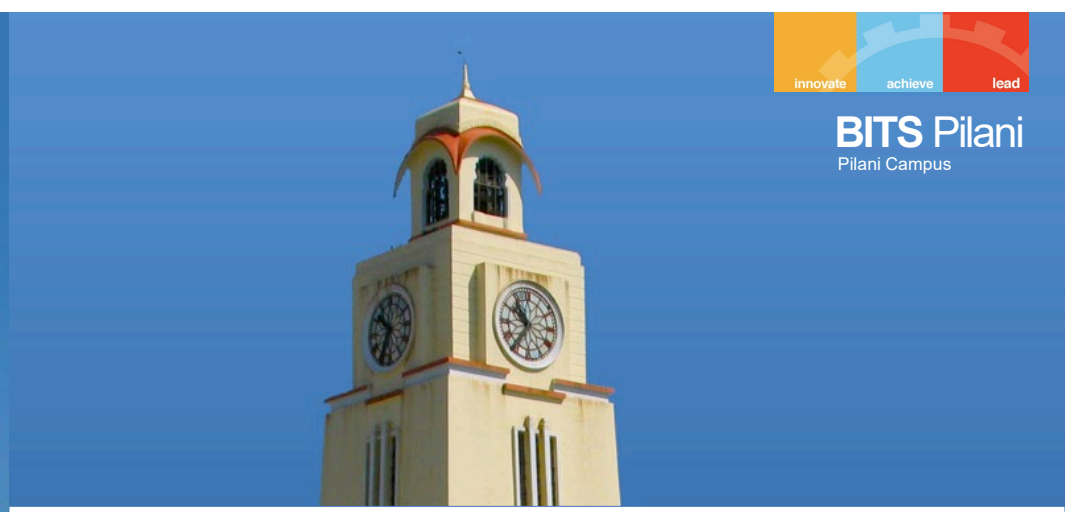
Thank you

BITS Pilani, Pilani Campus

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE



innovate achieve lead

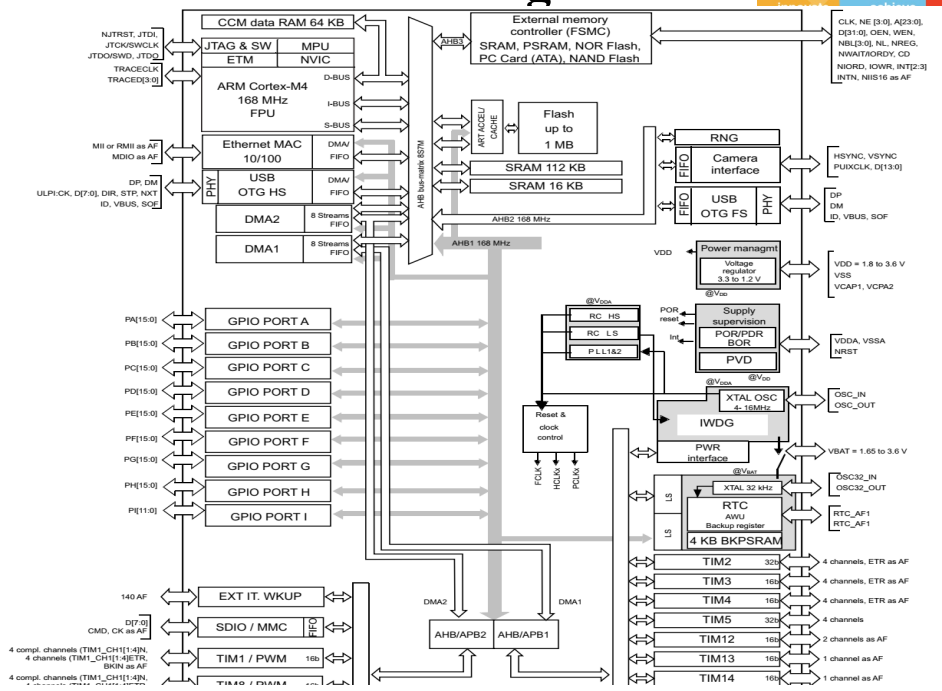
BITS Pilani
Pilani Campus

ESZG512/MELZG526/SEZG516
Embedded System Design
Contact Session 8

innovate achieve lead

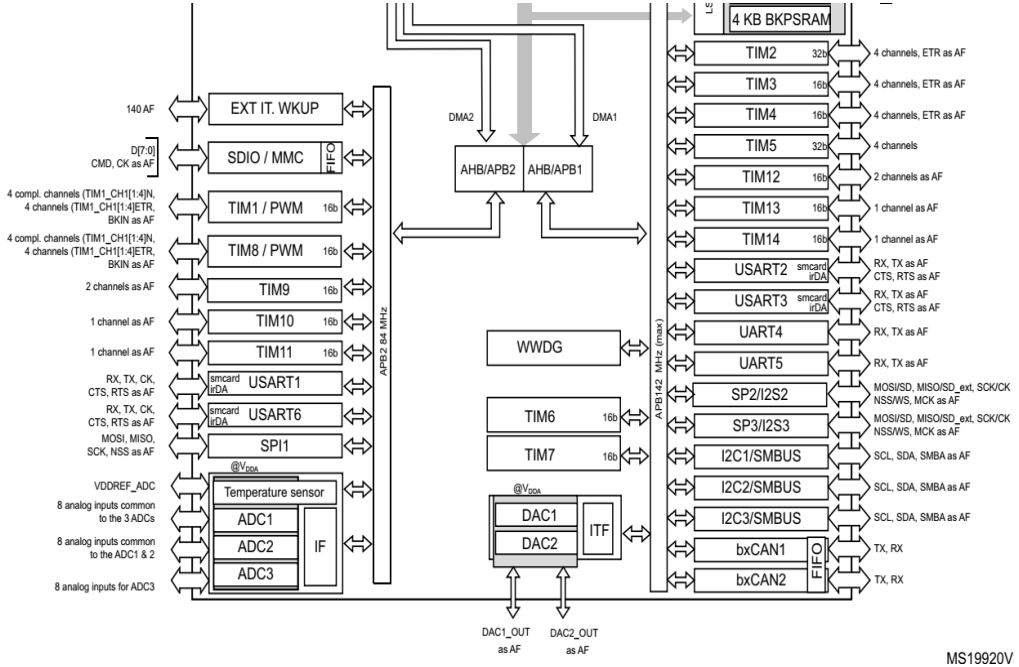
Contents

STM32F40xxx block diagram



Source: STM32F407xx Datasheet

STM32F40xxx block diagram



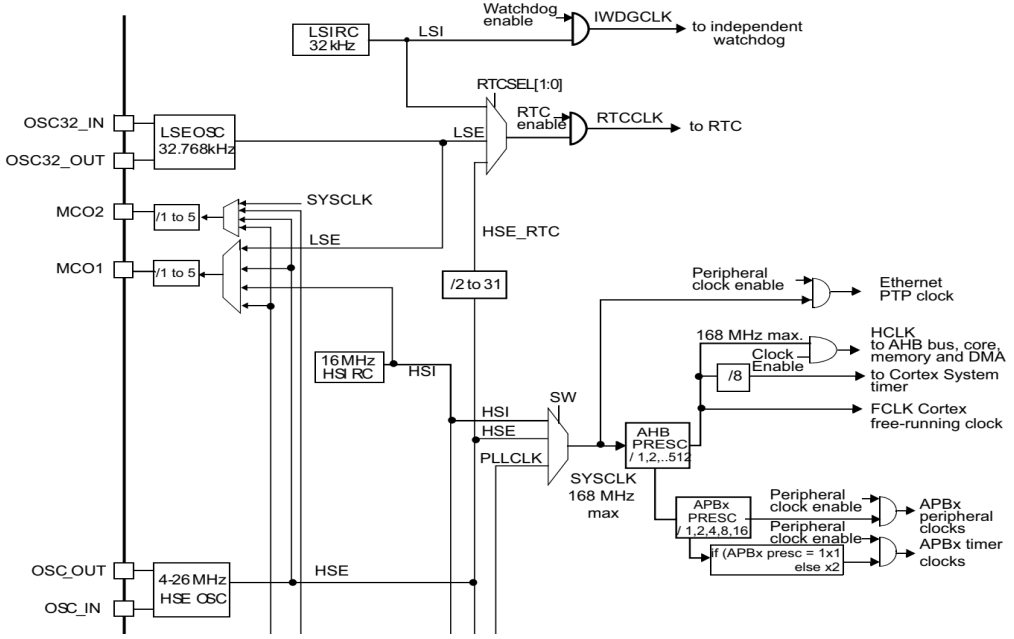
Source: STM32F407xx Datasheet

Clocks

- Three different clock sources can be used to drive the system clock (SYSCLK):
  - HSI oscillator clock
  - HSE oscillator clock
  - Main PLL (PLL) clock
- The devices have the two following secondary clock sources:
  - 32 kHz low-speed internal RC (LSI RC) which drives the independent watchdog and, optionally, the RTC used for Auto-wakeup from the Stop/Standby mode.
  - 32.768 kHz low-speed external crystal (LSE crystal) which optionally drives the RTC clock (RTCCLK)
- Each clock source can be switched ON or OFF independently when it is not used, to optimize power consumption.

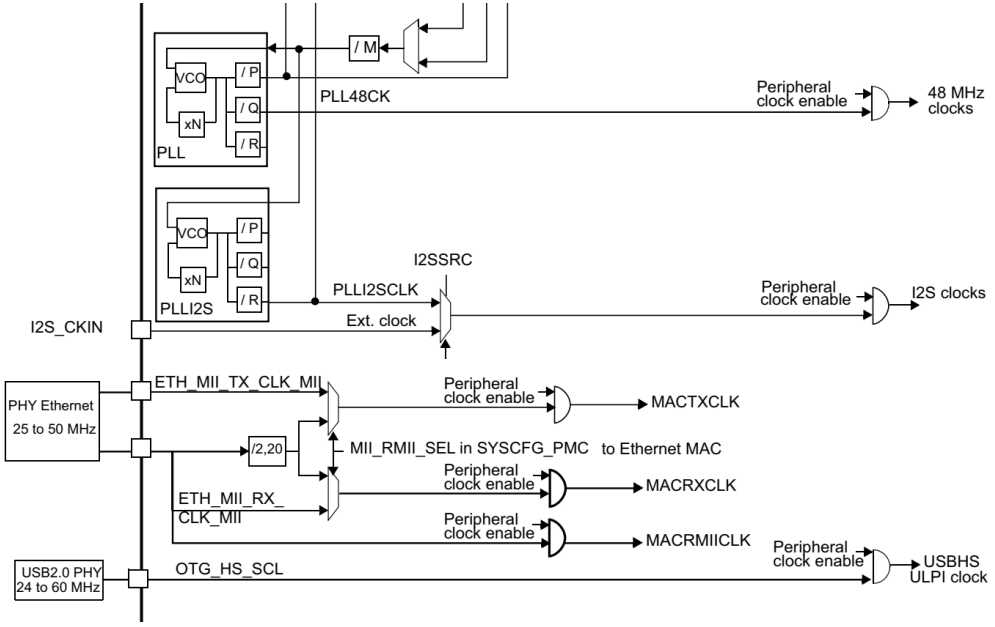
Source: STM32F407_417 Reference Manual RM0090

Clock tree



Source: STM32F407_417 Reference Manual RM0090

Clock tree



Source: STM32F407_417 Reference Manual RM0090

Clock tree

- The clock tree configuration is performed through a dedicated peripheral named Reset and Clock Control (RCC), and it is a process essentially composed by three steps:

1. The high-speed oscillator source is selected (HSI or HSE) and properly configured, if the HSE is used.
2. If we want to feed the SYSCLK with a frequency higher than the one provided by the high speed oscillator, then we need to configure the main PLL (which provides the PLLCLK signal). Otherwise we can skip this step.
3. The System Clock Switch (SW) is configured choosing the right clock source (HSI, HSE, or PLLCLK). Then we select the right AHB, APB1 and APB2 (if available) prescaler settings to reach the wanted frequency of the High-speed clock (HCLK - that is the one that feeds the core, DMAs and AHB bus), and the frequencies of Advanced Peripheral Bus 1 (APB1) and APB2 buses.

Reset and Clock Control

RCC PLL configuration register (RCC_PLLCFGR)

This register is used to configure the PLL clock outputs according to the formulas:

- $f_{(VCO\ clock)} = f_{(PLL\ clock\ input)} \times (PLLN / PLLM)$
- $f_{(PLL\ general\ clock\ output)} = f_{(VCO\ clock)} / PLLP$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				PLLQ3	PLLQ2	PLLQ1	PLLQ0	Reserved	PLLSRC	Reserved				PLLP1	PLLP0
				rw	rw	rw	rw		rw					rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	PLLN									PLLM5	PLLM4	PLLM3	PLLM2	PLLM1	PLLM0
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Source: STM32F407_417 Reference Manual RM0090

Reset and Clock Control

Bits 5:0 **PLLM:** Division factor for the main PLL (PLL) and audio PLL (PLLI2S) input clock

Set and cleared by software to divide the PLL and PLLI2S input clock before the VCO. These bits can be written only when the PLL and PLLI2S are disabled.

Caution: The software has to set these bits correctly to ensure that the VCO input frequency ranges from 1 to 2 MHz. It is recommended to select a frequency of 2 MHz to limit PLL jitter.

VCO input frequency = PLL input clock frequency / PLLM with $2 \leq PLLM \leq 63$

000000: PLLM = 0, wrong configuration
000001: PLLM = 1, wrong configuration
000010: PLLM = 2
000011: PLLM = 3
000100: PLLM = 4
...
111110: PLLM = 62
111111: PLLM = 63

Source: STM32F407_417 Reference Manual RM0090

Reset and Clock Control

Bits 14:6 **PLLN:** Main PLL (PLL) multiplication factor for VCO

Set and cleared by software to control the multiplication factor of the VCO. These bits can be written only when PLL is disabled. Only half-word and word accesses are allowed to write these bits.

Caution: The software has to set these bits correctly to ensure that the VCO output frequency is between 100 and 432 MHz.

VCO output frequency = VCO input frequency × PLLN with $50 \leq PLLN \leq 432$

000000000: PLLN = 0, wrong configuration
000000001: PLLN = 1, wrong configuration
...
000110010: PLLN = 50
...
001100011: PLLN = 99
001100100: PLLN = 100
...
110110000: PLLN = 432
110110001: PLLN = 433, wrong configuration
...
111111111: PLLN = 511, wrong configuration

Source: STM32F407_417 Reference Manual RM0090

Reset and Clock Control

Bits 17:16 **PLLP:** Main PLL (PLL) division factor for main system clock

Set and cleared by software to control the frequency of the general PLL output clock. These bits can be written only if PLL is disabled.

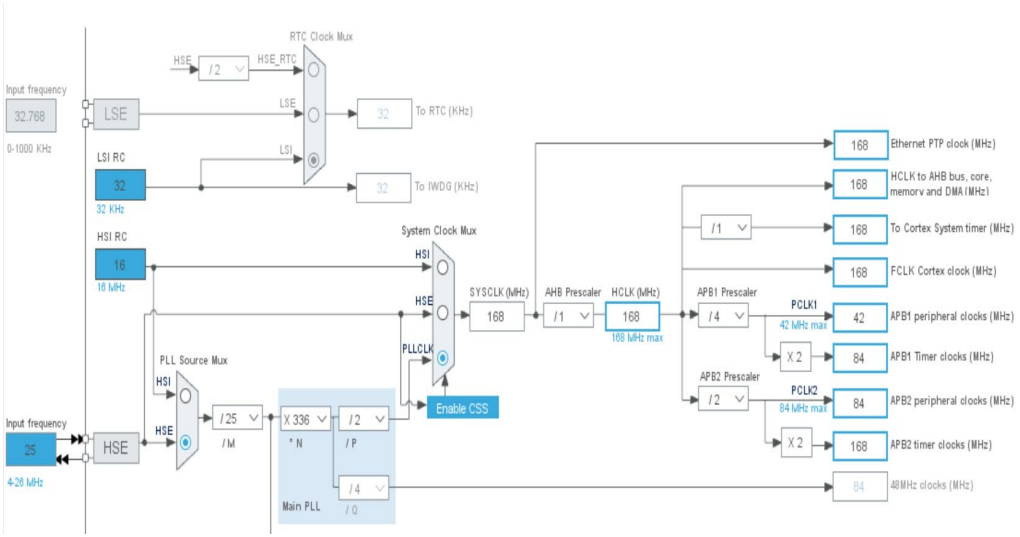
Caution: The software has to set these bits correctly not to exceed 168 MHz on this domain.

PLL output clock frequency = VCO frequency / PLLP with PLLP = 2, 4, 6, or 8

00: PLLP = 2
01: PLLP = 4
10: PLLP = 6
11: PLLP = 8

Source: STM32F407_417 Reference Manual RM0090

Clock Tree



Source: STM32CubeMX

General-purpose I/Os (GPIO)

Each general-purpose I/O port has

- Four 32-bit configuration registers
  (GPIOx_MODER, GPIOx_OTYPER, GPIOx_OSPEEDR and GPIOx_PUPDR),
- Two 32-bit data registers
  (GPIOx_IDR and GPIOx_ODR),
- Single 32-bit set/reset register (GPIOx_BSRR),
- Single 32-bit locking register (GPIOx_LCKR)
- Two 32-bit alternate function selection register
  GPIOx_AFRH and GPIOx_AFRL)

Source: STM32F407_417 Reference Manual RM0090

General-purpose I/Os (GPIO)

GPIO main features

- Up to 16 I/Os under control
- Output states: push-pull or open drain + pull-up/down
- Output data from output data register (GPIOx_ODR) or peripheral (alternate function output)
- Speed selection for each I/O
- Input states: floating, pull-up/down, analog
- Input data to input data register (GPIOx_IDR) or peripheral (alternate function input)
- Bit set and reset register (GPIOx_BSRR) for bitwise write access to GPIOx_ODR
- Locking mechanism (GPIOx_LCKR) provided to freeze the I/O configuration
- Analog function
- Alternate function input/output selection registers (at most 16 AFs per I/O)
- Fast toggle capable of changing every two clock cycles
- Highly flexible pin multiplexing allows the use of I/O pins as GPIOs or as one of several peripheral functions

Source: STM32F407_417 Reference Manual RM0090

General-purpose I/Os (GPIO)

- Each port bit of the general-purpose I/O (GPIO) ports can be individually configured by software in several modes:
  - Input floating
  - Input pull-up
  - Input-pull-down
  - Analog
  - Output open-drain with pull-up or pull-down capability
  - Output push-pull with pull-up or pull-down capability
  - Alternate function push-pull with pull-up or pull-down capability
  - Alternate function open-drain with pull-up or pull-down capability

Source: STM32F407_417 Reference Manual RM0090

General-purpose I/Os (GPIO)

GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Source: STM32F407_417 Reference Manual RM0090

General-purpose I/Os (GPIO)

GPIO port output type register (GPIOx_OTYPER)
(x = A..I/J/K)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

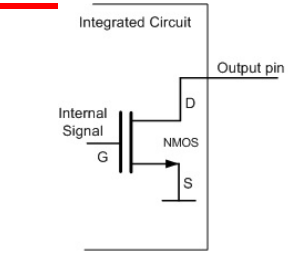
Bits 31:16 Reserved, must be kept at reset value.

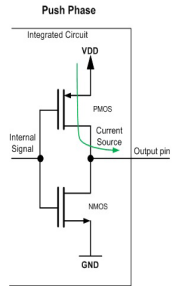
Bits 15:0 **OTy**: Port x configuration bits (y = 0..15)
These bits are written by software to configure the output type of the I/O port.
0: Output push-pull (reset state)
1: Output open-drain

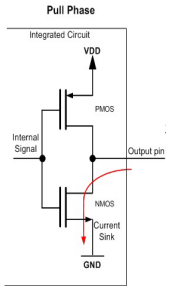
Source: STM32F407_417 Reference Manual RM0090

Open Drain Vs Push-Pull

- In Open Drain Type: If you make the output pin HIGH the pin will be connected to ground through the switch and if you make the Output pin LOW the pin will be left floating since the switch will be turned off.
- In Push-Pull Type: If you make the output HIGH the pin will be connected to Vdd through the NPN switch and if you make the Output pin LOW the pin will be connected to Ground thorough the PNP switch.
- Since in the Open Drain type the pin is left as floating a pull-high or pull-down resistor is usually added to the GPIO pin. But in modern day MCUs most of the GPIO output pins are Push-Pull type







Source: STM32F407_417 Reference Manual RM0090

General-purpose I/Os (GPIO)

GPIO port output data register (GPIOx_ODR) (x = A..I/J/K)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy**: Port output data (y = 0..15)

These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (x = A..I/J/K).

Source: STM32F407_417 Reference Manual RM0090

General-purpose I/Os (GPIO)

GPIO port bit set/reset register (GPIOx_BSRR) (x = A..I/J/K)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Bits 31:16 **BRy:** Port x reset bit y (y = 0..15)

These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

1: Resets the corresponding ODRx bit

Note: If both BSx and BRx are set, BSx has priority.

Bits 15:0 **BSy:** Port x set bit y (y= 0..15)

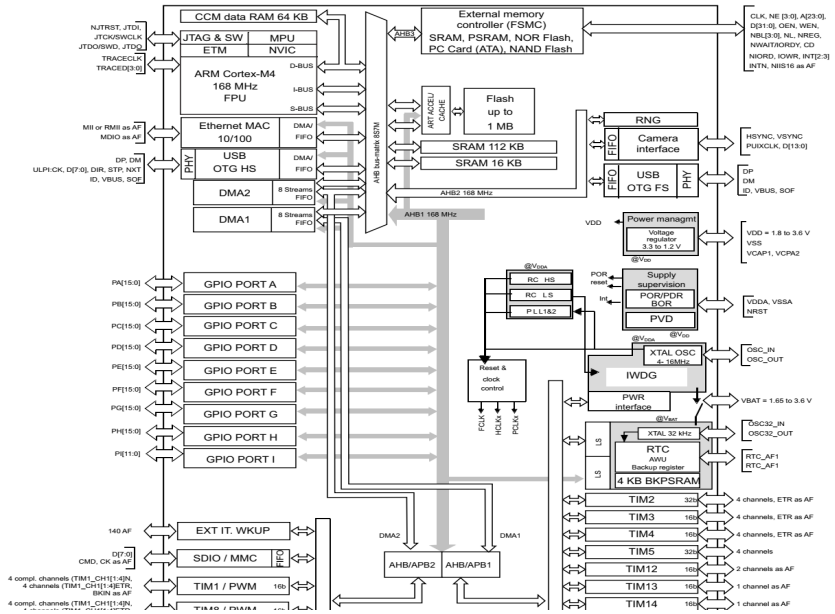
These bits are write-only and can be accessed in word, half-word or byte mode. A read to these bits returns the value 0x0000.

0: No action on the corresponding ODRx bit

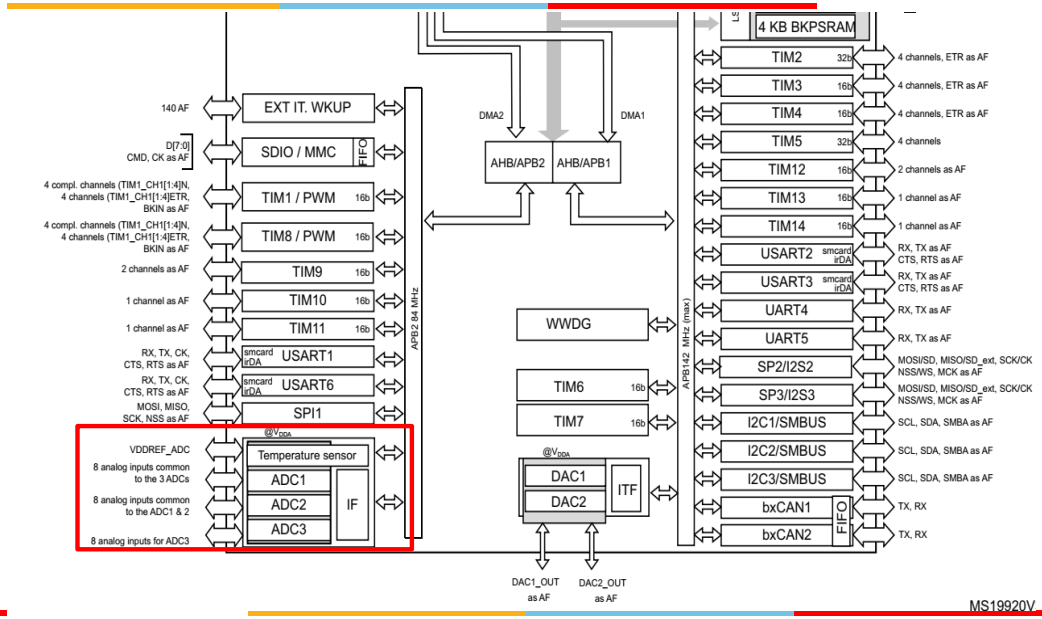
1: Sets the corresponding ODRx bit

Source: STM32F407_417 Reference Manual RM0090

STM32F40xxx block diagram



STM32F40xxx block diagram



Analog-to-digital converter (ADC)

- The 12-bit ADC is a successive approximation analog-to-digital converter.
- It has up to 19 multiplexed channels allowing it to measure signals from 16 external sources, two internal sources, and the VBAT channel.
- The A/D conversion of the channels can be performed in single, continuous, scan or discontinuous mode.
- The result of the ADC is stored into a left- or right-aligned 16-bit data register.

ADC main features

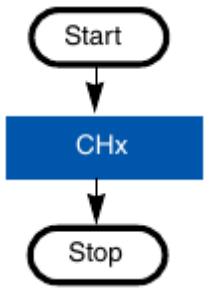
- 12-bit, 10-bit, 8-bit or 6-bit configurable resolution
- Interrupt generation at the end of conversion, end of injected conversion, and in case of analog watchdog or overrun events
- Single and continuous conversion modes
- Scan mode for automatic conversion of channel 0 to channel ‘n’
- Data alignment with in-built data coherency
- Channel-wise programmable sampling time
- External trigger option with configurable polarity for both regular and injected conversions
- Discontinuous mode

ADC main features

- Dual/Triple mode (on devices with 2 ADCs or more)
- Configurable DMA data storage in Dual/Triple ADC mode
- Configurable delay between conversions in Dual/Triple interleaved mode
- ADC conversion type (refer to the datasheets)
- ADC supply requirements: 2.4 V to 3.6 V at full speed and down to 1.8 V at slower speed
- ADC input range: VREF– ≤ VIN ≤ VREF+
- DMA request generation during regular channel conversion

Independent modes

- **Single-channel, single conversion mode**
- This is the simplest ADC mode. In this mode, the ADC performs the single conversion (single sample) of a single channel x and stops after completion of the conversion.

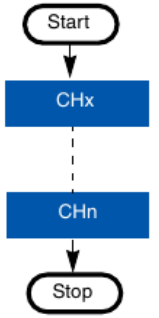


Independent modes

- Example of application
- This mode can be used for the measurement of a voltage level to decide if the system can be started or not. Measure the voltage level of the battery before starting the system: if the battery has a low level, the “low battery” message appears. In this case, do not start the system.

Independent modes

- **Multichannel (scan), single conversion mode**
- This mode is used to convert some channels successively in independent mode. With the ADC sequencer, you can use this ADC mode to configure any sequence of up to 16 channels successively with different sampling times and in different orders.

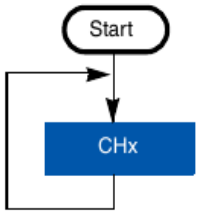


Independent modes

- Example of application
- This mode can be used when starting a system depends on some parameters like knowing the coordinates of the arm’s tip in a manipulator arm system. In this case, you have to read the position of each articulation in the manipulator arm system at power-on to determine the coordinates of the arm’s tip.
- This mode can also be used to make single measurements of multiple signal levels (voltage, pressure, temperature, etc.) to decide if the system can be started or not in order to protect the people and equipment.

Independent modes

- **Single-channel continuous conversion mode**
- The single-channel continuous conversion mode converts a single channel continuously and indefinitely in regular channel conversion.
- The continuous mode feature allows the ADC to work in the background. The ADC converts the channels continuously without any intervention from the CPU. Additionally, the DMA can be used in circular mode, thus reducing the CPU load.

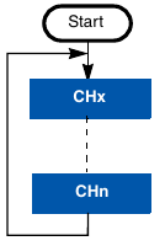


Independent modes

- Example of applications
- This ADC mode can be implemented to monitor a battery voltage, the measurement and regulation of an oven temperature, etc.
- In the case of the oven temperature regulation, the temperature is read and compared to the temperature set by the user. When the oven temperature reaches the desired temperature, the heating resistor is powered off.

Independent modes

- **Multichannel (scan) continuos conversion mode**
- The multichannel, or scan, continuous mode can be used to convert some channels successively with the ADC in independent mode. With the sequencer, you can configure any sequence of up to 16 channels successively with different sampling times and different orders. This mode is similar to the multichannel single conversion mode except that it does not stop converting after the last channel of the sequence but it restarts the conversion sequence from the first channel and continues indefinitely.

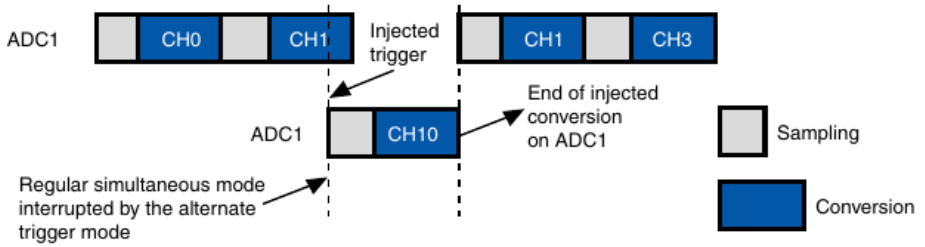


Independent modes

- Example of application
- This mode can be used to monitor multiple voltages and temperatures in a multiple battery charger. The voltage and temperature of each battery are read during the charging process. When the voltage or the temperature reaches the maximum level, the corresponding battery should be disconnected from the charger.

Independent modes

- **Injected conversion mode**
- This mode is intended for use when conversion is triggered by an external event or by software.
- The injected group has priority over the regular channel group. It interrupts the conversion of the current channel in the regular channel group.



Independent modes

- Example of application
- This mode can be used to synchronize the conversion of channels to an event. It is interesting in motor control applications where transistor switching generates noise that impacts ADC measurements and results in wrong conversions. Using a timer, the injected conversion mode can thus be implemented to delay the ADC measurements to after the transistor switching.

Programming ADC of STM32

Single channel, single conversion: PA1 pin as analog

Step-1. Enable clock of GPIOA port

RCC->AHB1ENR |= 0x1;

6.3.10 RCC AHB1 peripheral clock register (RCC_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reser-ved	OTGHS ULPIEN	OTGHSEN	ETHMACPTPEN	ETHMACRXEN	ETHMACTXEN	ETHMACEN	Res.	**DMA2DEN**	DMA2EN	DMA1EN	CCMDATARAMEN	Res.	BKPSRAMEN	Reserved	
	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Res.	GPIOKEN	GPIOJEN	GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Programming ADC of STM32

Step-2. Select analog mode for PA1 pin

GPIOA->MODER |=0xC;

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Programming ADC of STM32

Step-3. Enable clock access to ADC1

RCC->APB2ENR |= 0x100;

6.3.14 RCC APB2 peripheral clock enable register (RCC_APB2ENR)

Address offset: 0x44

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved					LTDCEN	Reserved			SAI1EN	SPI6EN	SPI5EN	Res.	TIM11EN	TIM10EN	TIM9EN
					rw				rw	rw	rw		rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reser-ved	SYSCFG EN	SPI4EN	SPI1EN	SDIOEN	ADC3EN	ADC2EN	ADC1EN	Reserved		USART6EN	USART1EN	Reserved		TIM8EN	TIM1EN
	rw	rw	rw	rw	rw	rw	rw			rw	rw			rw	rw

Programming ADC of STM32

Step-4. Software trigger or disable ADC

ADC1->CR2 = 0;

13.13.3 ADC control register 2 (ADC_CR2)

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved	SWSTART	EXTEN		EXTSEL[3:0]				reserved	JSWSTART	JEXTEN		JEXTSEL[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved				ALIGN	EOCS	DDS	DMA	Reserved						CONT	ADON
				rw	rw	rw	rw							rw	rw

Programming ADC of STM32

Step-5. Conversion sequence starts at Ch1 (PA1)

ADC1->SQR3 = 1;

13.13.11 ADC regular sequence register 3 (ADC_SQR3)

Address offset: 0x34

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		SQ6[4:0]					SQ5[4:0]					SQ4[4:1]			
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ4_0	SQ3[4:0]						SQ2[4:0]					SQ1[4:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Programming ADC of STM32

Step-6. Conversion sequence length 1

ADC1->SQR1 = 0;

13.13.9 ADC regular sequence register 1 (ADC_SQR1)

Address offset: 0x2C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								L[3:0]				SQ16[4:1]			
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ16_0	SQ15[4:0]					SQ14[4:0]					SQ13[4:0]				
rw	rw	rw	rw	rw	rw	rw	rw				rw	rw	rw	rw	rw

Programming ADC of STM32

Step-7. Enable ADC1

ADC1->CR2 |= 1;

13.13.3 ADC control register 2 (ADC_CR2)

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved	SWSTART	EXTEN		EXTSEL[3:0]				reserved	JSWSTART	JEXTEN		JEXTSEL[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved				ALIGN	EOCS	DDS	DMA	Reserved						CONT	ADON
				rw	rw	rw	rw							rw	rw

Programming ADC of STM32

Step-8. Start a conversion

ADC1->CR2 |= 0x40000000;

13.13.3 ADC control register 2 (ADC_CR2)

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved	SWST ART	EXTEN		EXTSEL[3:0]				reserved	JSWST ART	JEXTEN		JEXTSEL[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved				ALIGN	EOCS	DDS	DMA	Reserved						CONT	ADON
				rw	rw	rw	rw							rw	rw

Programming ADC of STM32

Step-9. Wait for conversion complete

while(!(ADC1->SR & 2));

13.13.1 ADC status register (ADC_SR)

Address offset: 0x00

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										OVR	STRT	JSTRT	JEOC	EOC	AWD
										rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

Programming ADC of STM32

Step-10. Read conversion result

analogValue = ADC1->DR;

13.13.14 ADC regular data register (ADC_DR)

Address offset: 0x4C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

Thank you

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE

BITS Pilani
Pilani Campus

ESZG512/MELZG526/SEZG516
Embedded System Design
Contact Session 10

Contents

Introduction to LPC 23xx: Timer

- **Features**
- Four Timer/Counters
- 4 channels/Timer
- A minimum of 2 capture i/ps & 2 Match o/ps are pinned out for all 4 timers
- Choice of several pins for each Timer
- 32 bit Timer/Counter with programmable 32 bit Prescaler
- Counter / Timer operation

Source: LPC23XX User manual

Introduction to LPC 23xx: Timer

- **TIMER/COUNTER 0-3 registers**
- **IR** Interrupt Register: The IR can be written to clear interrupts. The IR can be read to identify which of eight possible interrupt sources are pending
- **TCR** Timer Control Register: The TCR is used to control the Timer Counter functions. The Timer Counter can be disabled or reset through the TCR
- **TC** Timer Counter: The 32 bit TC is incremented every PR+1 cycles of PCLK. The TC is controlled through the TCR
- **PR** Prescale Register: The Prescale Counter is equal to this value, the next clock increments the TC and clears the PC

Source: LPC23XX User manual

Introduction to LPC 23xx: Timer

- **PC** Prescale Counter: The 32 bit PC is a counter which is incremented to the value stored in PR. When the value in PR is reached, the TC is incremented and the PC is cleared. The PC is observable and controllable through the bus interface
- **MCR** Match Control Register: The MCR is used to control if an interrupt is generated and if the TC is reset when a Match occurs
- **MR0/1/2/3** Match Register 0/1/2/3: MR0/1/2/3 can be enabled through the MCR to reset the TC, stop both the TC and PC, and/or generate an interrupt every time MR0/1/2/3 matches the TC

Source: LPC23XX User manual

Introduction to LPC 23xx: Timer

- **CCR** Capture Control Register: The CCR controls which edges of the capture inputs are used to load the Capture Registers and whether or not an interrupt is generated when a capture takes place.
- **CR0** Capture Register 0: CR0 is loaded with the value of TC when there is an event on the CAPn.0 (CAP0.0, CAP1.0, CAP2.0, or CAP3.0) input
- **CR1** Capture Register 1: CR1 is loaded with the value of TC when there is an event on the CAPn.0 (CAP1.0, CAP1.1, CAP2.1, or CAP3.1) input

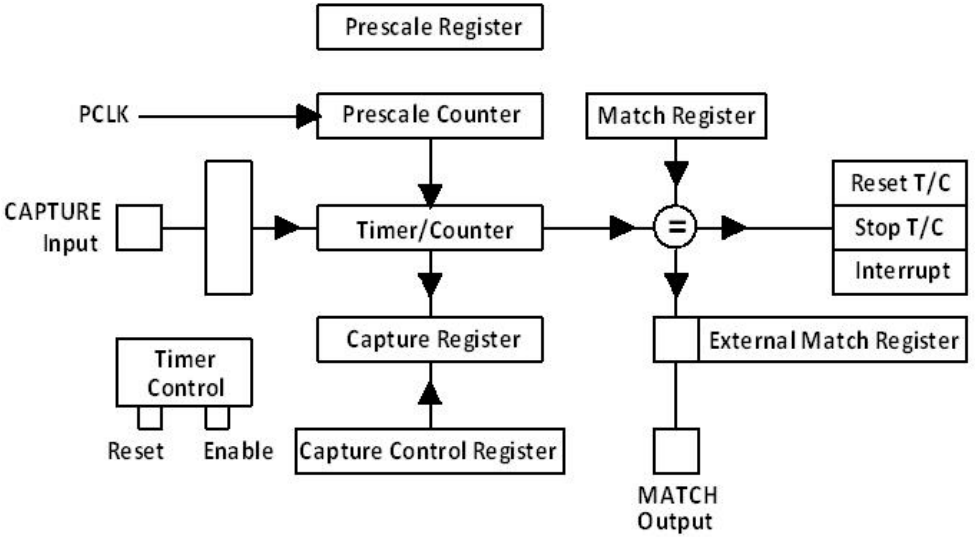
Source: LPC23XX User manual

Introduction to LPC 23xx: Timer

- **EMR** External Match Register: The EMR controls the external match pins MATn.0-3 (n corresponds to timers 0 to 3)
- **CTCR** Count Control Register: The CTCR selects between Timer and Counter mode, and in Counter mode selects the signal and edge(s) for counting

Source: LPC23XX User manual

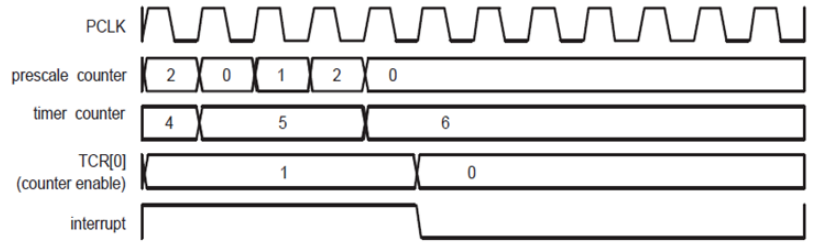
Introduction to LPC 23xx: Timer



Functional Block Diagram

Introduction to LPC 23xx: Timer

- Figure shows a timer configured to stop and generate an interrupt on match.
- The prescaler is again set to 2 and the match register set to 6.
- In the next clock after the timer reaches the match value, the timer enable bit in TCR is cleared, and the interrupt indicating that a match occurred is generated.



A Timer Cycle in Which PR=2, MRx=6, and Both Interrupt and Stop on Match are Enabled

Source: LPC23XX User manual

Programming of Timer

- CALCULATION OF TIME DELAY:
- Peripheral Frequency = CPU Clock / APB divider
- Since we are using Default 1/4 divider here our peripheral frequency will be
- Peripheral Frequency = 48 MHz / 4 = 12 MHz
- Then we have to calculate T0MR0 (count) value to generate required delay by matching
- Count = (Peripheral Frequency / Desired Frequency) – 1
  = (12,000,000 / 10) – 1
  = (12,000,000 – 1)
  = 11,99,999
- Now count 1199999 gives the T0MR0 value to generate 100ms delay. So to make it as 1 sec we have to multiply it with 10. And that we can use it in the Prescale register T0PR, thus we can generate a delay of 1 sec using this.

Programming of Timer

STEPS TO PROGRAM TIMERS:

- Reset timer0 initially to deactivate counting.
- Load calculated values in the Prescaler register T0PR and Match register T0MR0.
- Initialize T0PC and T0TC registers.
- Select operations using match registers when match is encountered.
- Start the Timer by enabling it through T0TCR register.
- Wait till the interrupt and then clear the flag by writing T0IR register.

Programming of Timer

```
#include<lpc23xx.h>
#define LED_D0 1<<15
#define LED_D1 1<<16
#define LED_D2 1<<17
#define LED_D3 1<<18
#define LED_D4 1<<19
#define LED_D5 1<<20
#define LED_D6 1<<21
#define LED_D7 1<<22
void delay(void);
int main (void)
{
        IODIR0 = LED_D0|LED_D1|LED_D2|LED_D3|LED_D4|LED_D5|LED_D6|LED_D7;  /* Define Output pins */

        while (1)       /* Loop forever */
        {
        /* bit pattern for LEDs connected at GPIO Port 0*/
        IOSET0=LED_D0|LED_D1|LED_D2|LED_D3|LED_D4|LED_D5|LED_D6|LED_D7; /* turn LEDs ON */

        delay();                        /* Delay */
        IOCLR0=LED_D0|LED_D1|LED_D2|LED_D3|LED_D4|LED_D5|LED_D6|LED_D7;    /* turn LEDs OFF */
        delay();                        /* Delay */
        }
}
```


Programming of Timer

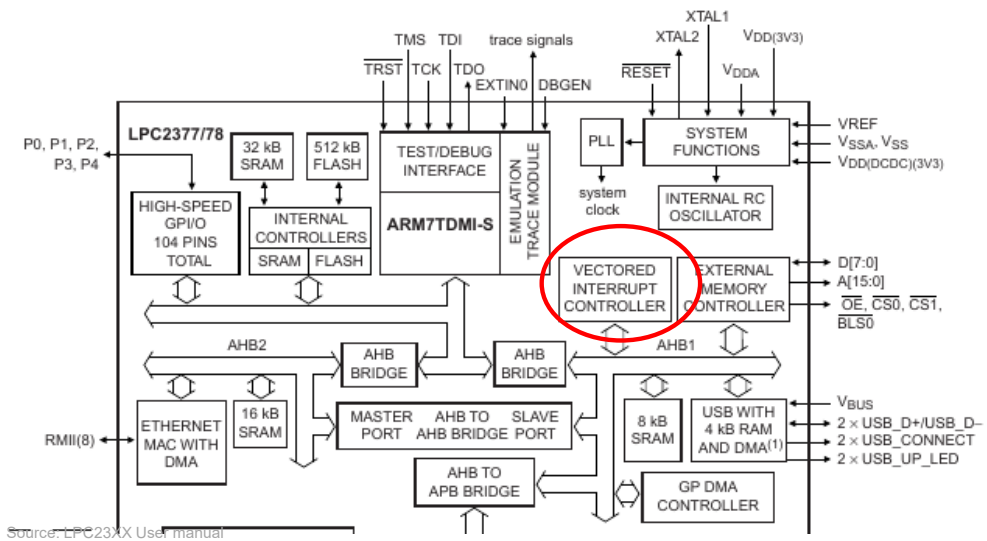
```
void delay(void)
{
  T0TCR=(1<<1);             //Reset Timer0
  T0MR0=1199999;            //Loading match register value
  T0PR=9;                   //Loading Prescalar register value
  T0PC=T0TC=0;
  T0MCR=(1<<0)|(1<<2); //Generates interrupt and stop on match
  T0TCR=(1<<0);             //Starting Timer
  while(!(T0IR&(1<<0)));  //Waiting for interrupt
  T0IR=(1<<0);          //Clearing interrupt: Writing a logic one to the
                        //corresponding IR bit will reset the interrupt.
}
```


Interrupts

- Interrupt : “An interrupt is a signal sent to the CPU which indicates that a system event has a occurred which needs immediate attention”.
- Interrupt ReQuest (IRQ) can be thought of as a special request to the CPU to execute a function(small piece of code) when an interrupt occurs.
- ISR : This function or ‘small piece of code’ is technically called an ‘Interrupt Service Routine‘ or ‘ISR’.
- So when an IRQ arrives to the CPU, it stops executing the current code and start executing the ISR. After the ISR execution has finished the CPU gets back to where it had stopped.

Interrupts

Interrupts are handled by Vectored Interrupt Controller(VIC)



Source: LPC23XX User manual

Interrupts

LPC23XX Vectored Interrupt Controller (VIC):

Features

- ARM PrimeCell Vectored Interrupt Controller
- Mapped to AHB address space for fast access
- Supports 32 vectored IRQ interrupts
- 16 programmable interrupt priority levels
- Fixed hardware priority within each programmable priority level
- Hardware priority level masking
- Any input can be assigned as an FIQ interrupt
- Software interrupt generation

Source: LPC23XX User manual

Interrupts

Interrupt structure of LPC23xx

- The VIC is a component from the ARM prime cell.
- VIC module is a highly optimized interrupt controller.
- The VIC is used to handle all the on-chip interrupt sources from peripherals.
- Each of the on-chip interrupt sources is connected to the VIC on a fixed channel: your application software can connect each of these channels to the CPU interrupt lines (FIQ, IRQ) in one of three ways.

Interrupts

- The VIC allows each interrupt to be handled as an FIQ interrupt, a vectored IRQ interrupt
- FIQ is the fastest followed by vectored IRQ with non-vectored IRQ being the slowest.

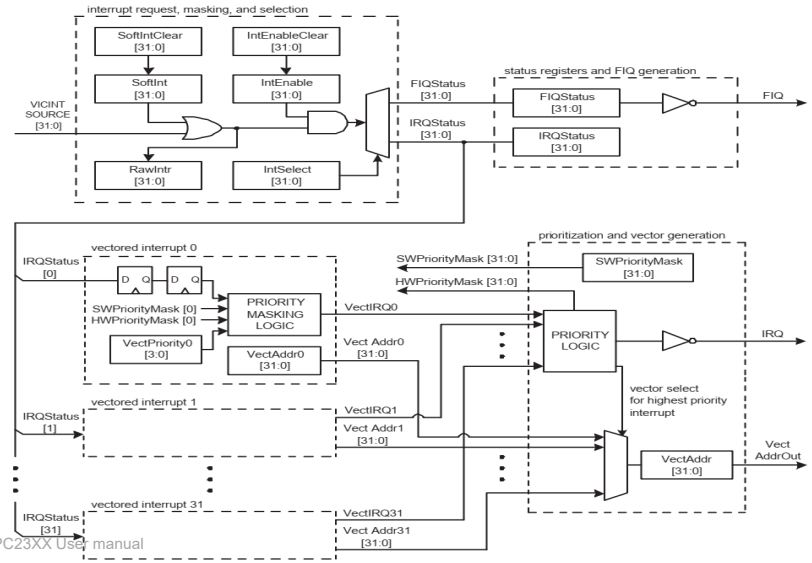
Interrupts

- The ARM processor core has two interrupt inputs called Interrupt Request (IRQ) and Fast Interrupt reQuest (FIQ).
- VIC takes 32 interrupt request inputs and programmably assigns them as FIQ or vectored IRQ types.
- Fast Interrupt reQuest (FIQ) requests have the highest priority.
- If more than one request is assigned to FIQ, the VIC ORs the requests to produce the FIQ signal to the ARM processor.
- Vectored IRQ's, which include all interrupt requests that are not classified as FIQs, have a programmable interrupt priority.
- The VIC ORs the requests from all of the vectored IRQs to produce the IRQ signal to the ARM processor.

Source: LPC23XX User manual

Interrupts

Block diagram of the Vectored Interrupt Controller



Source: LPC23XX User manual

Interrupts

VIC register map

- **VICIRQStatus** - IRQ Status Register: This register reads out the state of those interrupt requests that are enabled and classified as IRQ.
- **VICFIQStatus** - FIQ Status Requests: This register reads out the state of those interrupt requests that are enabled and classified as FIQ.
- **VICRawIntr** - Raw Interrupt Status Register: This register reads out the state of the 32 interrupt requests / software interrupts, regardless of enabling or classification.
- **VICIntSelect** - Interrupt Select Register: This register classifies each of the 32 interrupt requests as contributing to FIQ or IRQ.

Source: LPC23XX User manual

Interrupts

- **VICIntEnable** - Interrupt Enable Register: This register controls which of the 32 interrupt requests and software interrupts are enabled to contribute to FIQ or IRQ.
- **VICIntEnClr** - Interrupt Enable Clear Register: This register allows software to clear one or more bits in the Interrupt Enable register
- **VICSoftInt** - Software Interrupt Register: The contents of this register are ORed with the 32 interrupt requests from various peripheral functions.
- **VICSoftIntClear** - Software Interrupt Clear Register: This register allows software to clear one or more bits in the Software Interrupt register

Source: LPC23XX User manual

Interrupts

- **VICProtection** - Protection enable register: This register allows limiting access to the VIC registers by software running in privileged mode.
- **VICSWPriorityMask** - Software Priority Mask Register: Allows masking individual interrupt priority levels in any combination.
- **VICVectAddr0-31** - Vector address 0-31 register: Vector Address Registers 0-31 hold the addresses of the Interrupt Service routines (ISRs) for the 32 vectored IRQ slots.
- **VICVectPriority0-31** - Vector priority 0-31 register: Vector Priority Registers 0-31. Each of these registers designates the priority of the corresponding vectored IRQ slot
- **VICAddress** - Vector address register: When an IRQ interrupt occurs, the Vector Address Register holds the address of the currently active interrupt

Source: LPC23XX User manual

Interrupts

FIQ Interrupt

- Any interrupt source may be assigned as the FIQ interrupt.
- The VIC Interrupt Select Register has a unique bit for each interrupt.
- Setting this bit connects the selected channel to the FIQ interrupt.
- In an ideal system we would only have one FIQ interrupt.
- However setting multiple bits in the Interrupt Select Register will enable multiple FIQ interrupt sources.
- On entry the interrupt source can be determined by examining the VIC FIQ Status Register and the appropriate code executed.
- Several FIQ sources slows entry into the ISR code.

Interrupts

- Once you have selected an FIQ source the interrupt can be enabled in the VIC Interrupt Enable Register.
- Configuring the VIC, the peripheral generating the interrupt must be configured and its own interrupt registers enabled.
- Once an FIQ interrupt is generated, the processor will change to FIQ Mode and vector to FIQ vector.
- Place a jump to your ISR routine at this location in order to serve the interrupt.

Interrupts

Leaving an FIQ Interrupt

- Before you exit the ISR code you must make sure that any interrupt status flags in the peripheral have been cleared.
- If this is not done you will get continuous interrupts until the flag is cleared.
- To clear the flag: write a logic 1 not a logic 0.

Interrupts

Vectored IRQ

- The VIC provides a programmable hardware lookup table which delivers the address of the C function to run for a given interrupt source.
- The VIC contains 32 slots for vectored addressing.
- Each slot contains a Vector Address Register and a Vector Priority Register.
- The Vector Priority Register allows you to assign a priority to each interrupt slot.
- It supports 16 priority levels, 15 being the lowest priority and 0 the highest.
- After reset the priority of all the VIC slots is set to 15, and the individual priority can be elevated by the user.

Interrupts

- Vector Address Register must be initialised with the address of the appropriate C function to run when the interrupt associated with the slot occurs.
- So whenever an interrupt configured as a vectored interrupt is generated, the address of its ISR will be loaded into a fixed memory location called the Vector Address Register.
- The ARM7 CPU is going through its normal entry into the IRQ Mode and will jump to IRQ interrupt vector.

Interrupts

Leaving An IRQ Interrupt

- Interrupt status flags are cleared in the peripheral which generated the request.
- At the end of the interrupt you must do a dummy write to the Vector Address Register.
- This signals the end of the interrupt to the VIC, and any pending IRQ interrupt will be asserted.

Timer 0 interrupt programming

Steps:

1. Convert Timer0 interrupt to an IRQ interrupt
2. Set Vectored IRQ slot 4 to highest priority
3. Pass address of the IRQ into the VIC Slot
4. Enable Timer0 as IRQ interrupt
5. In IRQ interrupt handler:
    1. Clear interrupt flag
    2. VICAddress register must be written (with any value) at the end of an ISR

Thank you

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE

innovate achieve lead

BITS Pilani
Pilani Campus

ESZG512/MELZG526/SEZG516
Embedded System Design
Contact Session 11

Contents

innovate achieve lead

General-purpose timer

innovate achieve lead

- The timer peripheral is part of the essential set of peripherals embedded in all the STM32 microcontrollers.
- The number of timer peripherals and their respective features differ from one STM32 microcontroller family to another, but they all share some common features and operating modes.
- The general purpose timers embedded by the STM32 microcontrollers share the same backbone structure; they differ only on the level of features embedded by a given timer peripheral.
- The level of features integration for a given timer peripheral is decided based on the applications field that it targets.

Timers

The timer peripherals can be classified as:

- Advanced-configuration timers like TIM1 and TIM8 among others.
- General-purpose configuration timers like TIM2, TIM3, TIM4, and TIM5 among others
- Lite-configuration timers like TIM9, TIM10, TIM12 and TIM16 among others
- Basic-configuration timers like TIM6 and TIM7 among others.

TIM6 & TIM7 introduction

- Basic timer (TIM6 & TIM7) features include:
  - 16-bit auto-reload upcounter
  - 16-bit programmable prescaler used to divide (also “on the fly”) the counter clock frequency by any factor between 1 and 65536
  - Synchronization circuit to trigger the DAC
  - Interrupt/DMA generation on the update event: counter overflow

General-purpose timers (TIM9 to TIM14)

The features of the TIM9 to TIM14 general-purpose timers include:

• 16-bit auto-reload upcounter

• 16-bit programmable prescaler used to divide the counter clock frequency by any factor between 1 and 65536 (can be changed “on the fly”)

• Up to 2 independent channels for:

– Input capture

– Output compare

– PWM generation (edge-aligned mode)

– One-pulse mode output

• Synchronization circuit to control the timer with external signals and to interconnect several timers together

• Interrupt generation on the following events:

– Update: counter overflow, counter initialization (by software or internal trigger)

– Trigger event (counter start, stop, initialization or count by internal trigger)

– Input capture

– Output compare

General-purpose timers (TIM2 to TIM5)

General-purpose TIMx timer features include:

• 16-bit (TIM3 and TIM4) or 32-bit (TIM2 and TIM5) up, down, up/down auto-reload counter.

• 16-bit programmable prescaler used to divide (also “on the fly”) the counter clock frequency by any factor between 1 and 65536.

• Up to 4 independent channels for:

– Input capture

– Output compare

– PWM generation (Edge- and Center-aligned modes)

– One-pulse mode output

• Synchronization circuit to control the timer with external signals and to interconnect several timers.

• Interrupt/DMA generation on the following events:

– Update: counter overflow/underflow, counter initialization (by software or internal/external trigger)

– Trigger event (counter start, stop, initialization or count by internal/external trigger)

– Input capture

– Output compare

• Supports incremental (quadrature) encoder and hall-sensor circuitry for positioning purposes

• Trigger input for external clock or cycle-by-cycle current management

Advanced-control timers (TIM1 & TIM8)

- The advanced-control timers (TIM1&TIM8) consist of a 16-bit auto-reload counter driven by a programmable prescaler.
- It may be used for a variety of purposes, including measuring the pulse lengths of input signals (input capture) or generating output waveforms (output compare, PWM, complementary PWM with dead-time insertion)
- TIM1&TIM8 timer features include:

• 16-bit up, down, up/down auto-reload counter.

• 16-bit programmable prescaler allowing dividing (also "on the fly") the counter clock frequency either by any factor between 1 and 65536.

• Up to 4 independent channels for:

– Input Capture

– Output Compare

– PWM generation (Edge and Center-aligned Mode)

– One-pulse mode output

Advanced-control timers (TIM1 & TIM8)

- Complementary outputs with programmable dead-time
- Synchronization circuit to control the timer with external signals and to interconnect several timers together.
- Repetition counter to update the timer registers only after a given number of cycles of the counter.
- Break input to put the timer's output signals in reset state or in a known state.
- Interrupt/DMA generation on the following events:

– Update: counter overflow/underflow, counter initialization (by software or internal/external trigger)

– Trigger event (counter start, stop, initialization or count by internal/external trigger)

– Input capture

– Output compare

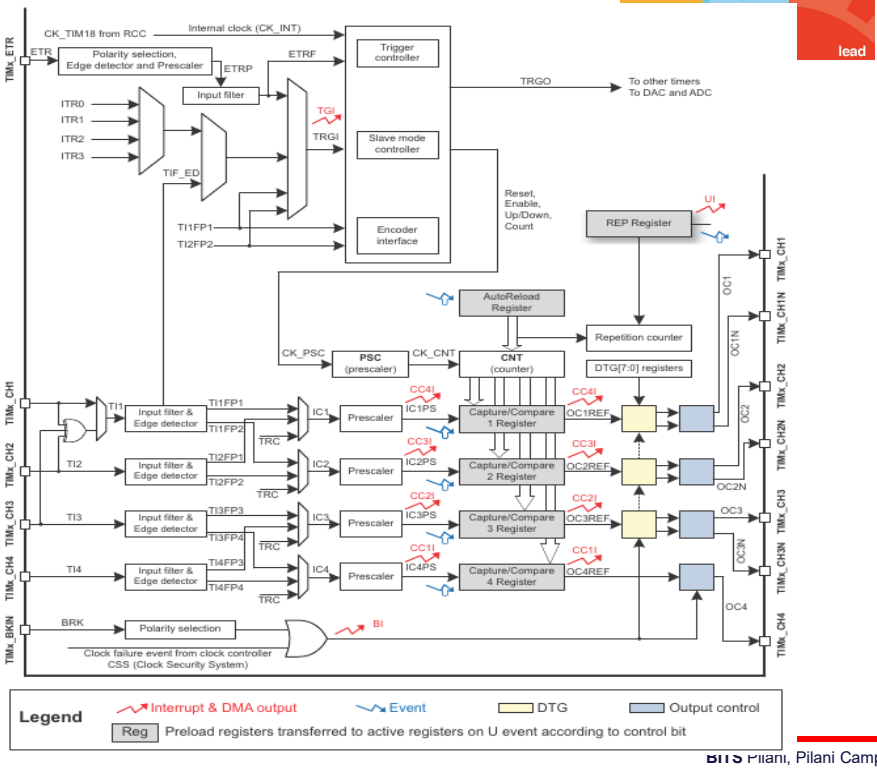
– Break input

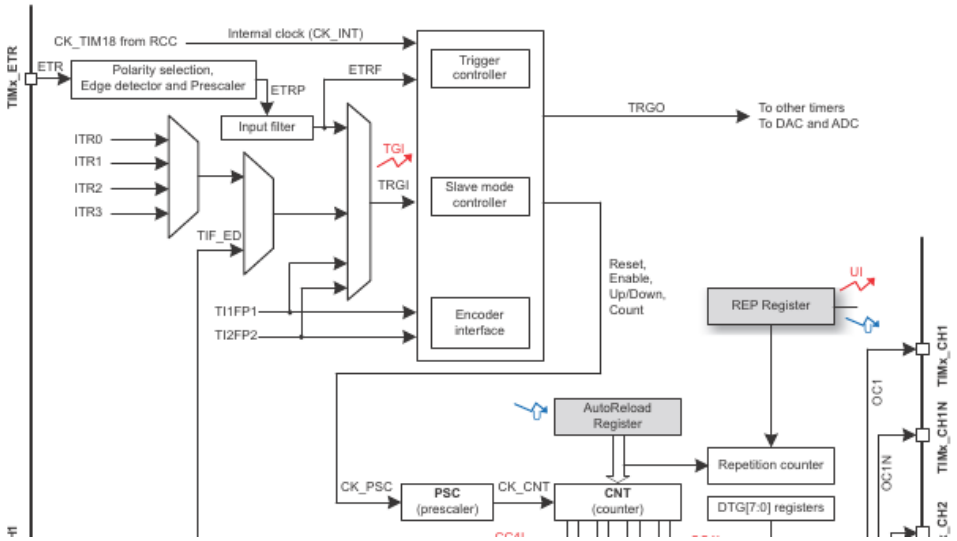
- Supports incremental (quadrature) encoder and hall-sensor circuitry for positioning purposes
- Trigger input for external clock or cycle-by-cycle current management

General-purpose timer

- All the STM32 general-purpose timer peripherals share the same backbone structure.
- Figure shows the block diagram for the TIM1 timer peripheral.
- The STM32 timer peripheral is made by the assembly of four units:

1. The master/slave controller unit
2. The time-base unit
3. The timer channels unit
4. The Break feature unit.





The master/slave controller unit:

The master/slave controller unit:

- The master/slave unit provides the time-base unit with the counting clock signal (for example the CK_PSC signal), as well as the counting direction control signal.
- This unit mainly provides the control signals for the time-base unit.
- The master/slave controller decides the right counting configuration for the time-base unit based on the timer master/slave configuration; it also decides the actual counting status.

The master/slave controller unit:

- The master/slave controller unit handles the inter-timers synchronization.
- This unit can be configured to output a synchronization signal (TRGO signal) next to a certain timer internal event.
- It can be configured as well to control the time-base counter in function of external events (like internal events of other timers or external signals).
- It is possible to configure one slave timer to increment its counter based on a master-timer events such as the timer update event.
- Not all the STM32 timer peripherals feature the same master/slave controller capabilities.

The time-base unit:

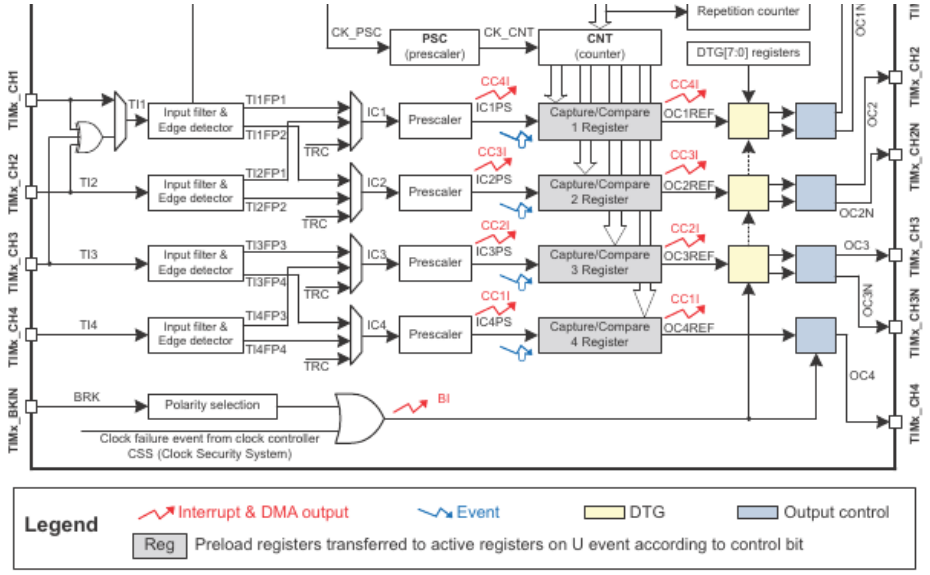
The time-base unit:

- The time-base unit is made by the timer counter in addition to a prescaler stage and a repetition counter.
- The clock signal fed into the time-base unit passes first through a prescaling stage before reaching the time-base counter.
- Depending on the content of the TIMx_PSC timer prescaler register, the counting signal frequency may be scaled down before reaching the counter stage.
- The output signal of the prescaling stage is the clock counting signal for the timer counter stage.

The time-base unit:

The timer counter is controlled by two timer registers:

- The TIMx_CNT timer register is used to read and write the content of the timer counter.
- The TIMx_ARR timer register contains the reload value of the timer counter.



The timer-channels unit:

The timer-channels unit:

- The timer channels are the working elements of the timer; they are the means by which a timer peripheral interacts with its external environment.
- In general, the timer channels are mapped to the STM32 microcontroller pins with few exceptions such as the timer channel 5 and 6 for the TIM1 timer peripheral on the STM32F30x microcontrollers family.
- A timer channel mapped to an STM32 microcontroller pin can be used either as an input or as an output.

The Break feature unit:

The Break feature unit:

- The Break feature unit is embedded only by timer peripherals that feature complementary outputs. In other words, only timer peripherals that have at least one channel with two complimentary outputs embed the Break feature.
- The Break feature acts on the output stage of timer channels configured in output mode. As soon as an active edge is detected on the break input, the outputs of timer channels configured in output mode are either turned off or forced to a predefined safe state.
- The Break feature is typically used for implementing safe shutdown functionality in electrical power inverters next to anomalies.

Example.1.

Toggle the LED connected at PD12 at rate of 1Hz using Timer-2 as general purpose timer.

Example.1.

General purpose timer programming steps:

1. Enable GPIOD clock *(RCC->AHB1ENR)*
2. Set PD12 pin as output mode *(GPIOD->MODER)*
3. Configure timer2 for 1Hz clock generation:
    1. Enable Timer2 clock *(RCC->APB1ENR)*
    2. Set PSC value as 1600-1 *(TIM2->PSC)*
    3. Set ARR value as 10000-1 *(TIM2->ARR)*
    4. Clear counter *(TIM2->CNT)*
    5. Enable Timer2 *(TIM2->CR1)*
4. In while loop condition check for UIF flag ==1 *(TIM2->SR)*
5. When UIF == 1, loop breaks and after that clear UIF *(TIM2->SR)*
6. Toggle PD12 *(GPIOD->ODR)*


```
#include "stm32f4xx.h"              // Device header
#define PIN12                       (1U<<12)
#define LED                 PIN12
#define SR_UIF              (1U<<0)

int main(void)
{

    RCC->AHB1ENR |= (1<<3); //Enable GPIOD clock
    GPIOD->MODER &=0x0C000000; //Clear pin mode
    GPIOD->MODER |=0x01000000; //PD.12 set pin to output mode

    // configure TIM2 to wrap around at 1 Hz

    RCC->APB1ENR |= (1U<<0);                                /* enable TIM2 clock */
    TIM2->PSC = 1600 - 1;           /* divided by 1600 i.e. 16000000/1600=10000*/
    TIM2->ARR = 10000 - 1;          /* divided by 10000 i.e. 10000/10000=1 */
    TIM2->CNT = 0;                                                  /* clear counter */
    TIM2->CR1 = (1U<<0);                                                    /* enable TIM2 */

    while (1)
    {
        while(!(TIM2->SR & SR_UIF))     // Wait for UIF
        {
        }
        TIM2->SR &= ~SR_UIF;   // Clear UIF
        GPIOD->ODR ^= LED;
    }
}
```


RCC->AHB1ENR |= (1<<3); //Enable GPIOD clock

6.3.10 RCC AHB1 peripheral clock register (RCC_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reser-ved	OTGHS ULPIEN	OTGHSEN	ETHMACPTPEN	ETHMACRXEN	ETHMACTXEN	ETHMACEN	Res.	**DMA2DEN**	DMA2EN	DMA1EN	CCMDATARAMEN	Res.	BKPSRAMEN	Reserved	
	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Res.	GPIOKEN	GPIOJEN	GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

```
GPIOD->MODER &=0x0C000000;          //Clear pin mode
GPIOD->MODER |=0x01000000;          //PD.12 set pin to output mode
```

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode


```
// configure TIM2 to wrap around at 1 Hz
RCC->APB1ENR |= (1U<<0);            /* enable TIM2 clock */
```

6.3.13 RCC APB1 peripheral clock enable register (RCC_APB1ENR)

Address offset: 0x40

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
UART8 EN	UART7 EN	DAC EN	PWR EN	Reser-ved	CAN2 EN	CAN1 EN	Reser-ved	I2C3 EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART 3 EN	USART 2 EN	Reser-ved
rw	rw	rw	rw		rw	rw		rw	rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	Reserved		WWDG EN	Reserved		TIM14 EN	TIM13 EN	TIM12 EN	TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw			rw			rw	rw	rw	rw	rw	rw	rw	rw	rw


```
TIM2->PSC = 1600 - 1;       /* divided by 1600 i.e. 16000000/1600=10000*/
```

18.4.11 TIMx prescaler (TIMx_PSC)

Address offset: 0x28

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **PSC[15:0]**: Prescaler value
The counter clock frequency CK_CNT is equal to f_{CK_PSC} / (PSC[15:0] + 1).
PSC contains the value to be loaded in the active prescaler register at each update event (including when the counter is cleared through UG bit of TIMx_EGR register or through trigger controller when configured in "reset mode").


```
TIM2->ARR = 10000 - 1;      /* divided by 10000 i.e. 10000/10000=1 */
```

18.4.12 TIMx auto-reload register (TIMx_ARR)

Address offset: 0x2C

Reset value: 0xFFFF FFFF

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ARR[31:16] (depending on timers)															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **ARR[15:0]**: Auto-reload value
ARR is the value to be loaded in the actual auto-reload register.
Refer to the *Section 18.3.1: Time-base unit* for more details about ARR update and behavior.
The counter is blocked while the auto-reload value is null.

TIM2->CNT = 0; /* clear counter */

18.4.10 TIMx counter (TIMx_CNT)

Address offset: 0x24

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNT[31:16] (depending on timers)															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **CNT[15:0]**: Counter value

TIM2->CR1 = (1U<<0); /* enable TIM2 */

18.4.1 TIMx control register 1 (TIMx_CR1)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 0 **CEN**: Counter enable

0: Counter disabled

1: Counter enabled

Note: External clock, gated mode and encoder mode can work only if the CEN bit has been previously set by software. However trigger mode can set the CEN bit automatically by hardware.

CEN is cleared automatically in one-pulse mode, when an update event occurs.


```
while (1)
{
    while(!(TIM2->SR & SR_UIF))     // Wait for UIF
    {
    }
```

18.4.5 TIMx status register (TIMx_SR)

Address offset: 0x10

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

Bit 0 **UIF**: Update interrupt flag

″ This bit is set by hardware on an update event. It is cleared by software.
0: No update occurred.
1: Update interrupt pending. This bit is set by hardware when the registers are updated:

″ At overflow or underflow (for TIM2 to TIM5) and if UDIS=0 in the TIMx_CR1 register.

″ When CNT is reinitialized by software using the UG bit in TIMx_EGR register, if URS=0 and UDIS=0 in the TIMx_CR1 register.

When CNT is reinitialized by a trigger event (refer to the synchro control register description), if URS=0 and UDIS=0 in the TIMx_CR1 register.

TIM2->SR &= ~SR_UIF; // Clear UIF

18.4.5 TIMx status register (TIMx_SR)

Address offset: 0x10

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

Bit 0 **UIF**: Update interrupt flag

″ This bit is set by hardware on an update event. It is cleared by software.
0: No update occurred.
1: Update interrupt pending. This bit is set by hardware when the registers are updated:

″ At overflow or underflow (for TIM2 to TIM5) and if UDIS=0 in the TIMx_CR1 register.

″ When CNT is reinitialized by software using the UG bit in TIMx_EGR register, if URS=0 and UDIS=0 in the TIMx_CR1 register.

When CNT is reinitialized by a trigger event (refer to the synchro control register description), if URS=0 and UDIS=0 in the TIMx_CR1 register.

```
GPIOD->ODR ^= LED;
    }
}
```

8.4.6 GPIO port output data register (GPIOx_ODR) (x = A..I/J/K)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy**: Port output data (y = 0..15)

These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (x = A..I/J/K).

Example.2.

Toggle the TIM2 Channel-1 at the rate of 1Hz using Timer-2 compare method

Example.2.

From Alternate function mapping, select PA5 as TIM2_CH1

Port	AF0	AF1	AF2	AF3	AF4	AF5	AF6
	SYS	TIM1/2	TIM3/4/5	TIM8/9/10 /11	I2C1/2/3	SPI1/SPI2/ I2S2/I2S2e xt	SPI3/I2Sext /I2S3
PA0	-	TIM2_CH1_ ETR	TIM 5_CH1	TIM8_ETR	-	-	-
PA1	-	TIM2_CH2	TIM5_CH2	-	-	-	-
PA2	-	TIM2_CH3	TIM5_CH3	TIM9_CH1	-	-	-
PA3	-	TIM2_CH4	TIM5_CH4	TIM9_CH2	-	-	-
PA4	-	-	-	-	-	SPI1_NSS	SPI3_NSS I2S3_WS
PA5	-	TIM2_CH1_ ETR	-	TIM8_CH1N	-	SPI1_SCK	-
PA6	-	TIM1_BKIN	TIM3_CH1	TIM8_BKIN	-	SPI1_MISO	-

Example.2.

Timer2 output compare programming

1. Enable GPIOA clock *(RCC->AHB1ENR)*
2. Set PA5 pin mode to alternate function *(GPIOA->MODER)*
3. Select PA5 alternate function type to TIM2_CH1 (AF01) *(GPIOA->AFR[0])*
4. Configure timer2 for 1Hz clock generation
    1. Enable Timer2 clock *(RCC->APB1ENR)*
    2. Set PSC value as 1600-1 *(TIM2->PSC)*
    3. Set ARR value as 10000-1 *(TIM2->ARR)*
    4. Set Timer2 as output compare toggle mode *(TIM2->CCMR1)*
    5. Enable timer2 channel1 in compare mode *(TIM2->CCER)*
    6. Clear counter *(TIM2->CNT)*
    7. Enable Timer2 *(TIM2->CR1)*

```
#include "stm32f4xx.h"                  // Device header
#define OC_TOGGLE ((1U<<4)|(1U<<5))
#define CCER_CC1E (1U<<0)

int main(void)
{
    RCC->AHB1ENR |= (1U<<0);                                    //Enable GPIOA clock
    //PA5 mode to alternate function
    GPIOA->MODER &=~(1U<<10);
    GPIOA->MODER |=(1U<<11);
  //PA5 alternate function type to TIM2_CH1 (AF01)
    GPIOA->AFR[0] |= 0x00100000;

    // configure TIM2 at 1 Hz

    RCC->APB1ENR |= (1U<<0);                                    /* enable TIM2 clock */
    TIM2->PSC = 1600 - 1;               /* divided by 1600 i.e. 16000000/1600=10000*/
    TIM2->ARR = 10000 - 1;              /* divided by 10000 i.e. 10000/10000=1 */
    TIM2->CCMR1 = OC_TOGGLE;  //Set output compare toggle mode
    TIM2->CCER |= CCER_CC1E;  // Enable tim2 ch1 in compare mode
    TIM2->CNT = 0;                                                          /* clear counter */
    TIM2->CR1 = (1U<<0);                                                        /* enable TIM2 */

    while (1)
    {

    }
}
```


RCC->AHB1ENR |= (1U<<0); //Enable GPIOA clock

6.3.10 RCC AHB1 peripheral clock register (RCC_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reser-ved	OTGHS ULPIEN	OTGHSEN	ETHMACPTPEN	ETHMACRXEN	ETHMACTXEN	ETHMACEN	Res.	**DMA2DEN**	DMA2EN	DMA1EN	CCMDATARAMEN	Res.	BKPSRAMEN	Reserved	
	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Res.	GPIOKEN	GPIOJEN	GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

//PA5 mode to alternate function

GPIOA->MODER &=~(1U<<10);

GPIOA->MODER |=(1U<<11);

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

//PA5 alternate function type to TIM2_CH1 (AF01)

GPIOA->AFR[0] |= 0x00100000;

8.4.9 GPIO alternate function low register (GPIOx_AFRL) (x = A..I/J/K)

Address offset: 0x20

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFRL7[3:0]				AFRL6[3:0]				AFRL5[3:0]				AFRL4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFRL3[3:0]				AFRL2[3:0]				AFRL1[3:0]				AFRL0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **AFRLy:** Alternate function selection for port x bit y (y = 0..7)
These bits are written by software to configure alternate function I/Os

AFRLy selection:

0000: AF0	1000: AF8
0001: AF1	1001: AF9
0010: AF2	1010: AF10
0011: AF3	1011: AF11
0100: AF4	1100: AF12
0101: AF5	1101: AF13
0110: AF6	1110: AF14
0111: AF7	1111: AF15

// configure TIM2 at 1 Hz
RCC->APB1ENR |= (1U<<0); /* enable TIM2 clock */

6.3.13 RCC APB1 peripheral clock enable register (RCC_APB1ENR)

Address offset: 0x40

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
UART8 EN	UART7 EN	DAC EN	PWR EN	Reser-ved	CAN2 EN	CAN1 EN	Reser-ved	I2C3 EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART 3 EN	USART 2 EN	Reser-ved
rw	rw	rw	rw		rw	rw		rw	rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	Reserved		WWDG EN	Reserved		TIM14 EN	TIM13 EN	TIM12 EN	TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw			rw			rw	rw	rw	rw	rw	rw	rw	rw	rw

TIM2->PSC = 1600 - 1; /* divided by 1600 i.e. 16000000/1600=10000*/

18.4.11 TIMx prescaler (TIMx_PSC)

Address offset: 0x28

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **PSC[15:0]**: Prescaler value

The counter clock frequency CK_CNT is equal to f_{CK_PSC} / (PSC[15:0] + 1).

PSC contains the value to be loaded in the active prescaler register at each update event (including when the counter is cleared through UG bit of TIMx_EGR register or through trigger controller when configured in "reset mode").

TIM2->ARR = 10000 - 1; /* divided by 10000 i.e. 10000/10000=1 */

18.4.12 TIMx auto-reload register (TIMx_ARR)

Address offset: 0x2C

Reset value: 0xFFFF FFFF

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ARR[31:16] (depending on timers)															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **ARR[15:0]**: Auto-reload value

ARR is the value to be loaded in the actual auto-reload register.

Refer to the *Section 18.3.1: Time-base unit* for more details about ARR update and behavior.

The counter is blocked while the auto-reload value is null.

(#define OC_TOGGLE ((1U<<4)|(1U<<5)))

TIM2->CCMR1 = OC_TOGGLE; //Set output compare toggle mode

18.4.7 TIMx capture/compare mode register 1 (TIMx_CCMR1)

Address offset: 0x18

Reset value: 0x0000

The channels can be used in input (capture mode) or in output (compare mode). The direction of a channel is defined by configuring the corresponding CCxS bits. All the other bits of this register have a different function in input and in output mode. For a given bit, OCxx describes its function when the channel is configured in output, ICxx describes its function when the channel is configured in input. Take care that the same bit can have a different meaning for the input stage and for the output stage.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2CE	OC2M[2:0]			OC2PE	OC2FE	CC2S[1:0]		OC1CE	OC1M[2:0]			OC1PE	OC1FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 6:4 **OC1M**: Output compare 1 mode

These bits define the behavior of the output reference signal OC1REF from which OC1 and OC1N are derived. OC1REF is active high whereas OC1 and OC1N active level depends on CC1P and CC1NP bits.
000: Frozen - The comparison between the output compare register TIMx_CCR1 and the counter TIMx_CNT has no effect on the outputs.(this mode is used to generate a timing base).
001: Set channel 1 to active level on match. OC1REF signal is forced high when the counter TIMx_CNT matches the capture/compare register 1 (TIMx_CCR1).
010: Set channel 1 to inactive level on match. OC1REF signal is forced low when the counter TIMx_CNT matches the capture/compare register 1 (TIMx_CCR1).
011: Toggle - OC1REF toggles when TIMx_CNT=TIMx_CCR1.
100: Force inactive level - OC1REF is forced low.
101: Force active level - OC1REF is forced high.
110: PWM mode 1 - In upcounting, channel 1 is active as long as TIMx_CNT<TIMx_CCR1 else inactive. In downcounting, channel 1 is inactive (OC1REF='0) as long as TIMx_CNT>TIMx_CCR1 else active (OC1REF=1).
111: PWM mode 2 - In upcounting, channel 1 is inactive as long as TIMx_CNT<TIMx_CCR1 else active. In downcounting, channel 1 is active as long as TIMx_CNT>TIMx_CCR1 else inactive.

Note: In PWM mode 1 or 2, the OCREF level changes only when the result of the comparison changes or when the output compare mode switches from "frozen" mode to "PWM" mode.

(#define CCER_CC1E (1U<<0))

TIM2->CCER |= CCER_CC1E; // Enable tim2 ch1 in compare mode

18.4.9 TIMx capture/compare enable register (TIMx_CCER)

Address offset: 0x20

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Res.	CC4P	CC4E	CC3NP	Res.	CC3P	CC3E	CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E
rw		rw	rw	rw		rw	rw	rw		rw	rw	rw		rw	rw

Bit 0 **CC1E**: *Capture/Compare 1 output enable.*

CC1 channel configured as output:
0: Off - OC1 is not active
1: On - OC1 signal is output on the corresponding output pin

CC1 channel configured as input:
This bit determines if a capture of the counter value can actually be done into the input capture/compare register 1 (TIMx_CCR1) or not.
0: Capture disabled
1: Capture enabled

TIM2->CNT = 0; /* clear counter */

18.4.10 TIMx counter (TIMx_CNT)

Address offset: 0x24

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNT[31:16] (depending on timers)															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **CNT[15:0]**: Counter value

TIM2->CR1 = (1U<<0); /* enable TIM2 */

18.4.1 TIMx control register 1 (TIMx_CR1)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 0 **CEN**: Counter enable
0: Counter disabled
1: Counter enabled

Note: External clock, gated mode and encoder mode can work only if the CEN bit has been previously set by software. However trigger mode can set the CEN bit automatically by hardware.

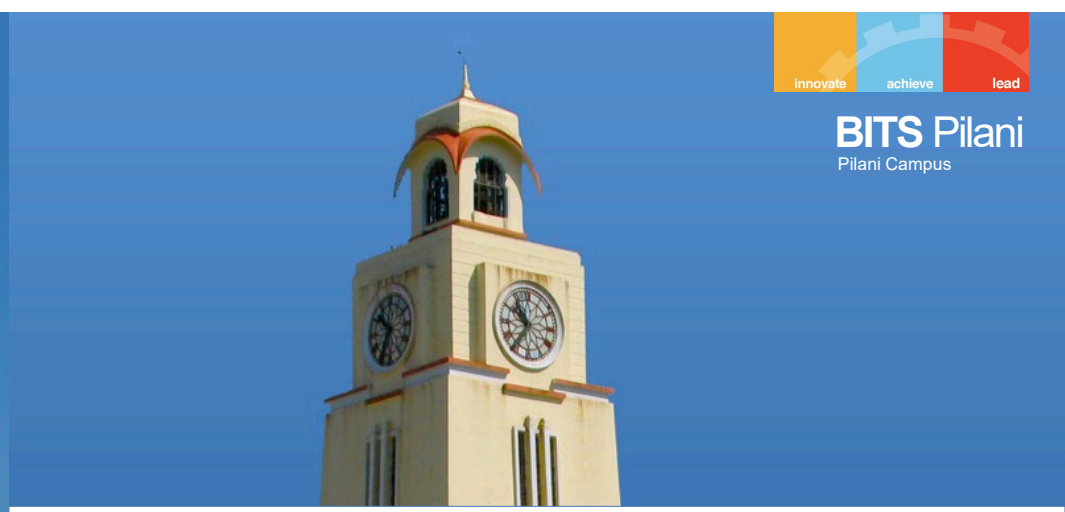
CEN is cleared automatically in one-pulse mode, when an update event occurs.

Thank you

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE



BITS Pilani
Pilani Campus

ESZG512/MELZG526/SEZG516
Embedded System Design
Contact Session 12

Contents

Introduction to Interrupts

- A combination of software and hardware to force the processor to stop its current activity and begin to execute a particular piece of code called an interrupt service routine (ISR).
- An ISR responds to a specific event generated by either hardware or software.
- When an ISR completes, the processor automatically resumes the activity that had been halted.
- Interrupts are widely used to respond to both internal and external hardware requests efficiently.
- Interrupts also allow a processor to perform multiple tasks simultaneously.
- Multiple tasks can be handled in a preemptive or non-preemptive manner.

Nested vectored interrupt controller (NVIC)

NVIC features

- 82 maskable interrupt channels for STM32F405xx/07xx and STM32F415xx/17xx, and up to 91 maskable interrupt channels for STM32F42xxx and STM32F43xxx (not including the 16 interrupt lines of Cortex®-M4 with FPU)
- 16 programmable priority levels (4 bits of interrupt priority are used)
- Low-latency exception and interrupt handling
- Power management control
- Implementation of system control registers

Nested vectored interrupt controller (NVIC)

- The NVIC and the processor core interface are closely coupled, which enables low latency interrupt processing and efficient processing of late arriving interrupts.
- All interrupts including the core exceptions are managed by the NVIC.
- For more information on exceptions and NVIC programming, refer to programming manual PM0214.

BITS Pilani, Pilani Campus

Interrupt Numbers

- Cortex-M processors support up to 256 types of interrupts. Each interrupt type, excluding the reset interrupt, is identified by a unique number, ranging from -15 to 240.
- Interrupt numbers are defined by ARM and chip manufacturers collectively.
- These numbers are fixed and software cannot re-define them. Interrupt numbers are divided into two groups.

BITS Pilani, Pilani Campus

Interrupt Numbers

1. The first 16 interrupts are system interrupts, also called **system exceptions**.
   - Exceptions are the interrupts that come from the processor core. These interrupt numbers are defined by ARM.
   - Specifically, the ARM CMSIS library defines all system exceptions by using negative values.
   - CMSIS stands for Cortex Microcontroller Software Interface Standard.

BITS Pilani, Pilani Campus

Interrupt Numbers

2. The remaining 240 interrupts are peripheral interrupts, also called non-system exceptions.
   - ➢The peripheral interrupt numbers start at 0.
   - ➢Peripheral interrupts are defined by chip manufacturers.
   - ➢The total number of peripheral interrupts supported varies among chips.

➢ This numbering scheme allows software to distinguish system exceptions and peripheral interrupts easily.

Vector table for STM32F405xx/07xx

Table 61. Vector table for STM32F405xx/07xx and STM32F415xx/17xx

Position	Priority	Type of priority	Acronym	Description	Address
	-	-	-	Reserved	0x0000 0000
	-3	fixed	Reset	Reset	0x0000 0004
	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000 0008
	-1	fixed	HardFault	All class of fault	0x0000 000C
	0	settable	MemManage	Memory management	0x0000 0010
	1	settable	BusFault	Pre-fetch fault, memory access fault	0x0000 0014
	2	settable	UsageFault	Undefined instruction or illegal state	0x0000 0018
	-	-	-	Reserved	0x0000 001C - 0x0000 002B
	3	settable	SVCall	System service call via SWI instruction	0x0000 002C
	4	settable	Debug Monitor	Debug Monitor	0x0000 0030
	-	-	-	Reserved	0x0000 0034
	5	settable	PendSV	Pendable request for system service	0x0000 0038
	6	settable	SysTick	System tick timer	0x0000 003C

Vector table for STM32F405xx/07xx

0	7	settable	WWDG	Window Watchdog interrupt	0x0000 0040
1	8	settable	PVD	PVD through EXTI line detection interrupt	0x0000 0044
2	9	settable	TAMP_STAMP	Tamper and TimeStamp interrupts through the EXTI line	0x0000 0048
3	10	settable	RTC_WKUP	RTC Wakeup interrupt through the EXTI line	0x0000 004C
4	11	settable	FLASH	Flash global interrupt	0x0000 0050
5	12	settable	RCC	RCC global interrupt	0x0000 0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000 0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000 005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000 0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000 0064
10	17	settable	EXTI4	EXTI Line4 interrupt	0x0000 0068
11	18	settable	DMA1_Stream0	DMA1 Stream0 global interrupt	0x0000 006C

Table 61. Vector table for STM32F405xx/07xx and STM32F415xx/17xx (continued)

Position	Priority	Type of priority	Acronym	Description	Address
12	19	settable	DMA1_Stream1	DMA1 Stream1 global interrupt	0x0000 0070
13	20	settable	DMA1_Stream2	DMA1 Stream2 global interrupt	0x0000 0074
14	21	settable	DMA1_Stream3	DMA1 Stream3 global interrupt	0x0000 0078
15	22	settable	DMA1_Stream4	DMA1 Stream4 global interrupt	0x0000 007C
16	23	settable	DMA1_Stream5	DMA1 Stream5 global interrupt	0x0000 0080
17	24	settable	DMA1_Stream6	DMA1 Stream6 global interrupt	0x0000 0084
18	25	settable	ADC	ADC1, ADC2 and ADC3 global interrupts	0x0000 0088
19	26	settable	CAN1_TX	CAN1 TX interrupts	0x0000 008C
20	27	settable	CAN1_RX0	CAN1 RX0 interrupts	0x0000 0090
21	28	settable	CAN1_RX1	CAN1 RX1 interrupt	0x0000 0094
22	29	settable	CAN1_SCE	CAN1 SCE interrupt	0x0000 0098
23	30	settable	EXTI9_5	EXTI Line[9:5] interrupts	0x0000 009C
24	31	settable	TIM1_BRK_TIM9	TIM1 Break interrupt and TIM9 global interrupt	0x0000 00A0
25	32	settable	TIM1_UP_TIM10	TIM1 Update interrupt and TIM10 global interrupt	0x0000 00A4
26	33	settable	TIM1_TRG_COM_TIM11	TIM1 Trigger and Commutation interrupts and TIM11 global interrupt	0x0000 00A8
27	34	settable	TIM1_CC	TIM1 Capture Compare interrupt	0x0000 00AC
28	35	settable	TIM2	TIM2 global interrupt	0x0000 00B0
29	36	settable	TIM3	TIM3 global interrupt	0x0000 00B4
30	37	settable	TIM4	TIM4 global interrupt	0x0000 00B8
31	38	settable	I2C1_EV	I^2C1 event interrupt	0x0000 00BC
32	39	settable	I2C1_ER	I^2C1 error interrupt	0x0000 00C0
33	40	settable	I2C2_EV	I^2C2 event interrupt	0x0000 00C4
34	41	settable	I2C2_ER	I^2C2 error interrupt	0x0000 00C8
35	42	settable	SPI1	SPI1 global interrupt	0x0000 00CC
36	43	settable	SPI2	SPI2 global interrupt	0x0000 00D0
37	44	settable	USART1	USART1 global interrupt	0x0000 00D4
38	45	settable	USART2	USART2 global interrupt	0x0000 00D8
39	46	settable	USART3	USART3 global interrupt	0x0000 00DC

Table 61. Vector table for STM32F405xx/07xx and STM32F415xx/17xx (continued)

Position	Priority	Type of priority	Acronym	Description	Address
40	47	settable	EXTI15_10	EXTI Line[15:10] interrupts	0x0000 00E0
41	48	settable	RTC_Alarm	RTC Alarms (A and B) through EXTI line interrupt	0x0000 00E4
42	49	settable	OTG_FS WKUP	USB On-The-Go FS Wakeup through EXTI line interrupt	0x0000 00E8
43	50	settable	TIM8_BRK_TIM12	TIM8 Break interrupt and TIM12 global interrupt	0x0000 00EC
44	51	settable	TIM8_UP_TIM13	TIM8 Update interrupt and TIM13 global interrupt	0x0000 00F0
45	52	settable	TIM8_TRG_COM_TIM14	TIM8 Trigger and Commutation interrupts and TIM14 global interrupt	0x0000 00F4
46	53	settable	TIM8_CC	TIM8 Capture Compare interrupt	0x0000 00F8
47	54	settable	DMA1_Stream7	DMA1 Stream7 global interrupt	0x0000 00FC
48	55	settable	FSMC	FSMC global interrupt	0x0000 0100
49	56	settable	SDIO	SDIO global interrupt	0x0000 0104
50	57	settable	TIM5	TIM5 global interrupt	0x0000 0108
51	58	settable	SPI3	SPI3 global interrupt	0x0000 010C
52	59	settable	UART4	UART4 global interrupt	0x0000 0110
53	60	settable	UART5	UART5 global interrupt	0x0000 0114
54	61	settable	TIM6_DAC	TIM6 global interrupt, DAC1 and DAC2 underrun error interrupts	0x0000 0118
55	62	settable	TIM7	TIM7 global interrupt	0x0000 011C
56	63	settable	DMA2_Stream0	DMA2 Stream0 global interrupt	0x0000 0120
57	64	settable	DMA2_Stream1	DMA2 Stream1 global interrupt	0x0000 0124
58	65	settable	DMA2_Stream2	DMA2 Stream2 global interrupt	0x0000 0128
59	66	settable	DMA2_Stream3	DMA2 Stream3 global interrupt	0x0000 012C
60	67	settable	DMA2_Stream4	DMA2 Stream4 global interrupt	0x0000 0130
61	68	settable	ETH	Ethernet global interrupt	0x0000 0134
62	69	settable	ETH_WKUP	Ethernet Wakeup through EXTI line interrupt	0x0000 0138
63	70	settable	CAN2_TX	CAN2 TX interrupts	0x0000 013C
64	71	settable	CAN2_RX0	CAN2 RX0 interrupts	0x0000 0140

Table 61. Vector table for STM32F405xx/07xx and STM32F415xx/17xx (continued)

Position	Priority	Type of priority	Acronym	Description	Address
65	72	settable	CAN2_RX1	CAN2 RX1 interrupt	0x0000 0144
66	73	settable	CAN2_SCE	CAN2 SCE interrupt	0x0000 0148
67	74	settable	OTG_FS	USB On The Go FS global interrupt	0x0000 014C
68	75	settable	DMA2_Stream5	DMA2 Stream5 global interrupt	0x0000 0150
69	76	settable	DMA2_Stream6	DMA2 Stream6 global interrupt	0x0000 0154
70	77	settable	DMA2_Stream7	DMA2 Stream7 global interrupt	0x0000 0158
71	78	settable	USART6	USART6 global interrupt	0x0000 015C
72	79	settable	I2C3_EV	I^2C3 event interrupt	0x0000 0160
73	80	settable	I2C3_ER	I^2C3 error interrupt	0x0000 0164
74	81	settable	OTG_HS_EP1_OUT	USB On The Go HS End Point 1 Out global interrupt	0x0000 0168
75	82	settable	OTG_HS_EP1_IN	USB On The Go HS End Point 1 In global interrupt	0x0000 016C
76	83	settable	OTG_HS_WKUP	USB On The Go HS Wakeup through EXTI interrupt	0x0000 0170
77	84	settable	OTG_HS	USB On The Go HS global interrupt	0x0000 0174
78	85	settable	DCMI	DCMI global interrupt	0x0000 0178
79	86	settable	CRYP	CRYP crypto global interrupt	0x0000 017C
80	87	settable	HASH_RNG	Hash and Rng global interrupt	0x0000 0180
81	88	settable	FPU	FPU global interrupt	0x0000 0184

Interrupt Service Routines

- When an interrupt is processed, the interrupt number is stored in the program status register (PSR).

Interrupt Service Routines

- An interrupt service routine (ISR), also called an interrupt handler, is a special subroutine that hardware invokes automatically in response to an interrupt.
- Each ISR has a default implementation in the system startup code (such as the assembly file startup_stm32xxxx. s).
- The default implementation of most ISRs is simply a dead loop.

Interrupt Vector Table

Interrupt Vector Table

- There is an interrupt service routine (ISR) associated with each type of interrupt.
- Cortex-M stores the starting memory address of every ISR in a special array called the interrupt vector table.

NVIC

- The nested vectored interrupt controller (NVIC) is built into Cortex-M cores to manage all interrupts.
- It offers three key functions:
  1. Enable and disable interrupts.
  2. Configure the preemption priority and sub-priority of a specific interrupt.
  3. Set and clear the handing bit of a specific interrupt.

NVIC

The hardware implementation of the NVIC registers is:

Table 45. NVIC register summary

Address	Name	Type	Required privilege	Reset value	Description
0xE000E100-0xE000E11F	NVIC_ISER0-NVIC_ISER7	RW	Privileged	0x00000000	*Table 4.3.2: Interrupt set-enable register x (NVIC_ISERx) on page 210*
0XE000E180-0xE000E19F	NVIC_ICER0-NVIC_ICER7	RW	Privileged	0x00000000	*Table 4.3.3: Interrupt clear-enable register x (NVIC_ICERx) on page 211*
0XE000E200-0xE000E21F	NVIC_ISPR0-NVIC_ISPR7	RW	Privileged	0x00000000	*Table 4.3.4: Interrupt set-pending register x (NVIC_ISPRx) on page 212*
0XE000E280-0xE000E29F	NVIC_ICPR0-NVIC_ICPR7	RW	Privileged	0x00000000	*Table 4.3.5: Interrupt clear-pending register x (NVIC_ICPRx) on page 213*
0xE000E300-0xE000E31F	NVIC_IABR0-NVIC_IABR7	RW	Privileged	0x00000000	*Table 4.3.6: Interrupt active bit register x (NVIC_IABRx) on page 214*
0xE000E400-0xE000E4EF	NVIC_IPR0-NVIC_IPR59	RW	Privileged	0x00000000	*Table 4.3.7: Interrupt priority register x (NVIC_IPRx) on page 215*
0xE000EF00	STIR	WO	Configurable	0x00000000	*Table 4.3.8: Software trigger interrupt register (NVIC_STIR) on page 216*

Note: The number of interrupts is product-dependent. Refer to reference manual/datasheet of relevant STM32 product for related information.

NVIC

Enable and Disable Peripheral Interrupts

- Cortex-M has eight 32-bit Interrupt Set-Enable Registers (ISER), ISER0 - ISER7, and eight 32-bit Interrupt Clear-enable Register (ICER), ICER0 - ICER7.
- Writing a bit to 1 in ISER and ICER enables and disables the corresponding interrupt, respectively.

Interrupt Set Enable Register 0 (ISER0)

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Enable Bit	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Interrupt Number	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	I2C1_EV	TIM4	TIM3	TIM2	TIM1_CC	TIM1_TRG	TIM1_UP	TIM1_BRK	EXTI9_5	CAN1_SCE	CAN1_RX1	CAN1_RX0	CAN1_TX	ADC1_ADC2	DMA1_CH7	DMA1_CH6	DMA1_CH5	DMA1_CH4	DMA1_CH3	DMA1_CH2	DMA1_CH1	EXTI4	EXTI3	EXTI2	EXTI1	EXTI0	RCC	FLASH	RTC_WKUP	TAMPER_STAMP	PVD	WWDG

NVIC

- Each bit in an ISER register can enable one peripheral interrupt.
- The interrupt number of peripheral interrupts ranges from 0 to 240.
- Not all peripheral interrupts are used.
- For example, STM32F407 has only 82 interrupts.

Interrupt Priority

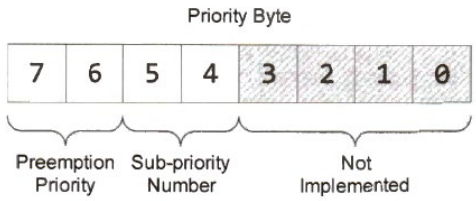
- Priority determines the order of interrupts to be serviced. Each interrupt has an interrupt priority register (IP), which has a width of 8 bits.
- Each consists of two fields : the preemption priority number and the sub-priority number.
- A lower value of a priority number represents a higher priority
- When there are multiple pending interrupts, the interrupt that has the lowest interrupt number is serviced first by the processor.

Interrupt Priority

- The preemption priority number defines the priority for preemption.
- If the processor receives a new interrupt that has a lower preemption priority number than the preemption priority number of the current interrupt in progress, the current interrupt is stopped, and the processor starts to serve the new interrupt.
- The preempted interrupt is resumed after the new interrupt handler routine completes.

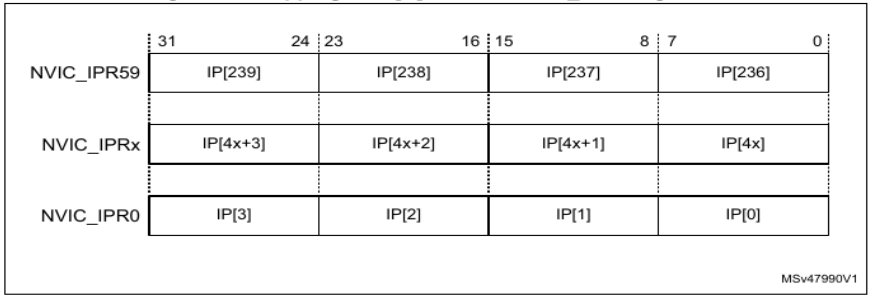
Interrupt Priority

- While Cortex-M processors use eight bits to store the priority number, STM32F407 processors only implement four bits.
- Thus, the STM32L microcontroller only supports 16 interrupt priority levels, ranging from 0 to 15.
- For a different Cortex-M processor, the interrupt priority byte might be different.



Interrupt Priority

Figure 19. Mapping of IP[N] fields in NVIC_IPRx registers



The following table shows the bit assignment of any NVIC_IPRx register. Each IP[N] field order can be expressed as N = 4 * x + byte offset.

Table 47. NVIC_IPRx bit assignment

Bits	Name	Function
[31:24]	Priority, byte offset = 3	Each priority field holds a priority value, 0-255. The lower the value, the greater the priority of the corresponding interrupt. The processor implements only bits[7:4] of each field, bits[3:0] read as zero and ignore writes.
[23:16]	Priority, byte offset = 2	
[15:8]	Priority, byte offset = 1	
[7:0]	Priority, byte offset = 0	

STM32F407x Timer interrupt programming

1. Enable GPIOD clock *(RCC->AHB1ENR)*
2. Set PD12 pin as output mode *(GPIOD->MODER)*
3. Configure timer2 for 1Hz clock generation with interrupt:
    1. Enable Timer2 clock *(RCC->APB1ENR)*
    2. Set PSC value as 1600-1 *(TIM2->PSC)*
    3. Set ARR value as 10000-1 *(TIM2->ARR)*
    4. Clear counter *(TIM2->CNT)*
    5. Enable Timer2 *(TIM2->CR1)*
    6. Enable TIM interrupt
    7. Enable TIM interrupt in NVIC
4. IRQ Handler: 1. Clear update interrupt flag *(TIM2->SR)*
   2. Toggle PD12 *(GPIOD->ODR)*

In main function while loop you can do other task.


```
#include "stm32f4xx.h"              // Device header
#define PIN12       (1U<<12)
#define LED         PIN12
int toggle = 1;
void TIM2_IRQHandler(void);

int main(void)
{
   RCC->AHB1ENR |= (1<<3);                  //Enable GPIOD clock
   GPIOD->MODER |=0x01000000;               //PD.12 set pin to output mode
   // configure TIM2 to wrap around at 1 Hz
   RCC->APB1ENR |= (1U<<0);                 /* enable TIM2 clock */
   TIM2->PSC = 1600 - 1;                    /* divided by 1600 i.e. 16000000/1600=10000*/
   TIM2->ARR = 10000 - 1;                   /* divided by 10000 i.e. 10000/10000=1 */
   TIM2->CNT = 0;                           /* clear counter */
   TIM2->CR1 = (1U<<0);                     /* enable TIM2 */
   TIM2->DIER |= (1<<0);                    /*Enable TIM interrupt*/
   /*Enable TIM interrupt in NVIC*/
    WordOffset = 28 >> 5;                   //   Word Offset = IRQn/32
    BitOffset = 28 & 0x1F;                  //   Bit Offset = IRQn mod 32
    NVIC->ISER[WordOffset]= 1<< BitOffset ;   //   Enable interrupt
   while (1)
   {

   }
}
void TIM2_IRQHandler(void)
{
   TIM2->SR &= ~(1<<0);         /*Clear update interrupt flag*/
   GPIOD->ODR ^= LED;           /*Toggle LED*/
   toggle = ~ toggle;
}
```


GPIOD->MODER &=0x0C000000; //Clear pin mode

GPIOD->MODER |=0x01000000; //PD.12 set pin to output mode

8.4.1 GPIO port mode register (GPIOx_MODER) (x = A..I/J/K)

Address offset: 0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y:2y+1 **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O direction mode.
00: Input (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

RCC->AHB1ENR |= (1<<3); //Enable GPIOD clock

6.3.10 RCC AHB1 peripheral clock register (RCC_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reser-ved	OTGHS ULPIEN	OTGHSEN	ETHMACPTPEN	ETHMACRXEN	ETHMACTXEN	ETHMACEN	Res.	**DMA2DEN**	DMA2EN	DMA1EN	CCMDATARAMEN	Res.	BKPSRAMEN	Reserved	
	rw	rw	rw	rw	rw	rw		rw	rw	rw			rw		
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Res.	GPIOKEN	GPIOJEN	GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

// configure TIM2 to wrap around at 1 Hz
RCC->APB1ENR |= (1U<<0); /* enable TIM2 clock */

6.3.13 RCC APB1 peripheral clock enable register (RCC_APB1ENR)

Address offset: 0x40

Reset value: 0x0000 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
UART8 EN	UART7 EN	DAC EN	PWR EN	Reser-ved	CAN2 EN	CAN1 EN	Reser-ved	I2C3 EN	I2C2 EN	I2C1 EN	UART5 EN	UART4 EN	USART 3 EN	USART 2 EN	Reser-ved
rw	rw	rw	rw		rw	rw		rw	rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SPI3 EN	SPI2 EN	Reserved		WWDG EN	Reserved		TIM14 EN	TIM13 EN	TIM12 EN	TIM7 EN	TIM6 EN	TIM5 EN	TIM4 EN	TIM3 EN	TIM2 EN
rw	rw			rw			rw	rw	rw	rw	rw	rw	rw	rw	rw

TIM2->PSC = 1600 - 1; /* divided by 1600 i.e. 16000000/1600=10000*/

18.4.11 TIMx prescaler (TIMx_PSC)

Address offset: 0x28

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **PSC[15:0]**: Prescaler value

The counter clock frequency CK_CNT is equal to f_{CK_PSC} / (PSC[15:0] + 1).

PSC contains the value to be loaded in the active prescaler register at each update event (including when the counter is cleared through UG bit of TIMx_EGR register or through trigger controller when configured in "reset mode").

TIM2->ARR = 10000 - 1; /* divided by 10000 i.e. 10000/10000=1 */

18.4.12 TIMx auto-reload register (TIMx_ARR)

Address offset: 0x2C

Reset value: 0xFFFF FFFF

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ARR[31:16] (depending on timers)															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **ARR[15:0]**: Auto-reload value

ARR is the value to be loaded in the actual auto-reload register.

Refer to the *Section 18.3.1: Time-base unit* for more details about ARR update and behavior.

The counter is blocked while the auto-reload value is null.

TIM2->CNT = 0; /* clear counter */

18.4.10 TIMx counter (TIMx_CNT)

Address offset: 0x24

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNT[31:16] (depending on timers)															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 15:0 **CNT[15:0]**: Counter value

TIM2->CR1 = (1<<0); /* enable TIM2 */

18.4.1 TIMx control register 1 (TIMx_CR1)

Address offset: 0x00

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 0 **CEN**: Counter enable
0: Counter disabled
1: Counter enabled

Note: External clock, gated mode and encoder mode can work only if the CEN bit has been previously set by software. However trigger mode can set the CEN bit automatically by hardware.

CEN is cleared automatically in one-pulse mode, when an update event occurs.

TIM2->DIER |= (1<<0); /*Enable TIM interrupt*/

18.4.4 TIMx DMA/Interrupt enable register (TIMx_DIER)

Address offset: 0x0C

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	Res	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res	CC4IE	CC3IE	CC2IE	CC1IE	UIE
	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

Bit 0 **UIE**: Update interrupt enable
0: Update interrupt disabled
1: Update interrupt enabled


```
WordOffset = 28 >> 5;                 //  Word Offset = IRQn/32
BitOffset = 28 & 0x1F;                //  Bit Offset = IRQn mod 32
NVIC->ISER[WordOffset]= 1<< BitOffset ;   //  Enable interrupt
```

4.3.2 Interrupt set-enable register x (NVIC_ISERx)

Address offset: 0x100 + 0x04 * x, (x = 0 to 7)

Reset value: 0x0000 0000

Required privilege: Privileged

NVIC_ISER0 bits 0 to 31 are for interrupt 0 to 31, respectively

NVIC_ISER1 bits 0 to 31 are for interrupt 32 to 63, respectively

....

NVIC_ISER6 bits 0 to 31 are for interrupt 192 to 223, respectively

NVIC_ISER7 bits 0 to 15 are for interrupt 224 to 239, respectively

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SETENA[31:16]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SETENA[15:0]															
rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs	rs

Bits 31:0 **SETENA**: Interrupt set-enable bits.
Write:
0: No effect
1: Enable interrupt
Read:
0: Interrupt disabled
1: Interrupt enabled.

In TIM2_IRQHandler()

TIM2->SR &= ~(1<<0); // Clear UIF

18.4.5 TIMx status register (TIMx_SR)

Address offset: 0x10

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CC4OF	CC3OF	CC2OF	CC1OF	Reserved		TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0			rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

Bit 0 **UIF**: Update interrupt flag
" This bit is set by hardware on an update event. It is cleared by software.
0: No update occurred.
1: Update interrupt pending. This bit is set by hardware when the registers are updated:
" At overflow or underflow (for TIM2 to TIM5) and if UDIS=0 in the TIMx_CR1 register.
" When CNT is reinitialized by software using the UG bit in TIMx_EGR register, if URS=0 and UDIS=0 in the TIMx_CR1 register.
When CNT is reinitialized by a trigger event (refer to the synchro control register description), if URS=0 and UDIS=0 in the TIMx_CR1 register.

GPIOD->ODR ^= LED;

8.4.6 GPIO port output data register (GPIOx_ODR) (x = A..I/J/K)

Address offset: 0x14

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy**: Port output data (y = 0..15)

These bits can be read and written by software.

Note: For atomic bit set/reset, the ODR bits can be individually set and reset by writing to the GPIOx_BSRR register (x = A..I/J/K).

External interrupt / event controller (EXTI)

- The external interrupt/event controller consists of up to 23 edge detectors for generating event/interrupt requests.
- Each input line can be independently configured to select the type (interrupt or event) and the corresponding trigger event (rising or falling or both).
- Each line can also masked independently.
- A pending register maintains the status line of the interrupt requests.

External interrupt / event controller (EXTI)

EXTI main features

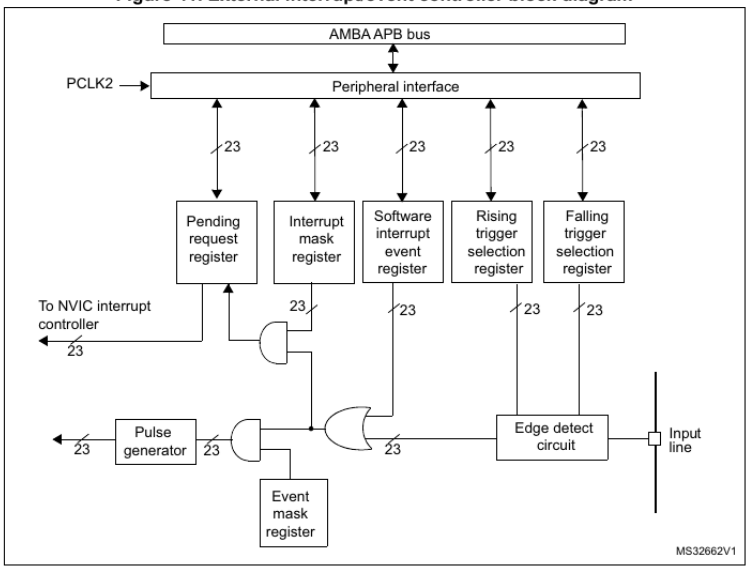
The main features of the EXTI controller are the following:

- independent trigger and mask on each interrupt/event line
- dedicated status bit for each interrupt line
- generation of up to 23 software event/interrupt requests
- detection of external signals with a pulse width lower than the APB2 clock period.

Refer to the electrical characteristics section of the STM32F4xx datasheets for details on this parameter.

EXTI block diagram

Figure 41. External interrupt/event controller block diagram



Thank you

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE

BITS Pilani
Pilani Campus

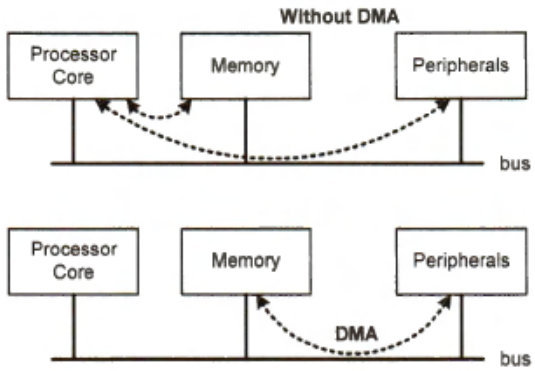
ESZG512/MELZG526/SEZG516
Embedded System Design
Contact Session 13

Contents

- Introduction to DMA
- STM32F407 DMA Features
- STM32F407 DMA Block diagram
- Programming of DMA memory to memory transfer

Direct Memory Access (DMA)

- Direct memory access (DMA) is a useful technique for transferring data between peripherals and memory, or between memory and memory, without using many processor cycles.



Direct Memory Access (DMA)

- The processor needs to program the DMA controller and send a command to start the DMA transfer.
- However, during the transfer process, the processor is not involved and can execute other tasks.
- Therefore, DMA is a very efficient and widely-used approach to interface peripherals.
- For slow peripherals, DMA releases the processor from waiting for peripheral data and allows the CPU to serve other tasks.
- For fast peripherals, such as ADC and DAC, DMA improves the data transfer throughput because memory accesses take place without the involvement of the processor.

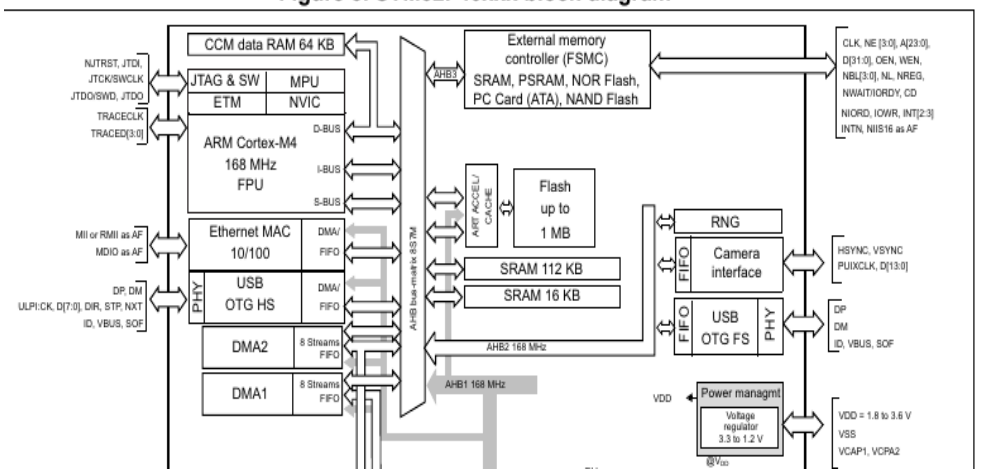
Direct Memory Access (DMA)

- For high-speed peripherals, DMA can help significantly reduce the rate at which interrupts are generated.
- Thus, DMA decreases the performance overhead from interrupts.
- With DMA support, a peripheral does not need to have local memory to store data.
- Instead, data can be efficiently stored in data memory via DMA.
- This design not only reduces the cost but also improves the energy efficiency of embedded systems.

Direct Memory Access (DMA)

DMA controllers reside on the AHB bus.

Figure 5. STM32F40xxx block diagram



Direct Memory Access (DMA)

- A DMA controller acts as a bus master and a bus slave.
- It has two ports.
- The slave port can accept data and commands from the processor when the processor sets up DMA transfers.
- The master port can initiate data transfer within the AHB bus or across the AHB/APB bridge.
- A DMA controller often manages multiple channels that can be programmed independently.
- While all channels of a DMA controller share the same interface to the AHB bus, each channel has its own dedicated interface to peripherals.
- Thus, multiple channels can perform DMA transfers simultaneously.

STM32F407xx DMA main features

- Dual AHB master bus architecture, one dedicated to memory accesses and one dedicated to peripheral accesses
- AHB slave programming interface supporting only 32-bit accesses
- 8 streams for each DMA controller, up to 8 channels (requests) per stream
- Four-word depth 32 first-in, first-out memory buffers (FIFOs) per stream, that can be used in FIFO mode or direct mode:
  - FIFO mode: with threshold level software selectable between 1/4, 1/2 or 3/4 of the FIFO size
  - Direct mode : Each DMA request immediately initiates a transfer from/to the memory. When it is configured in direct mode (FIFO disabled), to transfer data in memory-to- peripheral mode, the DMA preloads only one data from the memory to the internal FIFO to ensure an immediate data transfer as soon as a DMA request is triggered by a peripheral.

STM32F407xx DMA main features

- Each stream can be configured by hardware to be:
  - a regular channel that supports peripheral-to-memory, memory-to-peripheral and memory-to-memory transfers
  - a double buffer channel that also supports double buffering on the memory side
- Each of the 8 streams are connected to dedicated hardware DMA channels (requests)

STM32F407xx DMA main features

- Priorities between DMA stream requests are software-programmable (4 levels consisting of very high, high, medium, low) or hardware in case of equality (request 0 has priority over request 1, etc.)
- Each stream also supports software trigger for memory-to-memory transfers (only available for the DMA2 controller)
- Each stream request can be selected among up to 8 possible channel requests.
- This selection is software-configurable and allows several peripherals to initiate DMA requests

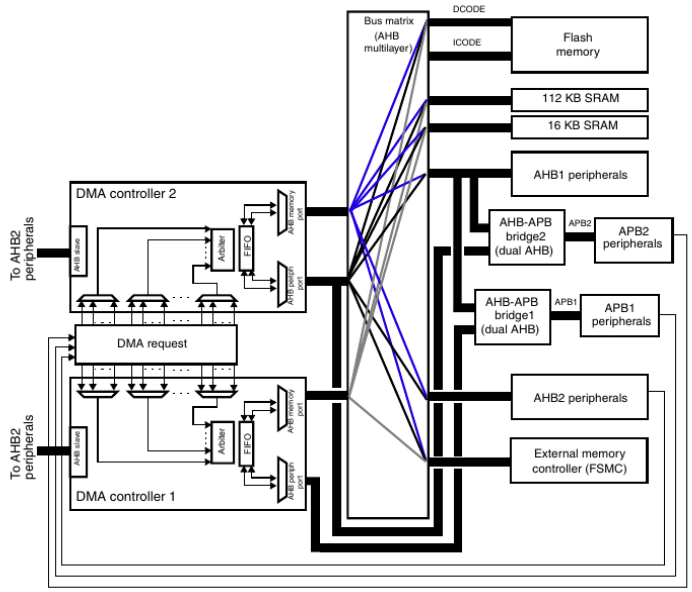
STM32F407xx DMA main features

- The number of data items to be transferred can be managed either by the DMA controller or by the peripheral:
  - DMA flow controller: the number of data items to be transferred is software-programmable from 1 to 65535
  - Peripheral flow controller: the number of data items to be transferred is unknown and controlled by the source or the destination peripheral that signals the end of the transfer by hardware
- Independent source and destination transfer width (byte, half-word, word): when the data widths of the source and destination are not equal, the DMA automatically packs/unpacks the necessary transfers to optimize the bandwidth. This feature is only available in FIFO mode
- Incrementing or non-incrementing addressing for source and destination

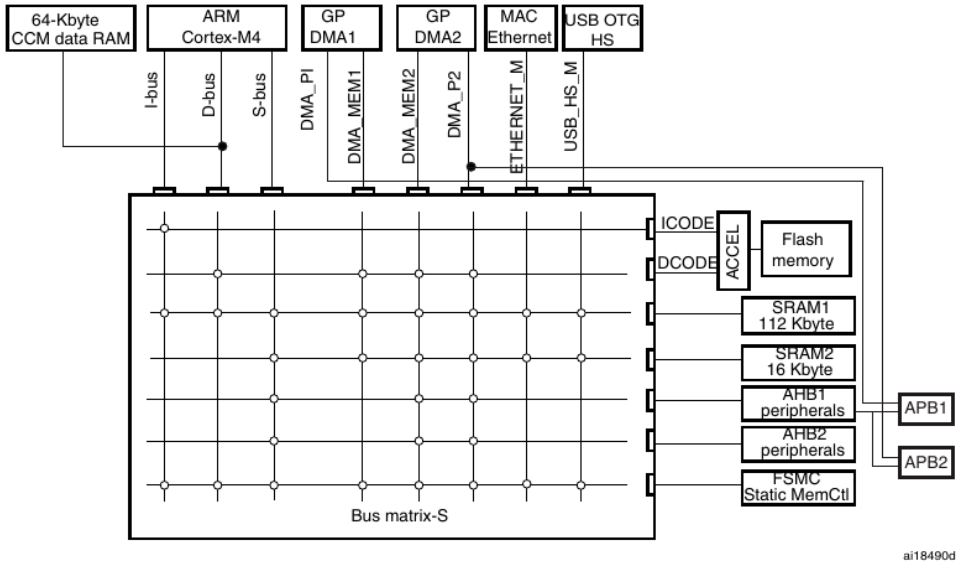
STM32F407xx DMA main features

- Supports incremental burst transfers of 4, 8 or 16 beats.
- The size of the burst is software-configurable, usually equal to half the FIFO size of the peripheral
- Each stream supports circular buffer management
- 5 event flags (DMA Half Transfer, DMA Transfer complete, DMA Transfer Error, DMA FIFO Error, Direct Mode Error) logically ORed together in a single interrupt request for each stream

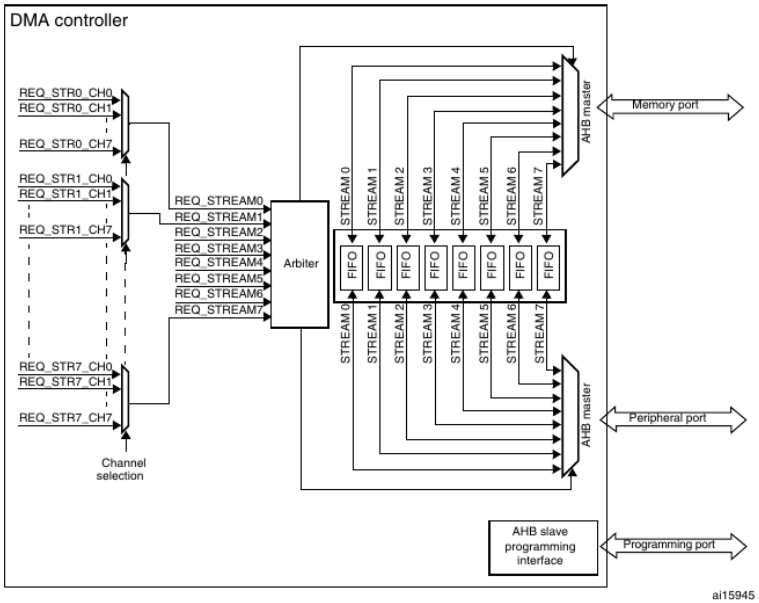
System implementation of the two DMA controllers



System architecture for STM32F405xx/07xx



DMA block diagram



- The STM32F407 microcontroller has two DMA controllers.
- Each DMA controller has eight DMA Streams. Together, they have 16 streams that may perform DMA operations independently.
- Each DMA stream has eight channels. The associations of peripherals to DMA channels/streams are hardwired and their connection depends on the product implementation.

Figure 35. Channel selection

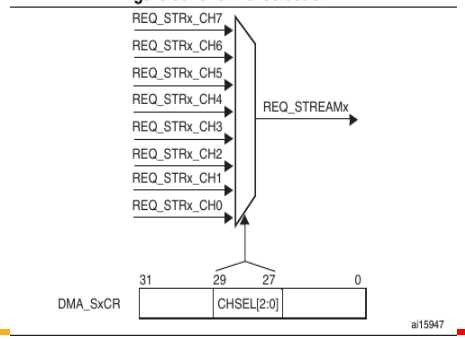


Table 42. DMA1 request mapping

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
Channel 1	I2C1_RX		TIM7_UP		TIM7_UP	I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4	UART5_RX	USART3_RX	UART4_RX	USART3_TX	UART4_TX	USART2_RX	USART2_TX	UART5_TX
Channel 5	UART8_TX(1)	UART7_TX(1)	TIM3_CH4 TIM3_UP	UART7_RX(1)	TIM3_CH1 TIM3_TRIG	TIM3_CH2	UART8_RX(1)	TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2		TIM5_UP	
Channel 7		TIM6_UP	I2C2_RX	I2C2_RX	USART3_TX	DAC1	DAC2	I2C2_TX

1. These requests are available on STM32F42xxx and STM32F43xxx only.

Arbiter

Table 43. DMA2 request mapping

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	ADC1	SAI1_A(1)	TIM8_CH1 TIM8_CH2 TIM8_CH3	SAI1_A(1)	ADC1	SAI1_B(1)	TIM1_CH1 TIM1_CH2 TIM1_CH3	
Channel 1		DCMI	ADC2	ADC2	SAI1_B(1)	SPI6_TX(1)	SPI6_RX(1)	DCMI
Channel 2	ADC3	ADC3		SPI5_RX(1)	SPI5_TX(1)	CRYP_OUT	CRYP_IN	HASH_IN
Channel 3	SPI1_RX		SPI1_RX	SPI1_TX		SPI1_TX		
Channel 4	SPI4_RX(1)	SPI4_TX(1)	USART1_RX	SDIO		USART1_RX	SDIO	USART1_TX
Channel 5		USART6_RX	USART6_RX	SPI4_RX(1)	SPI4_TX(1)		USART6_TX	USART6_TX
Channel 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
Channel 7		TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX(1)	SPI5_TX(1)	TIM8_CH4 TIM8_TRIG TIM8_COM

1. These requests are available on STM32F42xxx and STM32F43xxx.

- An arbiter manages the 8 DMA stream requests based on their priority for each of the two AHB master ports (memory and peripheral ports) and launches the peripheral/memory access sequences.
- Priorities are managed in two stages:
  - Software: each stream priority can be configured in the DMA_SxCR register.
  - There are four levels:
    - Very high priority
    - High priority
    - Medium priority
    - Low priority

Arbiter

- Hardware: If two requests have the same software priority level, the stream with the lower number takes priority over the stream with the higher number. For example, Stream 2 takes priority over Stream 4.

DMA streams

DMA streams

- Each of the 8 DMA controller streams provides a unidirectional transfer link between a source and a destination.

Each stream can be configured to perform:

- Regular type transactions: memory-to-peripherals, peripherals-to-memory or memory- to-memory transfers
- Double-buffer type transactions: double buffer transfers using two memory pointers for the memory (while the DMA is reading/writing from/to a buffer, the application can write/read to/from the other buffer).

DMA streams

- The amount of data to be transferred (up to 65535) is programmable and related to the source width of the peripheral that requests the DMA transfer connected to the peripheral AHB port.
- The register that contains the amount of data items to be transferred is decremented after each transaction.

Source, destination and transfer modes

Source, destination and transfer modes

- Both source and destination transfers can address peripherals and memories in the entire 4 GB area, at addresses comprised between 0x0000 0000 and 0xFFFF FFFF.
- The direction is configured using the DIR[1:0] bits in the DMA_SxCR register and offers three possibilities: memory-to-peripheral, peripheral-to-memory or memory-to-memory transfers. Table 44 describes the corresponding source and destination addresses.

Source, destination and transfer modes

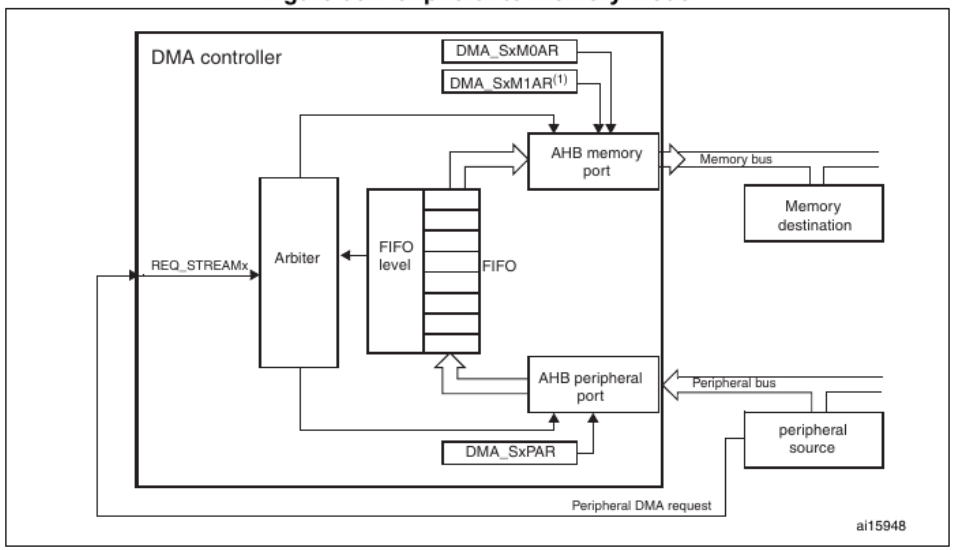
Table 44. Source and destination address

Bits DIR[1:0] of the DMA_SxCR register	Direction	Source address	Destination address
00	Peripheral-to-memory	DMA_SxPAR	DMA_SxM0AR
01	Memory-to-peripheral	DMA_SxM0AR	DMA_SxPAR
10	Memory-to-memory	DMA_SxPAR	DMA_SxM0AR
11	reserved	-	-

- When the data width (programmed in the PSIZE or MSIZE bits in the DMA_SxCR register) is a half-word or a word, respectively, the peripheral or memory address written into the DMA_SxPAR or DMA_SxM0AR/M1AR registers has to be aligned on a word or half-word address boundary, respectively.

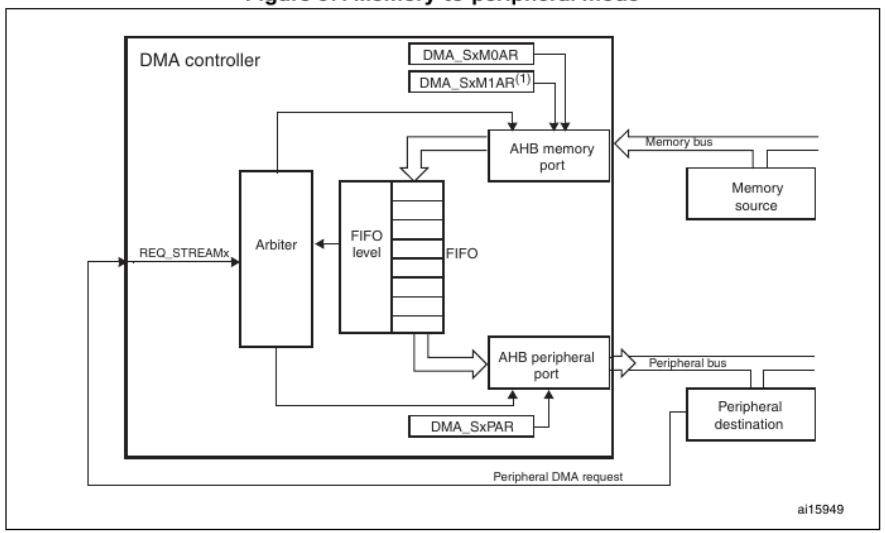
Peripheral-to-memory mode

Figure 36. Peripheral-to-memory mode



Memory-to-peripheral mode

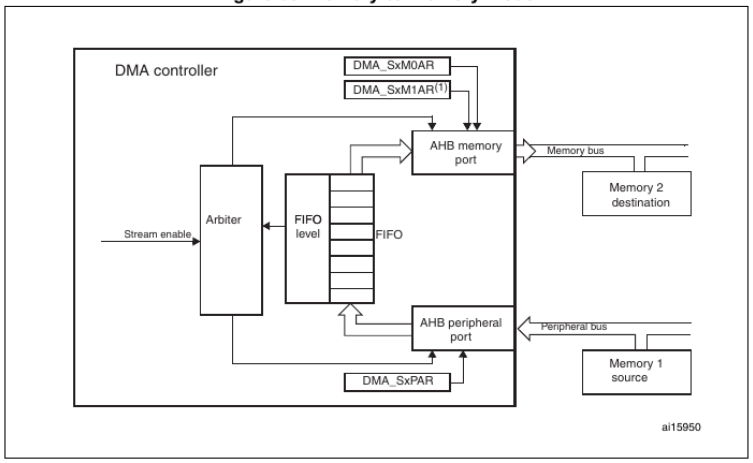
Figure 37. Memory-to-peripheral mode



1. For double-buffer mode.

Memory-to-memory mode

Figure 38. Memory-to-memory mode



1. For double-buffer mode.

The DMA channels can also work without being triggered by a request from a peripheral. This is the memory-to-memory mode

When memory-to-memory mode is used, the Circular and direct modes are not allowed. Only the DMA2 controller is able to perform memory-to-memory transfers.

Programming for DMA Memory to Memory transfer steps

/*Enable clock access to DMA 2 module*/

RCC->AHB1ENR |= (1<<22);

7.3.10 RCC AHB1 peripheral clock enable register (RCC_AHB1ENR)

Address offset: 0x30

Reset value: 0x0010 0000

Access: no wait state, word, half-word and byte access.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reser-ved	OTGHS ULPIEN	OTGHSEN	ETHMACPTPEN	ETHMACRXEN	ETHMACTXEN	ETHMACEN	Reserved		DMA2EN	DMA1EN	CCMDATARAMEN	Res.	BKPSRAMEN	Reserved	
	rw	rw	rw	rw	rw	rw			rw	rw			rw		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved			CRCEN	Reserved			GPIOIEN	GPIOHEN	GPIOGEN	GPIOFEN	GPIOEEN	GPIODEN	GPIOCEN	GPIOBEN	GPIOAEN
			rw				rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 22 **DMA2EN:** DMA2 clock enable

Set and cleared by software.

0: DMA2 clock disabled

1: DMA2 clock enabled

/*Disable DMA stream*/

DMA2_Stream0->CR = 0;

/*Wait until stream is disabled*/

while((DMA2_Stream0->CR & (1<<0)));

10.5.5 DMA stream x configuration register (DMA_SxCR) (x = 0..7)

This register is used to configure the concerned stream.

Address offset: 0x10 + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CHSEL[2:0]			MBURST [1:0]		PBURST[1:0]		Reser-ved	CT	DBM	PL[1:0]	
				rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PINCOS	MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR[1:0]		PFCTRL	TCIE	HTIE	TEIE	DMEIE	EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 0 **EN**: Stream enable / flag stream ready when read low

This bit is set and cleared by software.

0: Stream disabled

1: Stream enabled

/*Configure DMA parameters*/

/*Set MSIZE i.e. Memory data size to half-word*/

DMA2_Stream0->CR |= (1<<13);

DMA2_Stream0->CR &= ~(1<<14);

10.5.5 DMA stream x configuration register (DMA_SxCR) (x = 0..7)

This register is used to configure the concerned stream.

Address offset: 0x10 + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CHSEL[2:0]			MBURST [1:0]		PBURST[1:0]		Reser-ved	CT	DBM	PL[1:0]	
				rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PINCOS	MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR[1:0]		PFCTRL	TCIE	HTIE	TEIE	DMEIE	EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 14:13 **MSIZE[1:0]**: Memory data size

These bits are set and cleared by software.

00: byte (8-bit)

01: half-word (16-bit)

10: word (32-bit)

11: reserved

/*Set PSIZE i.e. Peripheral data size to half-word*/

DMA2_Stream0->CR |= (1<<11);

DMA2_Stream0->CR &= ~(1<<12);

10.5.5 DMA stream x configuration register (DMA_SxCR) (x = 0..7)

This register is used to configure the concerned stream.

Address offset: 0x10 + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CHSEL[2:0]			MBURST [1:0]		PBURST[1:0]		Reser-ved	CT	DBM	PL[1:0]	
				rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PINCOS	MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR[1:0]		PFCTRL	TCIE	HTIE	TEIE	DMEIE	EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 12:11 **PSIZE[1:0]**: Peripheral data size

These bits are set and cleared by software.

00: Byte (8-bit)

01: Half-word (16-bit)

10: Word (32-bit)

11: reserved

/*Enable Memory address increment*/

DMA2_Stream0->CR |= (1<<10);

10.5.5 DMA stream x configuration register (DMA_SxCR) (x = 0..7)

This register is used to configure the concerned stream.

Address offset: 0x10 + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CHSEL[2:0]			MBURST [1:0]		PBURST[1:0]		Reser-ved	CT	DBM	PL[1:0]	
				rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PINCOS	MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR[1:0]		PFCTRL	TCIE	HTIE	TEIE	DMEIE	EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 10 **MINC**: Memory increment mode

This bit is set and cleared by software.

0: Memory address pointer is fixed

1: Memory address pointer is incremented after each data transfer (increment is done according to MSIZE)

/*Enable peripheral address increment*/

DMA2_Stream0->CR |= (1<<9);

10.5.5 DMA stream x configuration register (DMA_SxCR) (x = 0..7)

This register is used to configure the concerned stream.

Address offset: 0x10 + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CHSEL[2:0]			MBURST [1:0]		PBURST[1:0]		Reser-ved	CT	DBM	PL[1:0]	
				rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PINCOS	MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR[1:0]		PFCTRL	TCIE	HTIE	TEIE	DMEIE	EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 9 **PINC**: Peripheral increment mode

This bit is set and cleared by software.

0: Peripheral address pointer is fixed

1: Peripheral address pointer is incremented after each data transfer (increment is done according to PSIZE)

/*Select memory to memory transfer*/

DMA2_Stream0->CR &= ~(1<<6);

DMA2_Stream0->CR |= (1<<7);

10.5.5 DMA stream x configuration register (DMA_SxCR) (x = 0..7)

This register is used to configure the concerned stream.

Address offset: 0x10 + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CHSEL[2:0]			MBURST [1:0]		PBURST[1:0]		Reser-ved	CT	DBM	PL[1:0]	
				rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PINCOS	MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR[1:0]		PFCTRL	TCIE	HTIE	TEIE	DMEIE	EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:6 **DIR[1:0]**: Data transfer direction

These bits are set and cleared by software.

00: Peripheral-to-memory

01: Memory-to-peripheral

10: Memory-to-memory

11: reserved

/*Enable transfer complete interrupt */

DMA2_Stream0->CR |= (1<<4);

10.5.5 DMA stream x configuration register (DMA_SxCR) (x = 0..7)

This register is used to configure the concerned stream.

Address offset: 0x10 + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CHSEL[2:0]			MBURST [1:0]		PBURST[1:0]		Reser-ved	CT	DBM	PL[1:0]	
				rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PINCOS	MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR[1:0]		PFCTRL	TCIE	HTIE	TEIE	DMEIE	EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 4 **TCIE**: Transfer complete interrupt enable

This bit is set and cleared by software.

0: TC interrupt disabled

1: TC interrupt enabled

/*Enable transfer Error interrupt */

DMA2_Stream0->CR |= (1<<2);

10.5.5 DMA stream x configuration register (DMA_SxCR) (x = 0..7)

This register is used to configure the concerned stream.

Address offset: 0x10 + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CHSEL[2:0]			MBURST [1:0]		PBURST[1:0]		Reser-ved	CT	DBM	PL[1:0]	
				rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PINCOS	MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR[1:0]		PFCTRL	TCIE	HTIE	TEIE	DMEIE	EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 2 **TEIE**: Transfer error interrupt enable
This bit is set and cleared by software.
0: TE interrupt disabled
1: TE interrupt enabled

/*Disable Direct mode*/

DMA2_Stream0->FCR |= (1<<2);

10.5.10 DMA stream x FIFO control register (DMA_SxFCR) (x = 0..7)

Address offset: 0x24 + 0x24 × *stream number*

Reset value: 0x0000 0021

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								FEIE	Reser ved	FS[2:0]			DMDIS	FTH[1:0]	
								rw		r	r	r	rw	rw	rw

Bit 2 **DMDIS**: Direct mode disable
This bit is set and cleared by software. It can be set by hardware.
0: Direct mode enabled
1: Direct mode disabled

/*Set DMA FIFO threshold*/

DMA2_Stream0->FCR |= (1<<0);

DMA2_Stream0->FCR |= (1<<1);

10.5.10 DMA stream x FIFO control register (DMA_SxFCR) (x = 0..7)

Address offset: 0x24 + 0x24 × *stream number*

Reset value: 0x0000 0021

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								FEIE	Reser ved	FS[2:0]			DMDIS	FTH[1:0]	
								rw		r	r	r	rw	rw	rw

Bits 1:0 **FTH[1:0]**: FIFO threshold selection
These bits are set and cleared by software.
00: 1/4 full FIFO
01: 1/2 full FIFO
10: 3/4 full FIFO
11: full FIFO

/*Enable DMA interrupt in NVIC*/

NVIC_EnableIRQ(DMA2_Stream0_IRQn);

/*Set peripheral address*/

DMA2_Stream0->PAR = src_buff;

10.5.7 DMA stream x peripheral address register (DMA_SxPAR) (x = 0..7)

Address offset: 0x18 + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PAR[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PAR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **PAR[31:0]**: Peripheral address

Base address of the peripheral data register from/to which the data will be read/written.

/*Set memory address*/

DMA2_Stream0->M0AR = dest_buff;

10.5.8 DMA stream x memory 0 address register (DMA_SxM0AR) (x = 0..7)

Address offset: 0x1C + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
M0A[31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
M0A[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **M0A[31:0]**: Memory 0 address

Base address of Memory area 0 from/to which the data will be read/written.
These bits are write-protected. They can be written only if:

- the stream is disabled (bit EN= '0' in the DMA_SxCR register) or
- the stream is enabled (EN='1' in DMA_SxCR register) and bit CT = '1' in the DMA_SxCR register (in Double buffer mode).

/*Set transfer length*/

DMA2_Stream0->NDTR = len;

10.5.6 DMA stream x number of data register (DMA_SxNDTR) (x = 0..7)

Address offset: 0x14 + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NDT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **NDT[15:0]**: Number of data items to transfer

Number of data items to be transferred (0 up to 65535). This register can be written only when the stream is disabled. When the stream is enabled, this register is read-only, indicating the remaining data items to be transmitted. This register decrements after each DMA transfer.
Once the transfer has completed, this register can either stay at zero (when the stream is in normal mode) or be reloaded automatically with the previously programmed value in the following cases:

- when the stream is configured in Circular mode.
- when the stream is enabled again by setting EN bit to '1'

If the value of this register is zero, no transaction can be served even if the stream is enabled.

/*Enable DMA stream*/

DMA2_Stream0->CR |= (1<<0);

10.5.5 DMA stream x configuration register (DMA_SxCR) (x = 0..7)

This register is used to configure the concerned stream.

Address offset: 0x10 + 0x18 × *stream number*

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CHSEL[2:0]			MBURST [1:0]		PBURST[1:0]		Reser-ved	CT	DBM	PL[1:0]	
				rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PINCOS	MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR[1:0]		PFCTRL	TCIE	HTIE	TEIE	DMEIE	EN
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bit 0 **EN**: Stream enable / flag stream ready when read low

This bit is set and cleared by software.
0: Stream disabled
1: Stream enabled

```
void DMA2_Stream0_IRQHandler(void)
{
  /*Check if transfer complete interrupt occured*/
  if((DMA2->LISR)& (1<<5))
```

10.5.1 DMA low interrupt status register (DMA_LISR)

Address offset: 0x00

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				TCIF3	HTIF3	TEIF3	DMEIF3	Reserved	FEIF3	TCIF2	HTIF2	TEIF2	DMEIF2	Reserved	FEIF2
r	r	r	r	r	r	r	r		r	r	r	r	r		r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				TCIF1	HTIF1	TEIF1	DMEIF1	Reserved	FEIF1	TCIF0	HTIF0	TEIF0	DMEIF0	Reserved	FEIF0
r	r	r	r	r	r	r	r		r	r	r	r	r		r

Bits 27, 21, 11, 5 **TCIFx**: Stream x transfer complete interrupt flag (x = 3..0)
This bit is set by hardware. It is cleared by software writing 1 to the corresponding bit in the DMA_LIFCR register.
0: No transfer complete event on stream x
1: A transfer complete event occurred on stream x

innovate achieve lead

```
{
  transfer_complete_flag = 1;
  /*Clear flag*/
  DMA2->LIFCR |= (1<<5);
}
```

10.5.3 DMA low interrupt flag clear register (DMA_LIFCR)

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CTCIF3	CHTIF3	CTEIF3	CDMEIF3	Reserved	CFEIF3	CTCIF2	CHTIF2	CTEIF2	CDMEIF2	Reserved	CFEIF2
				w	w	w	w		w	w	w	w	w		w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				CTCIF1	CHTIF1	CTEIF1	CDMEIF1	Reserved	CFEIF1	CTCIF0	CHTIF0	CTEIF0	CDMEIF0	Reserved	CFEIF0
				w	w	w	w		w	w	w	w	w		w

Bits 27, 21, 11, 5 **CTCIFx**: Stream x clear transfer complete interrupt flag (x = 3..0)
Writing 1 to this bit clears the corresponding TCIFx flag in the DMA_LISR register

innovate achieve lead

```
/*Check if transfer error occured*/
  if((DMA2->LISR)& (1<<3))
```

10.5.1 DMA low interrupt status register (DMA_LISR)

Address offset: 0x00

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				TCIF3	HTIF3	TEIF3	DMEIF3	Reserved	FEIF3	TCIF2	HTIF2	TEIF2	DMEIF2	Reserved	FEIF2
r	r	r	r	r	r	r	r		r	r	r	r	r		r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				TCIF1	HTIF1	TEIF1	DMEIF1	Reserved	FEIF1	TCIF0	HTIF0	TEIF0	DMEIF0	Reserved	FEIF0
r	r	r	r	r	r	r	r		r	r	r	r	r		r

Bits 25, 19, 9, 3 **TEIFx**: Stream x transfer error interrupt flag (x=3..0)
This bit is set by hardware. It is cleared by software writing 1 to the corresponding bit in the DMA_LIFCR register.
0: No transfer error on stream x
1: A transfer error occurred on stream x

innovate achieve lead

```
{
  /*Handle error by doing something*/
  /*Clear flag*/
  DMA2->LIFCR |= (1<<3);
}
```

10.5.3 DMA low interrupt flag clear register (DMA_LIFCR)

Address offset: 0x08

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved				CTCIF3	CHTIF3	CTEIF3	CDMEIF3	Reserved	CFEIF3	CTCIF2	CHTIF2	CTEIF2	CDMEIF2	Reserved	CFEIF2
				w	w	w	w		w	w	w	w	w		w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				CTCIF1	CHTIF1	CTEIF1	CDMEIF1	Reserved	CFEIF1	CTCIF0	CHTIF0	CTEIF0	CDMEIF0	Reserved	CFEIF0
				w	w	w	w		w	w	w	w	w		w

Bits 25, 19, 9, 3 **CTEIFx**: Stream x clear transfer error interrupt flag (x = 3..0)
Writing 1 to this bit clears the corresponding TEIFx flag in the DMA_LISR register

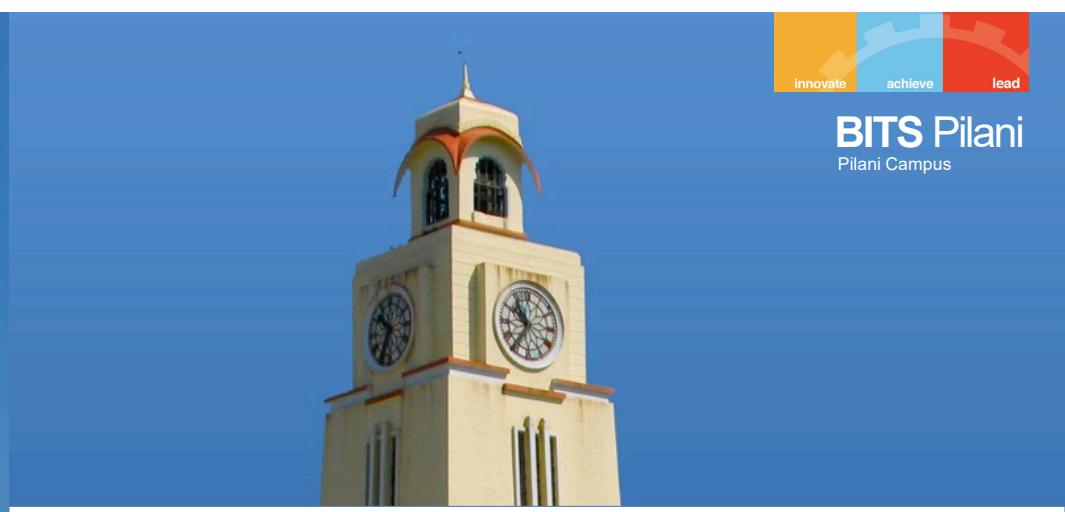
```
int main(void)
{

   dma2_mem2mem_configuration();
   dma_transfer_start((uint32_t)in_data_arr,(uint32_t)out_data_arr, BUFFER_SIZE);
   /*Wait until transfer complete*/
   while(!transfer_complete_flag);
   transfer_complete_flag = 0;

   while(1)
   {

   }

}
```


Thank you

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE



innovate achieve lead

BITS Pilani
Pilani Campus

ESZG512/MELZG526/SEZG516
Embedded System Design
Contact Session 14

innovate achieve lead

CS14 : (I2C_SPI_interfacing)

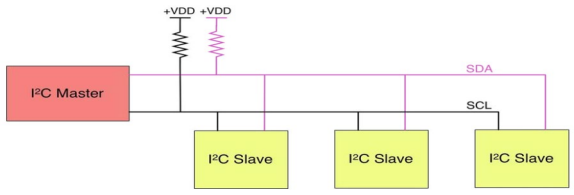
- I2C Protocol
- EEPROM Interfacing through I2C
- SPI Protocol

innovate achieve lead

BITS Pilani
Pilani | Dubai | Goa | Hyderabad

I2C

- Communication protocols required for communication between different hardware components of an Embedded system
- The Inter-Integrated Circuit (aka I²C - pronounced I-squared-C or very rarely I-two-C) is a hardware specification and protocol developed by the semiconductor division of Philips (now NXP Semiconductors) back in 1982.
- It is a master-slave, half-duplex, single-ended 8-bit oriented serial bus specification, which uses only two wires to interconnect a given number of slave devices to a master.



I2C

- The two wires forming an I²C bus are bidirectional open-drain lines, named Serial Data Line (SDA) and Serial Clock Line (SCL) respectively.
- The I²C protocol specifies that these two lines need to be pulled up with resistors.
- Being a protocol based on just two wires, there should be a way to address an individual slave device on the same bus.
- For this reason, I²C defines that each slave device provides a unique slave address for the given bus.
- The address may be 7- or 10-bit wide
- The protocol is also **byte oriented protocol** as a transmitting device expects to receive an acknowledgement from receiving device after every byte of data transfer.

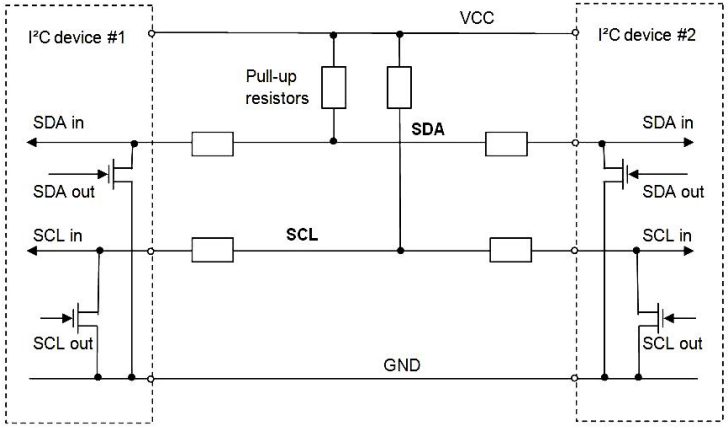
I2C

- I2C bus signals are SDA (Serial Data) and SCL (Serial Clock) . Both the signals are bidirectional in nature.
- The bus is also a 'WIRED AND' bus, which implies that if any of the device transmits a '0' , '0' will appear on the bus.

I²C modes

Mode[11]	Maximum speed	Maximum capacitance	Drive	Direction
Standard mode (Sm)	100 kbit/s	400 pF	Open drain*	Bidirectional
Fast mode (Fm)	400 kbit/s	400 pF	Open drain*	Bidirectional
Fast mode plus (Fm+)	1 Mbit/s	550 pF	Open drain*	Bidirectional
High-speed mode (Hs)	1.7 Mbit/s	400 pF	Open drain*	Bidirectional
High-speed mode (Hs)	3.4 Mbit/s	100 pF	Open drain*	Bidirectional
Ultra-fast mode (UFm)	5 Mbit/s	?	Push–pull	Unidirectional

I2C



I2C

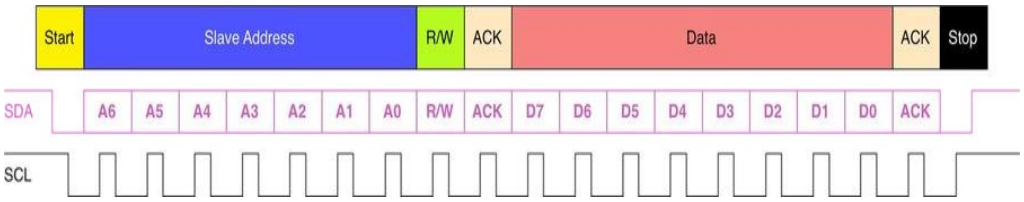
The I²C Protocol

- **In the I²C protocol all transactions are always initiated and completed by the master.**
- This is one of the few rules of this communication protocol to keep in mind while programming I²C devices.
- All messages exchanged over the I²C bus are broken up into two types of frame: an address frame, where the master indicates to which slave the message is being sent, and one or more data frames, which are 8-bit data messages passed from master to slave or vice versa.
- Data is placed on the SDA line after SCL goes low, and it is sampled after the SCL line goes high.

I2C

- The time between clock edges and data read/write is defined by devices on the bus and it vary from chip to chip.
- Both SDA and SCL are bidirectional lines, connected to a positive supply voltage via a current-source or pull-up resistors.
- When the bus is free, both lines are HIGH.
- The output stages of devices connected to the bus must have an open-drain or open-collector to perform the wired-AND function.
- The bus capacitance limits the number of interfaces connected to the bus.
- For a single master application, the master's SCL output can be a push-pull driver design if there are no devices on the bus that would stretch the clock.

I2C



START and STOP Condition

- All transactions begin with a START and are terminated by a STOP.
- A HIGH to LOW transition on the SDA line while SCL is HIGH defines a START condition.
- A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition.

I2C

- START and STOP conditions are always generated by the master.
- The bus is considered to be busy after the START condition.
- The bus is considered to be free again a certain time after the STOP condition.
- The bus stays busy if a repeated START (also called RESTART condition) is generated instead of a STOP condition.
- In this case, the START and RESTART conditions are functionally identical.

I2C

Byte Format

- Every word transmitted on the SDA line must be eight bits long, and this also includes the address frame.
- The number of bytes that can be transmitted per transfer is unrestricted.
- Each byte must be followed by an Acknowledge (ACK) bit.
- Data is transferred with the Most Significant Bit (MSB) first.
- If a slave cannot receive or transmit another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the master into a wait state.
- Data transfer continues when the slave is ready for another byte of data and releases clock line SCL.

Address Frame

- The address frame is always first in any new communication sequence.
- For a 7-bit address, the address is clocked out most significant bit (MSB) first, followed by a R/W bit indicating whether this is a read (1) or write (0) operation.
- The addressed slave should respond with an ACK bit.

Acknowledge (ACK) and Not Acknowledge (NACK)

- The ACK takes place after every byte. The ACK bit allows the receiver to signal the transmitter that the byte was successfully received and another byte may be sent.
- The master generates all clock pulses over the SCL line, including the ACK ninth clock pulse.

The ACK signal is defined as follows:

- The transmitter releases the SDA line during the acknowledge clock pulse so that the receiver can pull the SDA line LOW and it remains stable LOW during the HIGH period of this clock pulse.
- When SDA remains HIGH during this ninth clock pulse, this is defined as the Not Acknowledge (NACK) signal.
- The master can then generate either a STOP condition to abort the transfer, or a RESTART condition to start a new transfer.

There are five conditions leading to the generation of a NACK:

1. No receiver is present on the bus with the transmitted address so there is no device to respond with an acknowledge.

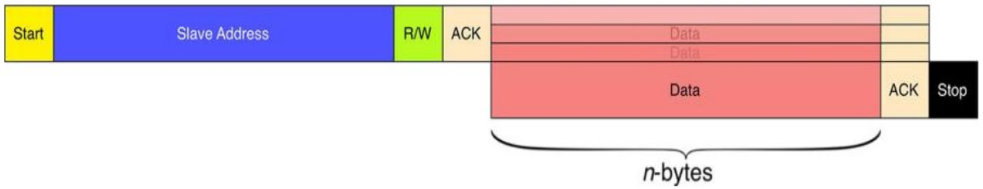
2. The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to start communication with the master.
3. During the transfer, the receiver gets data or commands that it does not understand.
4. During the transfer, the receiver cannot receive any more data bytes.
5. A master-receiver must signal the end of the transfer to the slave transmitter.

Data Frames

- After the address frame has been sent, data can begin being transmitted.
- The master will simply continue generating clock pulses on SCL at a regular interval, and the data will be placed on SDA by either the master or the slave, depending on whether the R/W bit indicated a read or write operation.
- Usually, the first or the first two bytes contains the address of the slave register to write to/read from.
- For example, for I²C EEPROMs the first two bytes following the address frame represent the address of the memory location involved in the transaction.

I2C

- Depending on the R/W bit, the successive bytes are filled by the master (if the R/W bit is set to 1) or the slave (if R/W bit is 0).
- The number of data frames is arbitrary, and most slave devices will auto-increment the internal register, meaning that subsequent reads or writes will come from the next register in line.
- This mode is also called sequential or burst mode and it is a way to speed up transfer speed.



I2C Arbitration

- If multiple masters find the bus idle at the same time, they might attempt to start the transfer by sending start sequence simultaneously.
- Ultimately, only one of the device can succeed in transmitting data .
- This is achieved via an arbitration process. Since the bus is WIRED AND,the device which transmits a '0' on bus first, will win the bus mastership.
- The address & R/W are included in the arbitration process.
- Each master device can read back what it has transmitted on bus.
- If the data available on the bus is matching with what it has transmitted, then the master continues transfer, else the master backs off.

I2C Arbitration : Scenario 1

- Master 1 attempts to transmit data to Slave A with address 0x32
- Master 2 attempts to transmit data to Slave B with address 0x27
- Both masters transmits start sequence followed by address bits as follows.
- Slave A address 0x32- 0110010
- Slave B address 0x27- 0100111
- Master 2 wins arbitration after transmission of 3rd bit (Transmission is MSB first), Master 1 backs off,
- Master 2 continues with address R/W and the data transfer

I2C Arbitration : Scenario 2

- Master 1 attempts to read data from Slave B (address 0x27)
- Master 2 attempts to write data to Slave B(address 0x27)
- Both masters transmits start sequence followed by address bits and R/W bit as follows.
- Slave A address 0x27-0100111, R/W=1
- Slave B address 0x27-0100111, R/W=0 (Win arbitration)
- Master 1 backs off after transmission of R/W, Master 2 continues with data transfer.

I2C Speed Modes

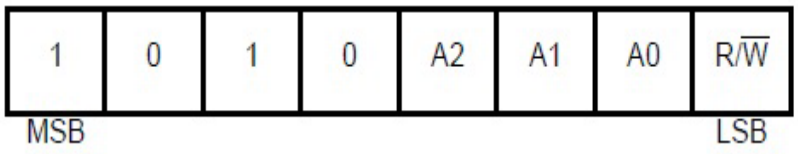
- I2C devices support various speed Modes.
- Compliant hardware guarantees that it can handle transmission speed up to the maximum clock rate specified by the mode.
- Standard mode: 100 kbit/s,
- Full Speed :400 kbit/s,
- Fast Mode: 1 Mbit/s,
- High Speed : 3.2 Mbit/s
- *Ultra Fast-mode : 5 Mbits/s*

- Many times single-master multiple-slave configurations may be sufficient.
- A master device is the one which initiates/terminates the transfer and generates the clock signals.
- Slave is a device addressed by the master.
- Both master as well as slave can receive and transmit data.

- There are many devices with built-in I2C bus: for example, EEPROM, RAM, LCD display drivers, ADC, DAC, etc.

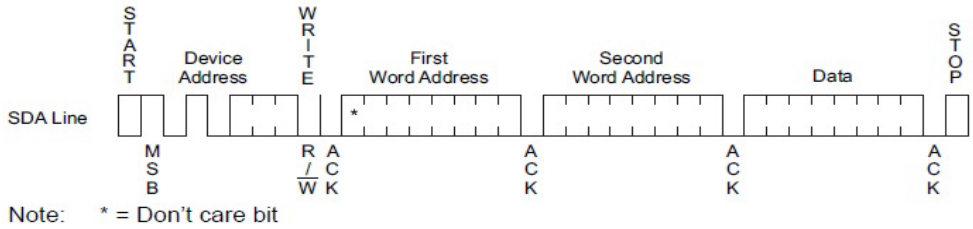
Device Addressing

- The 24C256 EEPROM requires an 8-bit device address word following a start condition to enable the chip for a read or write operation. Refer circuit schematic for $A_0A_1A_2$



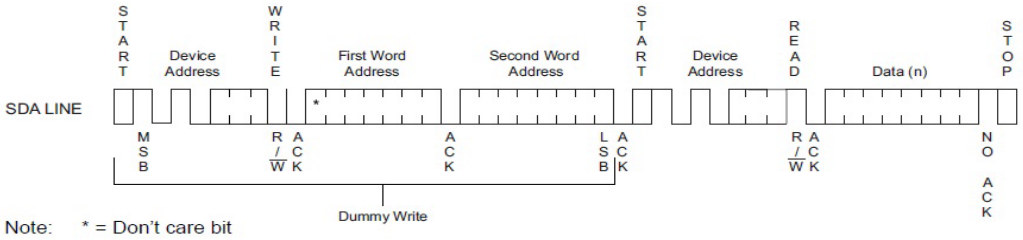
Write Protocol

- To write a single or multiple bytes of data into the EEPROM, we need to follow the protocol



Read Protocol

- To read a data byte from a particular address, we first need a dummy write sequence at that particular address, before we issue the read command.



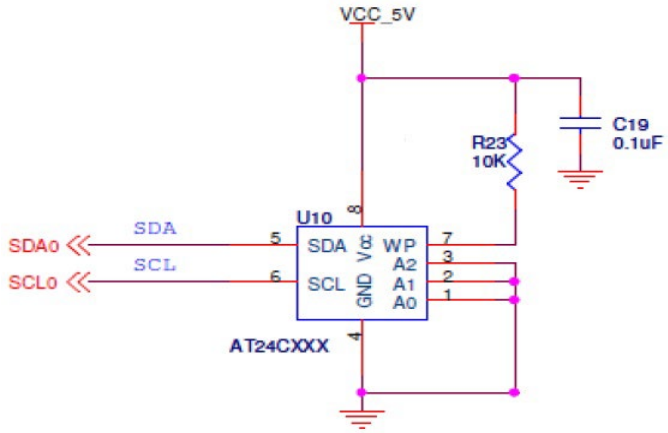
I2C Clock Frequency:

$$I^2C_{bitfrequency} = \frac{PCLK}{I2CSCLH + I2CSCLL}$$

LPC21XX I2C Registers

I2CONSET	**I2C Control Set Register**. When a one is written to a bit of this register, the corresponding bit in the I2C control register is set. Writing a zero has no effect on the corresponding bit in the I2C control register.
I2CONCLR	**I2C Control Clear Register**. When a one is written to a bit of this register, the corresponding bit in the I2C control register is cleared. Writing a zero has no effect on the corresponding bit in the I2C control register.
I2CSTAT	**I2C Status Register**. During I2C operation, this register provides detailed status codes that allow software to determine the next action needed.
I2CDAT	**I2C Data Register**. During master or slave transmit mode, data to be transmitted is written to this register. During master or slave receive mode, data that has been received may be read from this register.
I2SCLH	**SCH Duty Cycle Register High Half Word**. Determines the high time of the I2C clock.
I2SCLL	**SCL Duty Cycle Register Low Half Word**. Determines the low time of the I2C clock. I2nSCLL and I2nSCLHtogether determine the clock frequency generated by anI2C master.

Circuit Schematic



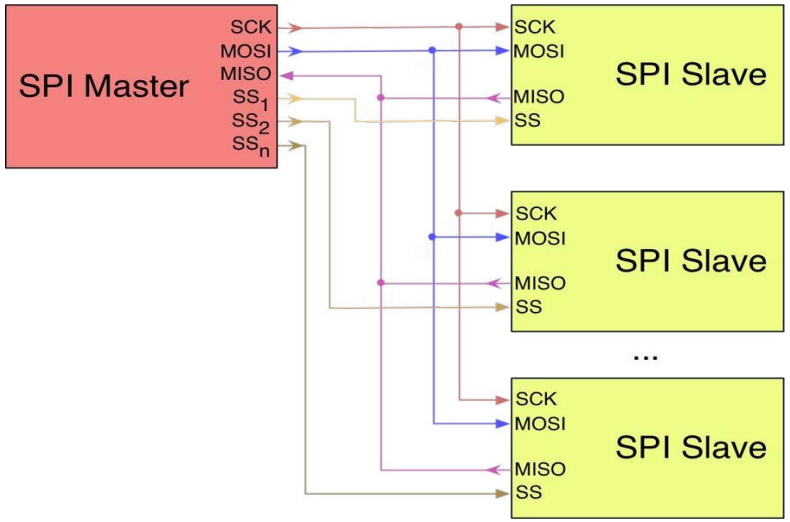
SPI

Introduction to the SPI Specification

- The Serial Peripheral Interface (SPI) is a specification about serial, synchronous and full-duplex communications between a master controller (which is usually implemented with an MCU or something with programmable functionalities) and several slave devices.
- The nature of the SPI interface allows full duplex as well as half duplex communications over the same bus.
- SPI specification is a de facto standard, and it was defined by Motorola[1] in late '70, and it is still largely adopted as communication protocol for many digital ICs.

SPI

- The structure of a typical SPI bus



SPI

A typical SPI bus is formed by four signals, even if it is possible to drive some SPI devices with just three I/Os (in this case we talk about 3-wire SPI):

- **SCK:** this signal I/O is used to generate the clock to synchronize data transfer over the SPI bus.
- It is generated by the master device, and this means that in an SPI bus every transfer is always started by the master.
- Different from the I²C specification, the SPI is intrinsically faster and the SPI clock speed is usually several MHz.
- Nowadays is quite common to find SPI devices able to exchange data at a rate up to 100MHz.
- SPI protocol allows to devices with different communication speeds to coexist over the same bus.

SPI

- **MOSI:** the name of this signal I/O stands for Master Output Slave Input, and it is used to send data from the master device to a slave .
- Different from the I²C bus, where just one wire is used to exchange data both the ways, the SPI protocol defines two distinct lines to exchange data between master and slaves.
- **MISO:** it stands for Master Input Slave Output and it corresponds to the I/O line used to send data from a slave device to the master.
- **SSn:** it stands for Slave Select and in a typical SPI bus there exist 'n' separated lines used to address the specific SPI devices involved in a transaction.

SPI

- Different from the I²C protocol, the SPI does not use slave addresses to select devices, but it demands this operation to a physical line that is asserted LOW to perform a selection.
- In a typical SPI bus only one slave device can be active at same time by asserting low its SS line.
- This is the reason why devices with different communication speed can coexist on the same bus.

SPI

Clock Polarity and Phase

- In addition to setting the bus clock frequency, the master and slaves must also agree on the clock polarity and phase with respect to the data exchanged over MOSI and MISO lines.
- SPI Specification by Motorola names these two settings as CPOL (Clock polarity) and CPHA (clock phase) respectively, and most silicon vendors have adopted that convention.
- The combinations of polarity and phase are often referred to as SPI bus modes which are commonly numbered according Table.
- The most common mode are mode 0 and mode 3, but the majority of slave devices support at least a couple of bus modes

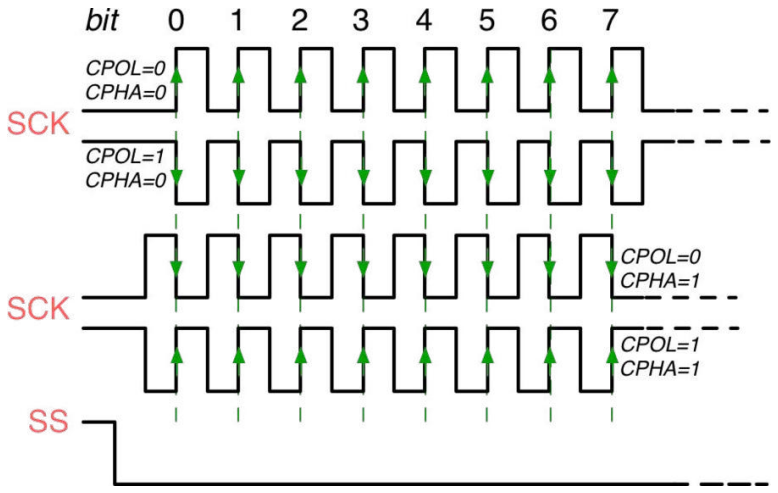
SPI

- SPI bus modes according CPOL and CPHA configuration

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

SPI

- The timing diagram is shown in Figure, and it is further described below:



SPI

- At CPOL=0 the base value of the clock is zero, i.e. the active state is 1 and idle state is 0.
  - For CPHA=0, data is captured on the SCK rising edge (LOW → HIGH transition) and data is output on a falling edge (HIGH → LOW clock transition).
  - For CPHA=1, data is captured on the SCK falling edge and data is output on a rising edge.
- At CPOL=1 the base value of the clock is one (inversion of CPOL=0), i.e. the active state is 0 and idle state is 1.
  - For CPHA=0, data is captured on SCK falling edge and data is output on a rising edge.
  - For CPHA=1, data is captured on SCK rising edge and data is output on a falling edge.

SPI

- That is, CPHA=0 means sampling on the first clock edge, while CPHA=1 means sampling on the second clock edge, regardless of whether that clock edge is rising or falling.
- Note that with CPHA=0, the data must be stable for a half cycle before the first clock cycle.

SPI bus features

1. SPI bus provides bidirectional, synchronous serial communication between uC and peripherals.
2. It is based upon a master-slave protocol where master is the device that drives the clock signal.
3. SPI is capable of up to 3 Mbps (megabits-per-second) data transfer rate.
4. SPI has a hold facility that allows the transmitter to suspend data transfer.
6. Data in SPI can be transferred as multiple bytes known as blocks or pages.
7. The SPI clock is symmetrical (an equal low and high time).

SPI bus features

8. Developed by Motorola and supported by various uC products.
9. In SPI mode, the serial 8-bit synchronous data transmission and reception is possible simultaneously.
10. Multiple masters and multiple slaves are allowed on the bus.
11. Master initiates the data frame. Master generates the clock (SCK) and selects the slave device and then data transfer in both directions takes place.

Bus Configuration and SPI Protocol of Multiple slaves

Multiple slave devices can be connected in parallel or daisy chained utilizing the same SPI bus.

Parallel Configuration

For the parallel connection, each device on the bus should have a separate CS line, while SCK, SDI and SDO lines are connected in parallel.

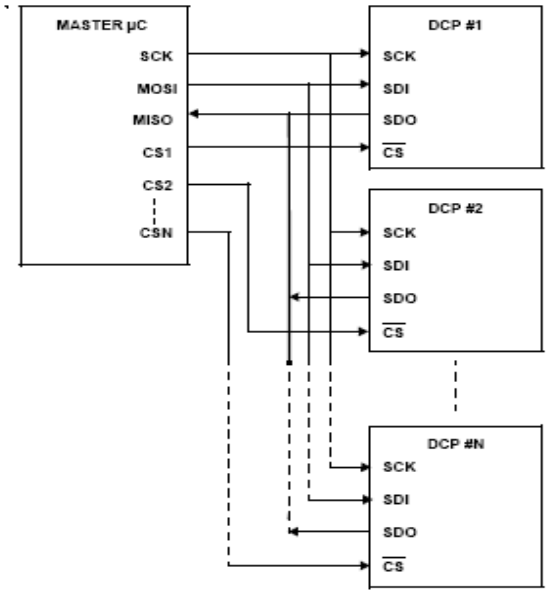
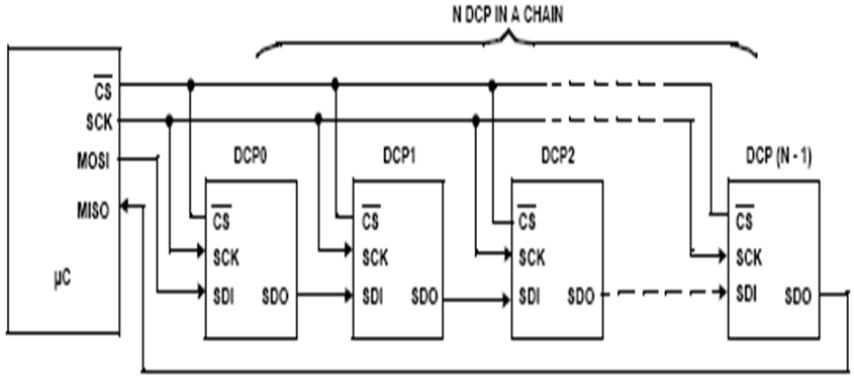


FIGURE 3. PARALLEL CONFIGURATION OF MULTIPLE DCPs

Daisy Chain Configuration

- In this configuration CS and SCK lines connected in parallel, and each SDO pin of previous chip is connected to SDI pin.
- Daisy Chaining simplifies the connection by reducing the length and connections of the data lines, but restricts access to a single device in chain.



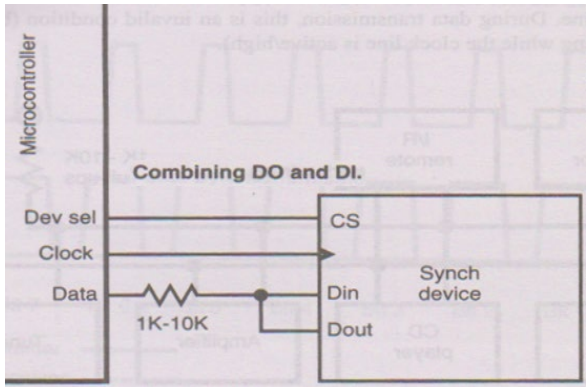
DAISY CHAIN CONFIGURATION OF MULTIPLE DCPs

Daisy Chain Configuration

- In other words, all the devices in chain will be involved in a write or read operation.
- We can consider that every device in the chain is a portion of one big shift register, where serial data is shifted out on each clock going through all the DCPs, from DCP0 to DCP1 and all the way down to the last DCP(N-1) in this chain.

When wiring up a SPI device, there is one trick that you can do to simplify the microcontroller connection is to combine the Dl and DO lines into one pin Fig.

In this method of connecting the two devices, when the Data pin on the microcontroller has completed sending the serial data, the output driver can be turned off and the microcontroller can read the data coming from the device.



- The current limiting resistor between the Data pin and DI/DO limits any current flows when both the microcontroller and device are driving the line.
- This trick can be used in a variety of different situations, but it must be done so carefully.
- The resistor will prevent any possible I/O pin damage due to bus contention (two devices driving the same line to different levels at the same time).
- However, if the resistor is not properly wired into the circuit or if the MCU attempts to read while it is driving, then invalid data will be stored or read.

innovate achieve lead

THANK YOU!!!!!

Embedded System Design

BITS Pilani
Pilani|Dubai|Goa|Hydarabad

Prof. Manoj S Kakade
Dept of EEE

BITS Pilani
Pilani|Dubai|Goa|Hydarabad

ESZG512/MELZG526/SEZG516
Embedded System Design
Contact Session 15

innovate achieve lead

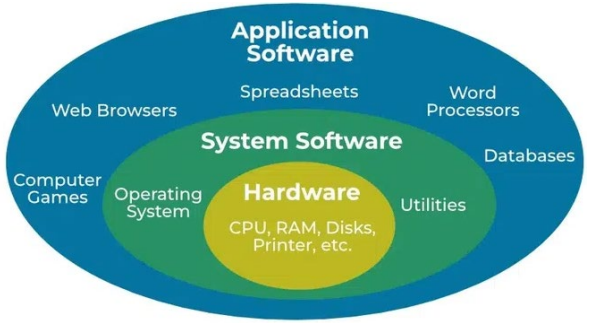
CS15 : (Embedded Software Design)

- Assembly
- Compiling
- Linking
- Foreground and background systems
- What is an Operating System?
- Process or Task
- Task States
- Threads

innovate achieve lead

Computer Systems

- Hardware
- Firmware
- Software
  - Application software
  - System software
    - Programming systems
    - Utilities
    - Operating systems



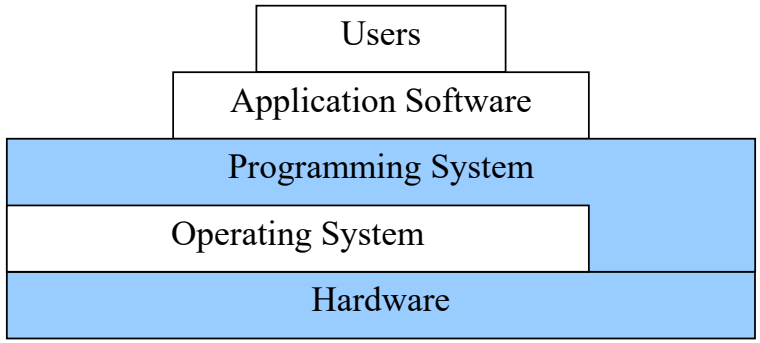
- Firmware's are generally a type of software used to control hardware devices.
- Software (application software) runs on top of the operating system and has no direct interface with hardware. Firmware is low-level software that stands between the hardware and the operating system.

Computer Software

- Application software is designed to solve a specific problem.
- System software is intended to support the operation and use of the computer itself.
  - Provide a general programming environment.
  - Provide functions used by application software and users.
  - Provide mechanisms to share the hardware in an orderly fashion.
- Utility software is software designed to help analyze, configure, optimize or maintain a computer. It is used to support the computer infrastructure - in contrast to application software, which is aimed at directly performing tasks that benefit ordinary users. Eg : Antivirus Software, File Management Tool, Compression Tool, Disk Management Tool, Disk Cleanup Tool etc..

Foundations of Computer Systems

Foundations of Computer Systems



Evolution of Programming

Machine Language

- Binary format

  11100101100111110001000000010000
  11100101100111110000000000001000
  11100000100000010101000000000000
  11100101100011110101000000001000

- Hexadecimal format

  E59F1010
  E59F0008
  E0815000
  E58F5008

Evolution of Programming

Assembly Language

- Mnemonic codes

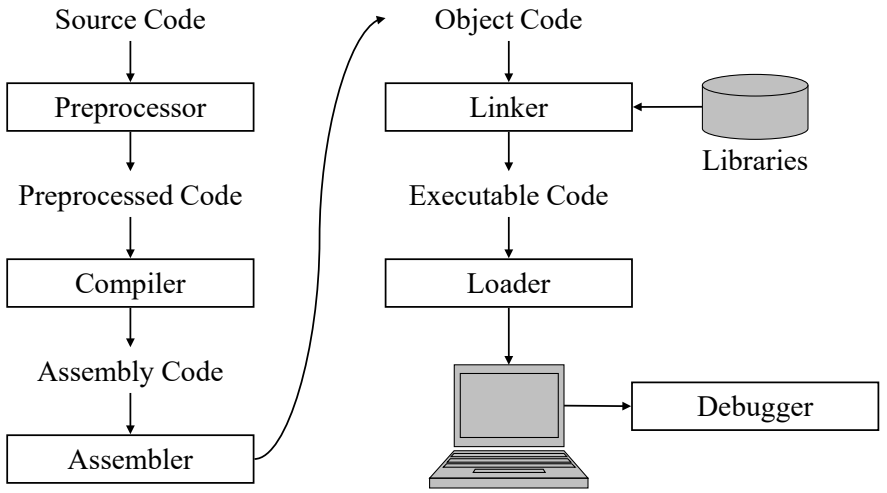
E59F1010	**LDR**	**R1, num1**
E59F0008	**LDR**	**R0, num2**
E0815000	**ADD**	**R5, R1, R0**
E58F5008	**STR**	**R5, sum**

High-Level Language

- C language

  sum = num1 + num2;

From Source to Executable



EMBEDDED SOFTWARE DEVELOPMENT TOOLS

- Introduction
  - Application programs are typically developed, compiled, and run on host system
  - Embedded programs are targeted to a target processor (different from the development/host processor and operating environment) that drives a device or controls
  - What tools are needed to develop, test, and locate embedded software into the target processor and its operating environment?

  - Distinction
  - Host: Where the embedded software is developed, compiled, tested, debugged, optimized, and prior to its translation into target device. (Because the host has keyboards, editors, monitors, printers, more memory, etc. for development, while the target may have not of these capabilities for developing the software.)
  - Target: After development, the code is cross-compiled, translated – cross- assembled, linked (into target processor instruction set) and ***located*** into the target

Development Phases of an Embedded System

Phase 1:Analysis

(Requirements & specifications)

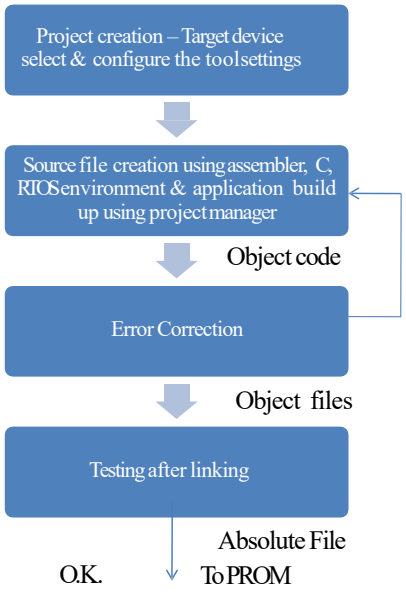
Phase 2: Design

(Selection of development tools)

Phase 3: Implementation

Phase 4: Testing and Debugging

Software Development Cycle



Integrated Development Environment (IDE)

- An IDE gives a single focal point for the application development for
  1. Creating source file through a project manager.
  2. Rich featured source code.
  3. Organization of source file into a project.
  4. Provision of database for many devices.
  5. a)Compiles, points errors and interactively correct the errors.
     b)Assembles and links to application.
     c)Links a device data sheets, user guides & development tool manuals.

Integrated Development Environment (IDE)

- **Assembler**
- It optimally exploits the chosen microcontroller chips special features. Its instructions gives direct control of stack, I/O and registers etc.
- **Compiler**
- Creates a object code file
- e.g. C51 compiler from Keil
- **Dissembler**
- It translates the object codes into the mnemonics form of assembly language
- **Library and Library Manager**
- An ordered & specially formatted program collection of object codes so that a linker can use it.
- **Cross assembler and Cross Compiler**
- Cross compiler and Cross assembler convert the object code file for a CPU, is converted to the object code file for another processor & vice versa.

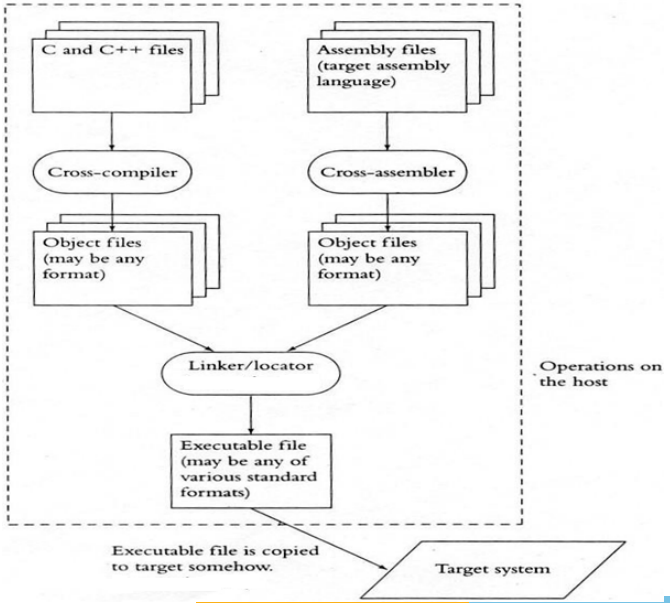
Integrated Development Environment (IDE)

- **Linker**
  - Linker creates an absolute object module file from other library modules.
  - Object to Hex file converter gives hex file for the use of device programmer.
- **Debugger**
  - Provides many functionalities like start-stop debugging, single-step tracing etc.
- **Simulator**
  - It simulates hardware performance for software to be embedded in that without actually employing a specific device. It simulates all functions of a microcontroller circuit, including interfaced additional memory and peripherals.

Integrated Development Environment (IDE)

- **Cross-Compilers**
- Native tools are good for the host, but to port/locate embedded code to target, the host must have a tool-chain that includes a cross-compiler, one which runs on the host but produces code for the target processor
- **Cross-Assemblers and Tool Chain**
- Host uses cross-assembler to assemble code in target's instruction syntax for the target
- Tool chain is a collection of compatible, translation tools, which are 'pipelined' to produce a complete binary/machine code that can be linked and located into the target processor

Tool Chain for Building Embedded Software



EMBEDDED SOFTWARE DEVELOPMENT TOOLS

- Linker/Locators for Embedded Software
- Native linkers are different from cross-linkers (or locators) that perform additional tasks to locate embedded binary code into target processors.
- Address Resolution –
- Native Linker: produces host machine code on the hard-drive (in a named file), which the loader loads into RAM, and then schedules (under the OS control) the program to go to the CPU.
- In RAM, the application program/code's logical addresses for, e.g., variable/operands and function calls, are ordered or organized by the linker. The locator then maps the logical addresses into physical addresses – a process called address resolution. The locator then loads the code accordingly into RAM . In the process the locator also resolves the addresses for calls to the native OS routines
- Locator: produces target machine code (which the locator glues into the RTOS) and the combined code (called map) gets copied into the target ROM. The locator doesn't stay in the target environment, hence all addresses are resolved, guided by locating-tools and directives, prior to running the code

Foreground and Background Systems

- The foreground/background model for managing task execution decomposes the set of tasks comprising the application into two subsets called background tasks and foreground tasks.
- The traditional view of such systems allocates tasks that interact with the user or other I/O devices to the foreground set and the remainder to the background set.
- The interpretation is slightly modified in the embedded world.

Foreground and Background Systems

- The foreground tasks are those initiated by interrupt or by a real-time constraint that must be met.
- They will be assigned the higher priority levels in the system.
- In contrast, background tasks are noninterrupt driven and are assigned the lower priorities.
- Once started, the background task will typically run to completion; however, it can be interrupted or preempted by any foreground task at any time.
- Often separate ready queues will be maintained for the two types of tasks.

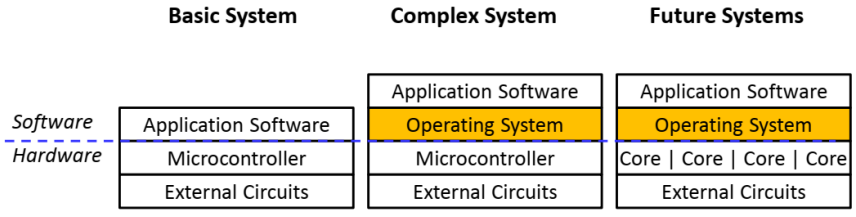
Foreground and Background Systems

- The background tasks should include all those that do not have tight time constraints.
- Tasks that are designed to continuously monitor system integrity or involve heavy processing are good candidates.

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

What is an Operating System?

- A software layer between the application software and the hardware



Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

What is an Operating System?

- Typical embedded system (ES) solves a problem by decomposing it into smaller pieces called tasks that work together in an organized way
- System is called multitasking system and design aspects include:
  - Exchanging/sharing data between tasks
  - Synchronizing tasks
  - Scheduling tasks
- The piece of software that provides the required coordination is called an operating system (OS)
- When the control must ensure that task execution satisfies a set of specified time constraints, the OS is called a real-time operating system (RTOS)

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

What is an Operating System?

- Primary software modules within an operating system that implement such control
  - Scheduler
  - Dispatcher
  - Intertask communication
- Scheduler determines
  - Which task will run
  - When the task will run
- The dispatcher performs the necessary operations to start the task
- Intertask communication mechanism for exchanging data and information between tasks or processes same machine different machines
- **Kernel**- Smallest portion of operating system provides above services

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

What is an Operating System?

- Full Featured Operating System provides additional libraries of functions, device drivers, rich communication packages, human computer interface.

Real Time Operating System - RTOS

- Real time operating system is a special purpose operating system implies rigid time requirements must be met. If requirements are not met results system functionality is inaccurate and compromised.
- Such systems usually interact with the physical environment: sensors, measurement devices.

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

What is an Operating System?

Come in two flavors

Hard Real Time

- System delays are known or at least bounded said to be operating correctly if can return results within any time constraints

Soft Real Time

- Critical tasks get priority over other tasks retain priority until complete

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Process or Task

- Embedded program (a static entity) = a collection of firmware modules
- When a firmware module is executing, it is called a process or task
- A task is usually implemented in C by writing a function
- A task or process simply identifies a job that is to be done within an embedded application
- When a process is created, it is allocated a number of resources by the OS, which may include:
  - Process stack
  - Memory address space
  - Registers (through the CPU)
  - A program counter (PC)
  - I/O ports, network connections, file descriptors, etc.
- These resources are generally not shared with other processes

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

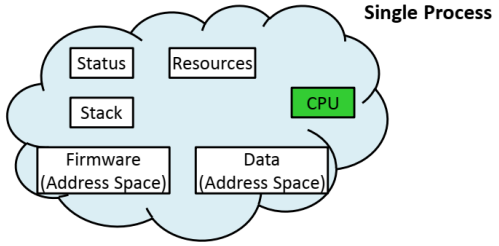
Types of Tasks

- Periodic tasks
  - Found in hard real-time applications
  - Examples: control, every 10 ms; multimedia, every 22.727us;
- Intermittent tasks
  - Found in all types of applications
  - Examples: send email every night at 4am; calibrate a sensor on startup; save all data when power goes down;
- Background tasks
  - A soft real-time or non real-time task
  - Will be accomplished only if CPU time is available
- Complex tasks
  - Found in all types of applications
  - Examples: Microsoft Word; Apache web server;

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Single Process

- Traditional view of computing: focuses on program. One says that the program (or task within the program) runs on the computer
- In embedded applications, we change the point of view to that of microprocessor: CPU is used to execute the firmware. CPU is just another resource
- The time it takes a task to complete is called execution time



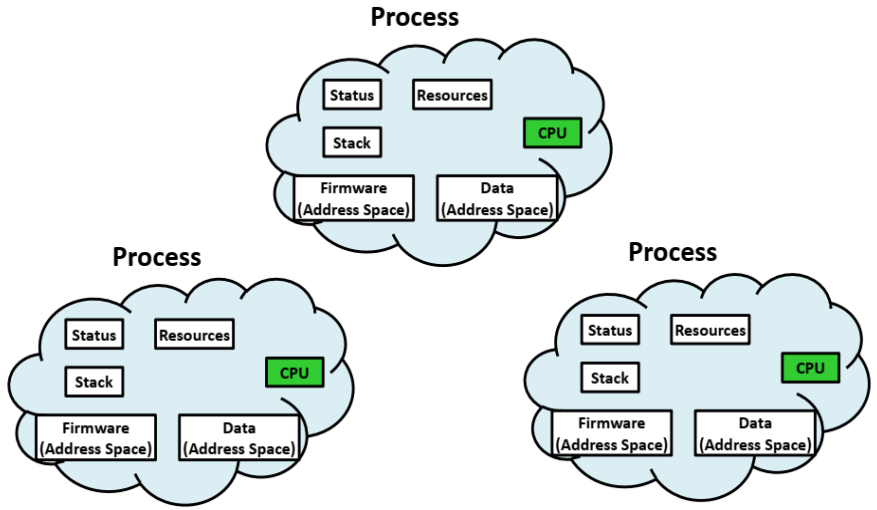
Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Multiple Processes

- If another task is added to the system, potential resource contention problems arise
- This is resolved by carefully managing how the resources are allocated to each task and by controlling how long each can retain the resources
- The main resource, CPU, is given to tasks in a time multiplexed fashion (i.e., time sharing); when done fast enough, it will appear as if both tasks are using it at the same time
- The execution time of the program will be extended, but operation will give the appearance of simultaneous execution. Such a scheme is called multitasking

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Multiple Processes

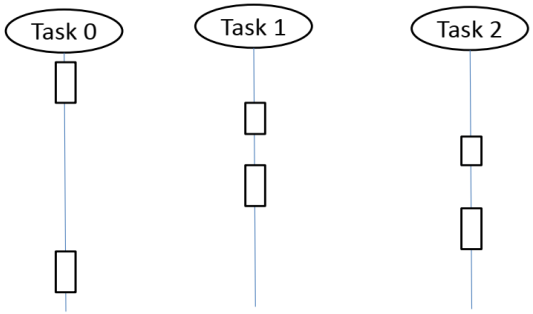


Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Multiple Processes

Sequence Diagram

- At any instant in time, only one process is actively executing; it said to be in run state
- The other processes are in ready or in waiting state



Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Task Scheduling

- A schedule is set up to specify when, under what conditions, and for how long each task will be given the use of the CPU (and other resources)
- The criteria for deciding which task is to run next are collectively called a scheduling strategy, which generally falls into three categories:
- **Multiprogramming**
  - each task continues until it performs an operation that requires waiting for an external event
- **Real-Time**
  - tasks with specified temporal deadlines are guaranteed to complete before those deadlines expire
- **Time sharing**
  - running task is required to give up the CPU so that another task may get a turn

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Task States

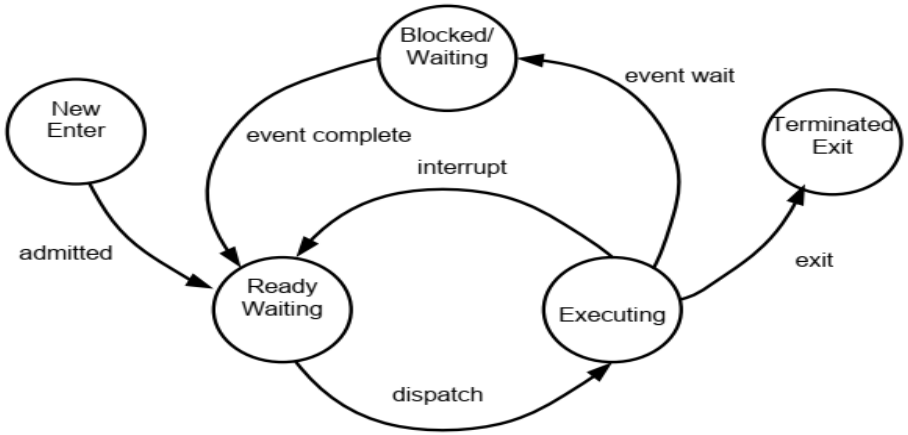
- Primarily 4 states

1. Running or Executing
2. Ready to Run (but not running)
3. Waiting (for something other than the CPU)
4. Inactive

- Only one task can be Running at a time, unless we use a multicore CPU
- Task waiting for CPU is Ready to Run
- When a task has requested I/O or put itself to sleep, it is Waiting
- An Inactive task is waiting to be allowed into the schedule

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Task States



Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Process

- Process is a program in execution, comprises active processes
- as process executes it's often changing state
- Transition between states is referred to as context switch
- State must be saved / restored to switch between tasks
  - Program Counter (PC)
  - Register values
  - Process or status flags (Status Register)
  - Stack Pointer (SP)
  - Memory state
  - Peripheral configurations
  - etc

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Context switch

- Task's context comprises important information about the state of the task, Values of any variables held in the CPU's registers, Value of the program counter, State of the stack etc.
- Each time running task is stopped – preempted or blocked CPU is given to another task that is ready switch to a new context is executed
- Context switch first requires state of the currently active task be saved

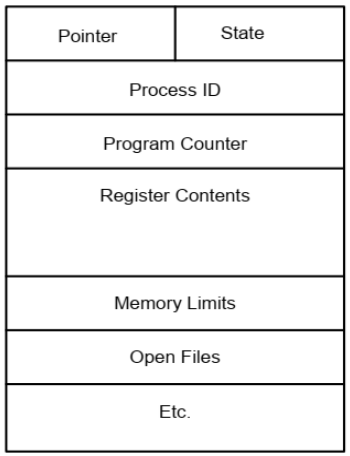
Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Task / Process Control Block

Task / Process Control Block

- In tasked based approach each process represented in the operating system by a data structure called Task Control Block – TCB also known as a process control block



Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Task / Process Control Block

TCB contains all of the important information about the task

- A typical TCB contains following information
- Pointer (for linking the TCB to various queues)
- Process ID and state
- Program counter
- CPU registers
- Scheduling information (priorities and pointers to scheduling queues)
- Memory management information (tag tables and cache information)
- Scheduling information (time limits or time and resources used)
- I/O status information (resources allocated or open files)

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

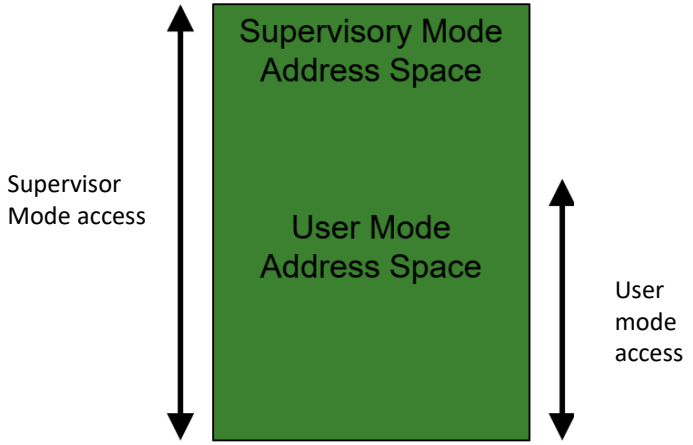
Address Space of a Process

Address Space of a Process

- When a process is created by the OS, it is given a portion of the physical memory in which to work
- The set of addresses delimiting that code and the data memory, proprietary to each process, is called its address space
- Processes are segregated
  - Supervisor mode
  - User mode – limited to a subset of instructions
- A process may create or spawn child processes (each with its own data address space, data, status, and stack)
- A process may create multiple threads (each with its own stack and status information)

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Address Space of a Process



Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

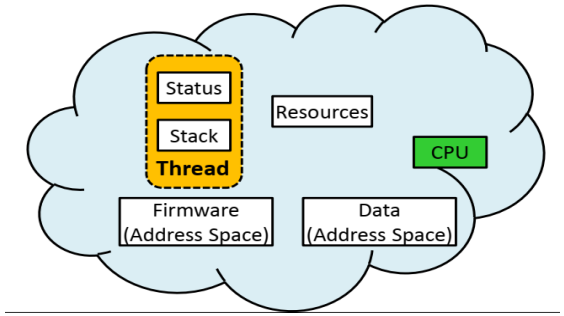
Threads

- A process or task is characterized by a collection of resources that are utilized to execute a program
- The smallest subset of these resources (a copy of the CPU registers including the PC and a stack) that is necessary for the execution of the program is called a thread
- A thread is a unit of computation with code and context, but no private data
- A thread can be in only one process; a process without a thread can do nothing!

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Single-process single-thread

- The sequential execution of a set of instructions through a task or process in an embedded application is called a thread of execution or thread of control
- This model is referred as single-process single-thread



Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

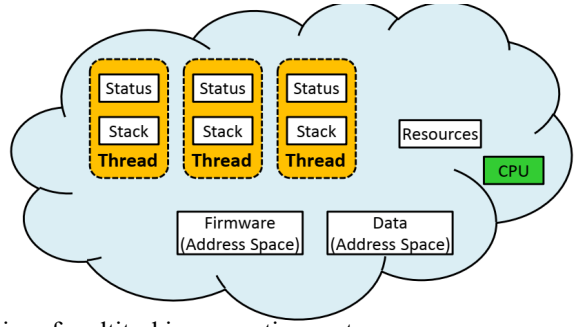
Multiple Threads

- During partitioning and functional decomposition of the function intended to be performed → by an ES identify which actions would benefit from parallel execution
  - For example, allocate a sub job for each type of I/O
- Each of the sub jobs has its own thread of execution
  - Such a system is called a single-process multithread design
- Threads are not independent of each other (unlike processes or tasks)
  - Threads can access any address within the process, including other threads' stacks
- An OS that supports tasks with multiple threads is called a multithreaded operating system

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Single-Process Multiple-Threads

Single-Process Multiple-Threads



All four categories of multitasking operating system:
– Single process single thread
– Multiprocess single thread
– Single process multiple threads
– Multiprocess multiple threads

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Processes (tasks) vs. Threads

Processes (tasks) vs. Threads

- At the minimum, a process or task needs the following:

1. The code or firmware, the instructions
   –These are in the memory and have addresses
2. The data that the code is manipulating
   –The data starts in the memory and may be moved to registers. the data has addresses

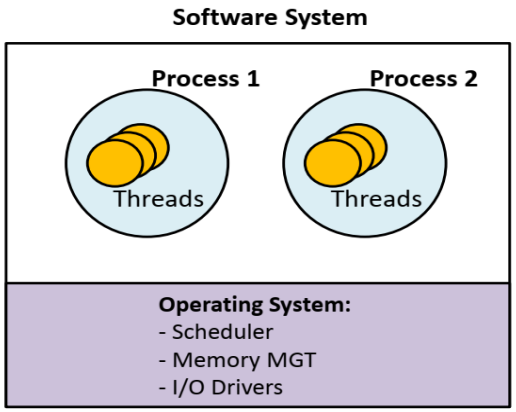
Shared among member Threads

3. CPU and associated physical registers
4. A stack
5. Status information

Proprietary to each Thread

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Complete software system with two processes



Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

- Context switch between threads can be substantially simpler and faster than between processes
- When switching between threads much less information must be saved and restored
- An operating system that supports tasks with multiple threads referred to as a multithreaded operating system
- Can easily extend design to support multiple processes can further decompose each process into multiple subtasks such a system called multiprocess – multithread design

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Reentrant Code

Reentrant Code

- Child processes (and their threads) share the same firmware memory area → two different threads can execute the same function
- Functions using only local variables are inherently reentrant
- Functions using global variables, variables local to the process, variables passed by reference, or shared resources are not reentrant
- Any shared functions must be designed to be reentrant

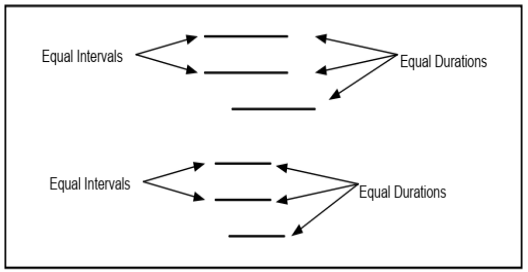
Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

- In world of embedded systems can control operation of systems in number of different ways
- Can loosely classify such systems into two broad categories
- **Time based**
- **Reactive**

- **Time Based Systems**
- Systems whose behaviour controlled by time Can be
  - **Absolute**
  - **Relative**
  - **Following an interval**

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

- Absolute time
  - Real world time
- Duration
  - Relative time measure
  - Non-equal intervals
- Interval
  - Distinct from duration
  - Interval marked by specific start and end times
  - Equal intervals have same start and stop can have same duration



Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Tasks and Intertask Communication

- Multitasking / Multithreading system supports multiple tasks
- Important job in multitasking system

✓ Exchanging data between tasks

✓ Synchronizing tasks

✓ Sharing resources

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

Interprocess / Interthread Communication

- When threads operating independently our systems have few if any Conflicts, Chances for corruption, Contentions
- Real systems must deal with all such problems resources and inter thread communication must take place in robust manner
- Interaction may be direct or indirect Must be synchronized and co-ordinated want to prevent race conditions outcome of task or computation depends upon order in which tasks execute

Source: James. K. Peckol, "Embedded System Design – A Contemporary Design Tool"

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

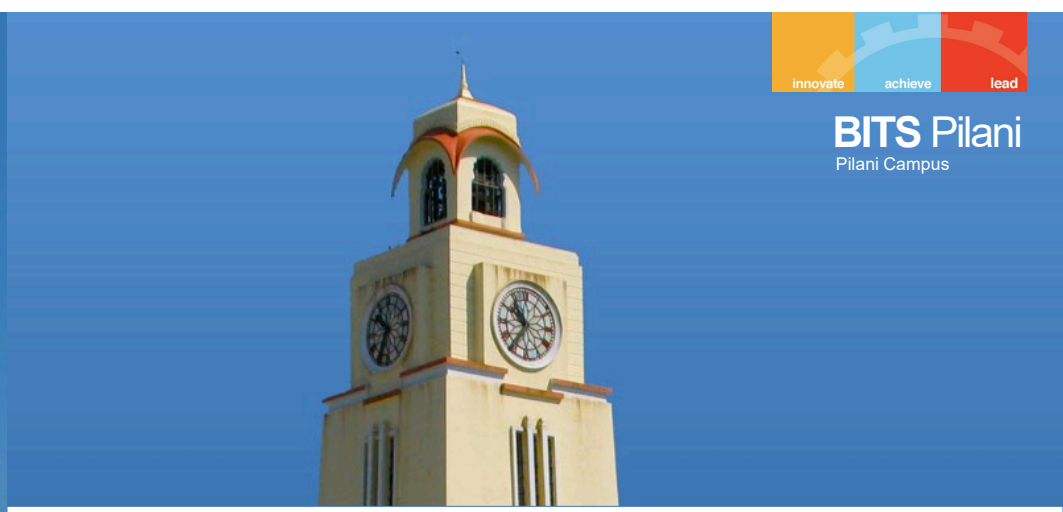
Thank you

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Embedded System Design

BITS Pilani
Pilani Campus

Prof. Manoj S Kakade
Dept of EEE



EEEZG512/ESZG512/MELZG526/SEZG516
Embedded System Design
Contact Session 16

Contents

Cortex Microcontroller Software Interface Standard-Real-Time Operating System (CMSIS-RTOS)-RTX

Reference Book:

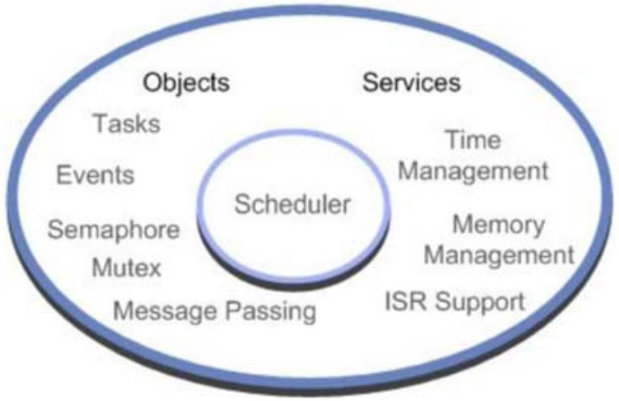
- Trevor Martin,"The Designer's Guide to the Cortex-M Processor Family", Second Edition, Elsevier Ltd.

Introduction

- "Cortex Microcontroller Software Interface Standard (CMSIS)" defines a standard API for Cortex-M RTOS.
- RTOS structure allows you to take a more object-orientated design approach
- RTOS also provides you with multithreaded support on a small microcontroller.
- RTOS-based project is composed of well-defined threads it helps to improve project management, code reuse, and software testing.
- RTOS has additional memory requirements and increased interrupt latency.
- Keil RTX RTOS: 500 bytes of RAM and 5k bytes of code.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

RTOS kernel



RTOS kernel

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

CMSIS-RTOS API

Accessing the CMSIS-RTOS API:

- To access any of the CMSIS-RTOS features in our application code it is necessary to include the following header file:

  #include<cmsis_os.h>

- This header file is maintained by ARM as part of the CMSIS-RTOS standard.
- Keil RTX RTOS this is the default API.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Threads

- In CMSIS-RTOS the basic unit of execution is a "thread."
- A thread is very similar to a "C" procedure but has some very fundamental differences.

```
unsigned int procedure (void)
{

  ……

  return(ch);
}
```

```
void thread (void)
{
    while(1)
    {
     ……
    }
}
```

- While we always return from our "C" function, once started an RTOS thread must contain a loop so that it never terminates and thus runs forever.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Threads

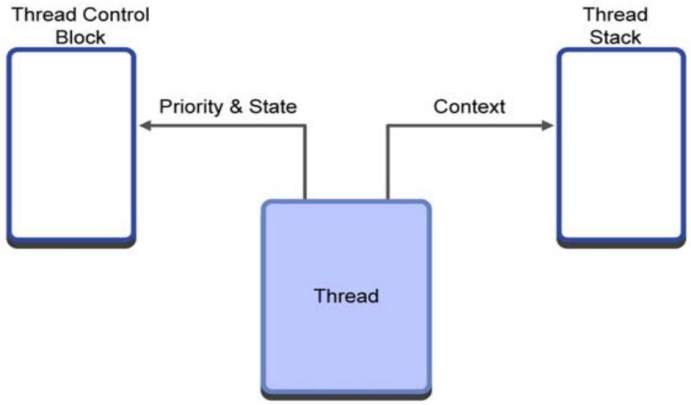
- When a thread is created, it is also allocated its own thread ID.
- This is a variable which acts as a handle for each thread and is used when we want to manage the activity of the thread.

  osThreadId id1,id2,id3;

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Threads

Context switch:



Each thread has its own stack for saving its data during a context switch.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Threads

Threads may be in one of three states:

- **Running** -The currently running thread
- **Ready** -Threads ready to run
- **Wait** -Blocked threads waiting for an OS event

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

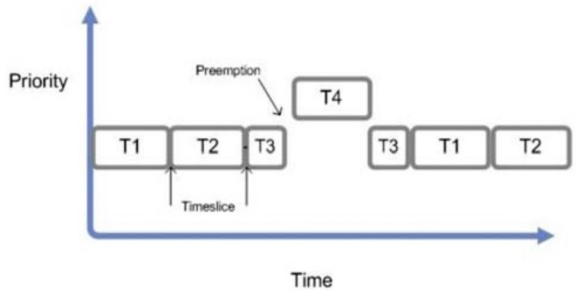
Starting the RTOS

```
void thread1 (void);
void thread2 (void);
osThreadId thrdID1, thrdID2;
void main (void)
{
osKernelInitialize ();
IODIR1 5 0x00FF0000; // Do any C code you want
Init_Thread(); //Create a Thread
osKernelStart(); //Start the RTOS
}
```

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Threads

- Threads of equal priority will be scheduled in a round-robin fashion.
- High priority tasks will preempt low priority tasks and enter the running state "on demand."



Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

- A First CMSIS-RTOS Project – Demo in Keil

Creating Threads

- By default, the main() function is automatically created as the first thread running.
- We want to control main as a thread we must get its thread ID.
- The first RTOS function we must therefore call is osThreadGetId() which returns the thread ID number of the currently running thread.
- This is then stored in its ID handle.

Creating Threads

- First a thread structure is defined; this allows us to define the thread operating parameters.

```
osThreadId thread1_id; //thread handle
void thread1 (void const *argument);
//function prototype for thread1
osThreadDef(thread1, osPriorityNormal, 1, 0);
//thread definition structure
```

- Once the thread structure has been defined the thread can be created using the osThreadCreate() API call.
- Then the thread is created from within the application code; this is often done within the main thread but can be at any point in the code.

```
thread1_id 5 osThreadCreate(osThread(thread1), NULL);
```


Creating Threads

- When each thread is created, it is also assigned its own stack for storing data during the context switch.
- This should not be confused with the native Cortex processor stack; it is really a block of memory that is allocated to the thread.

Example Demo in Keil

- Creating and Managing Threads: We will create and manage some additional threads. Each of the threads created will toggle a GPIO pin on GPIO port B to simulate flashing an LED.

Thread Management and Priority

CMSIS-RTOS priority levels:

osPriorityIdle

osPriorityLow

osPriorityBelowNormal

osPriorityNormal

osPriorityAboveNormal

osPriorityHigh

osPriorityRealTime

osPriorityError

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Thread Management and Priority

- Once the threads are running, there are a small number of OS system calls which are used to manage the running threads.
- It is also then possible to elevate or lower a thread's priority either from another function or from within its own code.

```
osStatus osThreadSetPriority(threadID, priority);
osPriority osThreadGetPriority(threadID);
```

Example:

We will look at assigning different priorities to threads

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Multiple Instances

- One of the interesting possibilities of an RTOS is that you can create multiple running instances of the same base thread code.
- First we create the thread structure and set the number of thread instances to two:

```
osThreadDef(thread1, osPriorityNormal, 2, 0);
```

- Then we can create two instances of the thread assigned to different thread handles. A parameter is also passed to allow each instance to identify which UART it is responsible for.

```
ThreadID_1_0 = osThreadCreate(osThread(thread1), UART1);
ThreadID_1_1 = osThreadCreate(osThread(thread1), UART2);
```

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Example:

We will look at creating one thread and then create multiple run-time instances of the same thread.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Time Management

- RTOS also provides some timing services which can be accessed through RTOS system calls.

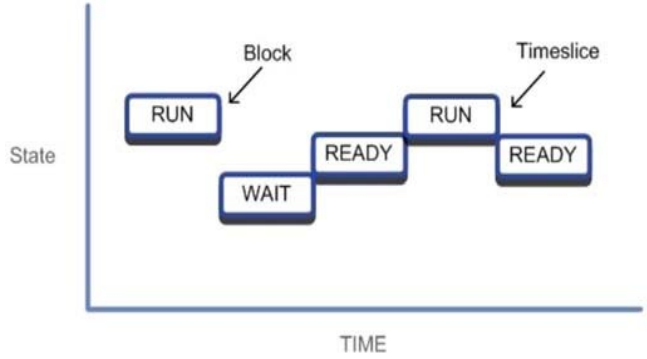
Time Delay

- The most basic of these timing services is a simple timer delay function.

  void osDelay (uint32_t millisec)

- This call will place the calling thread into the WAIT_DELAY state for the specified number of milliseconds.
- The scheduler will pass execution to the next thread in the READY state
- When the timer expires, the thread will leave the wait_delay state and move to the READY state.
- The thread will resume running when the scheduler moves it to the RUNNING state.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"



During their lifetime threads move through many states.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Waiting for an Event

- In addition to a pure time delay it is possible to make a thread halt and enter the waiting state until the thread is triggered by another RTOS event.
- RTOS events can be a signal, message, or mail event.
- The osWait() API call also has a timeout period defined in milliseconds that allows the thread to wake up and continue execution if no event occurs.

  osStatus osWait (uint32_t millisec)

- When the interval expires, the thread moves from the wait to the READY state and will be placed into the running state by the scheduler.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Example:

We will look at using the basic time delay function.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Virtual Timers

- The CMSIS-RTOS API can be used to define any number of virtual timers which act as count down timers.
- When they expire, they will run a user call-back function to perform a specific action.
- Each timer can be configured as a one shot or repeat timer.
- A virtual timer is created by first defining a timer structure.

```
osTimerDef(timer0,led_function);
```

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

- This defines a name for the timer and the name of the call back function. The timer must then be instantiated in an RTOS thread.

```
osTimerId timer0_handle 5 osTimerCreate (timer(timer0), osTimerPeriodic, (void *)0);
```

- This creates the timer and defines it as a periodic timer or a single shot timer (osTimerOnce). The final parameter passes an argument to the call back function when the timer expires.

```
osTimerStart ( timer0_handle,0x100);
```

- The timer can then be started at any point in a thread the timer start function invokes the timer by its handle and defines a count period in milliseconds.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Example:

We will configure a number of virtual timers to trigger a callback function at various frequencies.

Idle Demon

- If during our RTOS program we have no thread running and no thread ready to run (eg, they are all waiting on delay functions) then the RTOS will use the spare run-time to call an "Idle Demon" that is again located in the RTX_Conf_CM.c file.
- This idle code is in effect a low priority thread within the RTOS which only runs when nothing else is ready.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

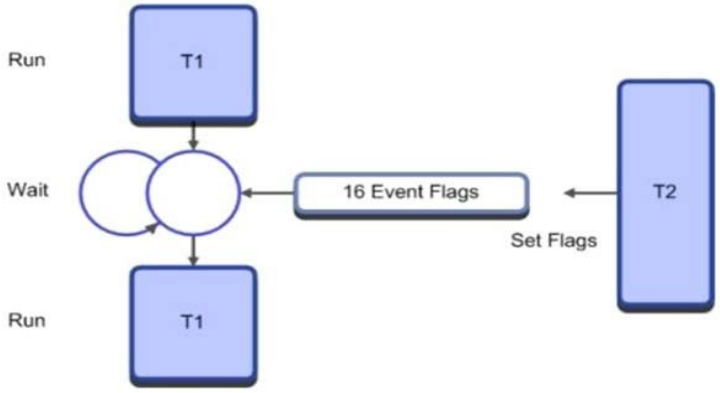
Inter-Thread Communication

- The CMSIS-RTOS API supports inter-thread communication with signals, semaphores, mutexes, mailboxes, and message queues.

Signals

- CMSIS-RTOS Keil RTX supports up to 16 signal flags for each thread.
- These signals are stored in the thread control block.
- It is possible to halt the execution of a thread until a particular signal flag or group of signal flags are set by another thread in the system

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"



A thread may be placed into a waiting state until a pattern of flags is set by another thread. When this happens, it will return to the ready state and wait to be scheduled by the kernel

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

- The signal wait system calls will suspend execution of the thread and place it into the wait_event state.
- Execution of the thread will not start until all the flags set in the signal wait API call have been set.
- It is also possible to define a periodic timeout after which the waiting thread will move back to the ready state, so that it can resume execution when selected by the scheduler.
- A value of 0xFFFF defines an infinite timeout period.

  osEvent osSignalWait (int32_t signals,uint32_t millisec);
- If the signals variable is set to zero when osSignalWait is called then setting any flag will cause the thread to resume execution.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Example:

We will look at using signals to trigger activity between two threads. While this is a simple program it introduces the concept of synchronizing the activity of threads together.

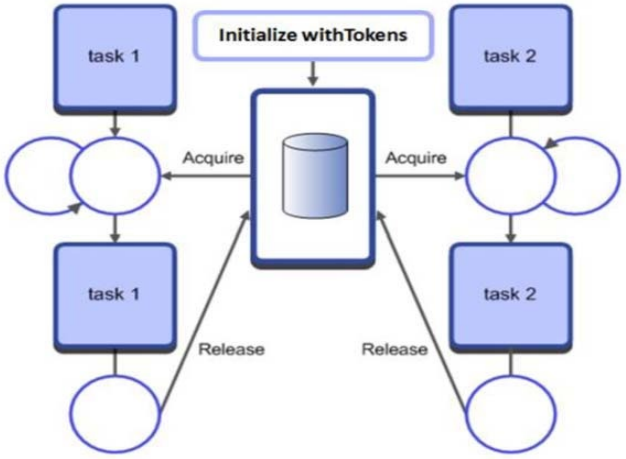
Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Semaphores

- Semaphores are a method of synchronizing activity between two or more threads.
- A semaphore is a container that holds a number of tokens.
- As a thread executes, it will reach an RTOS call to acquire a semaphore token.
- If the semaphore contains one or more tokens, the thread will continue executing and the number of tokens in the semaphore will be decremented by one.
- If there are currently no tokens in the semaphore, the thread will be placed in a waiting state until a token becomes available.
- At any point in its execution, a thread may add a token to the semaphore causing its token count to increment by one

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Semaphores help to control access to program resources.



Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

To use a semaphore in the CMSIS-RTOS you must first declare a semaphore container:

```
osSemaphoreId sem1;
osSemaphoreDef(sem1);
```

Then within a thread the semaphore container can be initialized with a number of tokens.

```
sem1 = osSemaphoreCreate(osSemaphore(sem1), SIX_TOKENS);
```

Example:

We will look at the configuration of a semaphore and use it to signal between two tasks.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Mutex

- Mutex stands for "Mutual Exclusion."
- A mutex is a specialized version of semaphore.
- Like a semaphore, a mutex is a container for tokens.
- A mutex can only contain one token which cannot be created or destroyed.
- The principle use of a mutex is to control access to a chip resource such as a peripheral. For this reason a mutex token is binary and bounded.
- Apart from this it really works in the same way as a semaphore.
- First of all we must declare the mutex container and initialize the mutex:

```
osMutexId uart_mutex;
osMutexDef (uart_mutex);
```

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

- Once declared the mutex must be created in a thread.

```
uart_mutex = osMutexCreate(osMutex(uart_mutex));
```

- Then any thread needing to access the peripheral must first acquire the mutex token:

```
osMutexWait(osMutexId mutex_id,uint32_t millisec;
```

- Finally, when we are finished with the peripheral the mutex must be released:

```
osMutexRelease(osMutexId mutex_id);
```

- Mutex use is much more rigid than semaphore use, but is a much safer mechanism when controlling absolute access to underlying chip registers.

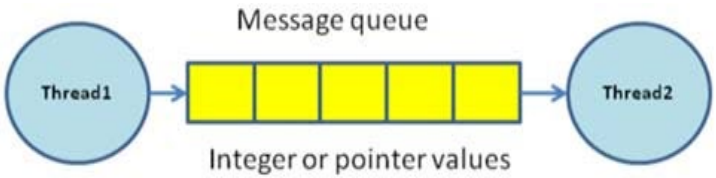
Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Example:

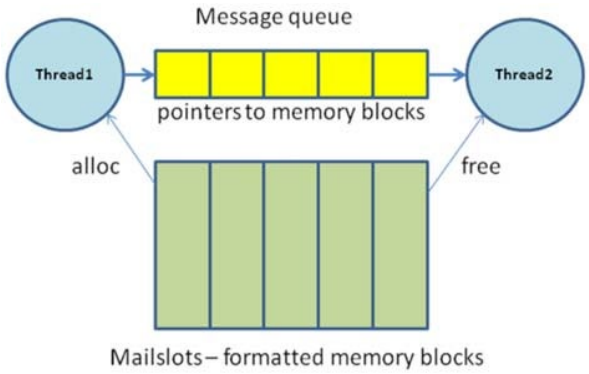
Program writes streams of characters to the microcontroller UART from different threads.

Data Exchange

- CMSIS-RTOS provides two methods of data transfer between threads.
- The first method is a message queue which creates a buffered data "pipe" between two threads.
- The message queue is designed to transfer integer values



- The second form of data transfer is a mail queue.
- This is very similar to a message queue except that it transfers blocks of data rather than a single integer



- Message and mail queues both provide a method for transferring data between threads.
- This allows you to view your design as a collection of objects (threads) interconnected by data flows.
- The data flow is implemented by message and mail queues.
- This provides both a buffered transfer of data and a well-defined communication interface between threads.

Examples:

Message Queue

We will look at defining a message queue between two threads and then use it to send process data.

Mailbox

Demonstrates configuration of a mailbox and using it to post messages between tasks.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

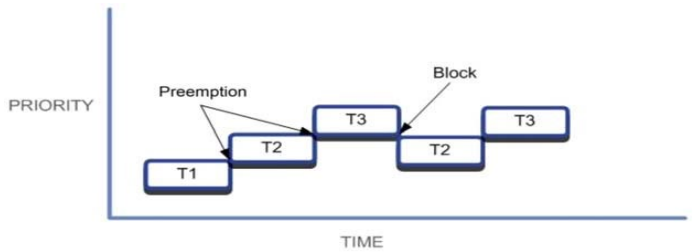
Scheduling Options

- CMSIS-RTOS allows you to build an application with three different kernel scheduling options.
- These are round-robin scheduling, preemptive scheduling, and cooperative multitasking.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Preemptive Scheduling

- If the round-robin option is disabled in the RTX_Config_CM.c file, each thread must be declared with a different priority.
- When the RTOS is started and the threads are created, the thread with the highest priority will run



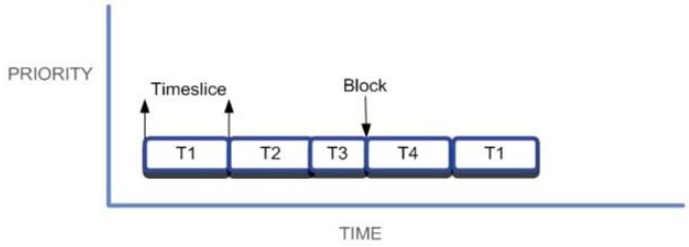
Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

- This thread will run until it blocks, that is, it is forced to wait for an event flag, semaphore, or other object.
- When it blocks, the next ready thread with the highest priority will be scheduled and will run until it blocks, or a higher priority thread becomes ready to run.
- So with preemptive scheduling we build a hierarchy of thread execution, with each thread consuming variable amounts of run-time.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Round-Robin Scheduling

- A round-robin-based scheduling scheme can be created by enabling the round-robin option in the RTX_Conf_CM.c file and declaring each thread with the same priority



Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

- In this scheme, each thread will be allotted a fixed amount of run-time before execution is passed to the next ready thread.
- If a thread blocks before its timeslice has expired, execution will be passed to the next ready thread.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Round-Robin Preemptive Scheduling

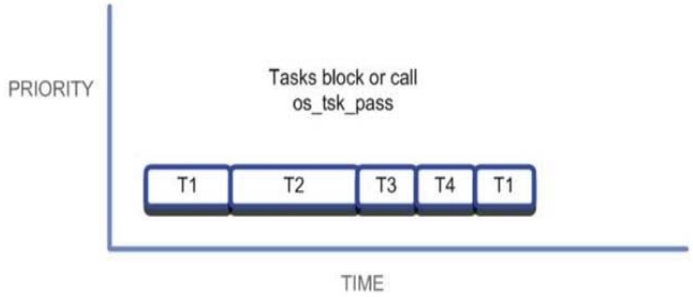
- The default scheduling option for the Keil RTX is round-robin preemptive.
- For most applications this is the most useful option and you should use this scheduling scheme unless there is a strong reason to do otherwise.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

Cooperative Multitasking

- A final scheduling option is cooperative multitasking.
- In this scheme, round-robin scheduling is disabled and each thread has the same priority.
- This means that the first thread to run will run forever unless it blocks.
- Then execution will pass to the next ready thread
- Threads can block on any of the standard OS objects, but there is also an additional OS call, osThreadYeild(), that schedules a thread to the ready state and passes execution to the next ready thread.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"



In a cooperative RTOS each thread will run until it reaches a blocking OS call or uses the osThreadYield() call.

Source: Trevor Martin, "The Designer's Guide to the Cortex-M Processor Family"

innovate achieve lead

Thank you

BITS Pilani, Deemed to be University under Section 3 of UGC Act, 1956

Q.A processor system uses 16-bit memory addresses. It has a 2K-byte cache organized in a direct-mapped manner with 64 bytes per cache block. Assume that the size of each memory word is 1 byte.
(A)Calculate the number of bits in each of the Tag, Index, and Offset fields of the memory address.
(B)When a program is executed, the processor reads data sequentially from the following word addresses: 128, 144.
All the above addresses are shown in decimal values. Assume that the cache is initially empty. For each of the above addresses, indicate whether the cache access will result in a hit or a miss.

Ans.
Block size = 64 bytes = 2^6 bytes = 2^6 words (since 1 word = 1 byte)
Therefore, Number of bits in the Offset field = 6

Cache size = 2K-byte = 2^{11} bytes
Number of cache blocks = Cache size / Block size = $2^{11}/2^6 = 2^5$
Therefore, Number of bits in the Index field = 5

Total number of address bits = 16
Therefore, Number of bits in the Tag field = 16 - 6 - 5 = 5

For a given 16-bit address, the 5 most significant bits represent the Tag, the next 5 bits represent the Index (Block), and the 6 least significant bits represent the Offset (Word).
Access # 1:
Address = $(128)_{10}$ = $(0000000010000000)_2$
(Note: Address is shown as a 16-bit number, because the processor uses 16-bit addresses)
For this address, Tag = 00000, Block = 00010, Word = 000000
Since the cache is empty before this access, this will be a cache miss
After this access, Tag field for cache block 00010 is set to 00000
Access # 2:
Address = $(144)_{10}$ = $(0000000010010000)_2$
For this address, Tag = 00000, Block = 00010, Word = 010000
Since tag field for cache block 00010 is 00000 before this access, this will be a cache hit (because address tag = block tag)

Q. Caches are important to providing a high-performance memory hierarchy to ARM processors. Below is a list of 64-bit memory address references, given as word addresses. 0xFFAA0003, 0xFFAA00B4, 0xFFAA002B, 0xFFAA0002, 0xFFAA00BF, 0xDDCC00BE, 0xDDCC00B5.
For each of these references, identify the binary word address, the tag, the index, and the offset given a direct-mapped cache with two-word blocks and a total size of eight blocks. Also list if each reference is a hit or a miss, assuming the cache is initially empty. [7]
Ans.
As cache has two word blocks, require offset 1 bit.
Total size is 8 blocks, index is 3 bits.
Remaining bits are Tag bits 64-3-1 = 60 bits.
At first time Tag=0xFFAA000 and Index =1 then it is miss but for second time it matches address tag and index bit, so it is hit. (address tag = block tag)

Word Address	Binary Address	Tag	Index	Offset	Hit/Miss
0xFFAA0003	0000 0011	0xFFAA000	1	1	M
0xFFAA00B4	1011 0100	0xFFAA00B	2	0	M
0xFFAA002B	0010 1011	0xFFAA002	5	1	M
0xFFAA0002	0000 0010	0xFFAA000	1	0	H
0xFFAA00BF	1011 1111	0xFFAA00B	7	1	M
0xDDCC00BE	1011 1110	0xDDCC00B	7	0	M
0xDDCC00B5	1011 0101	0xDDCC00B	2	1	M