EE 447

INTRODUCTION

to

MICROPROCESSORS

LECTURE NOTES

Fall 2018

Gözde Bozdağı Akar

İlkay Ulusoy

Contents

CHAPTER 1

Introduction to Microprocessors and Microcontrollers

1. Course Objectives

- To develop an in-depth understanding of
    - the operation of microprocessors and microcontrollers,
    - machine language programming,
    - microprocessor interfacing techniques,
    - designing and interfacing of microcontroller-based embedded systems in both hardware and software,
    - basic programming of ARM Cortex chips in assembly language and learning the fundamentals of embedded system design.
- Ultimate goal: to be able to apply this knowledge to more advanced structures.

2. Embedded systems

It is a special purpose system that is used to perform one of few dedicated functions. Simply any computer system embedded inside an electronic device but is not itself a general-purpose computer.

- dedicated to specific function(s)
- interact with environment
- real-time requirements

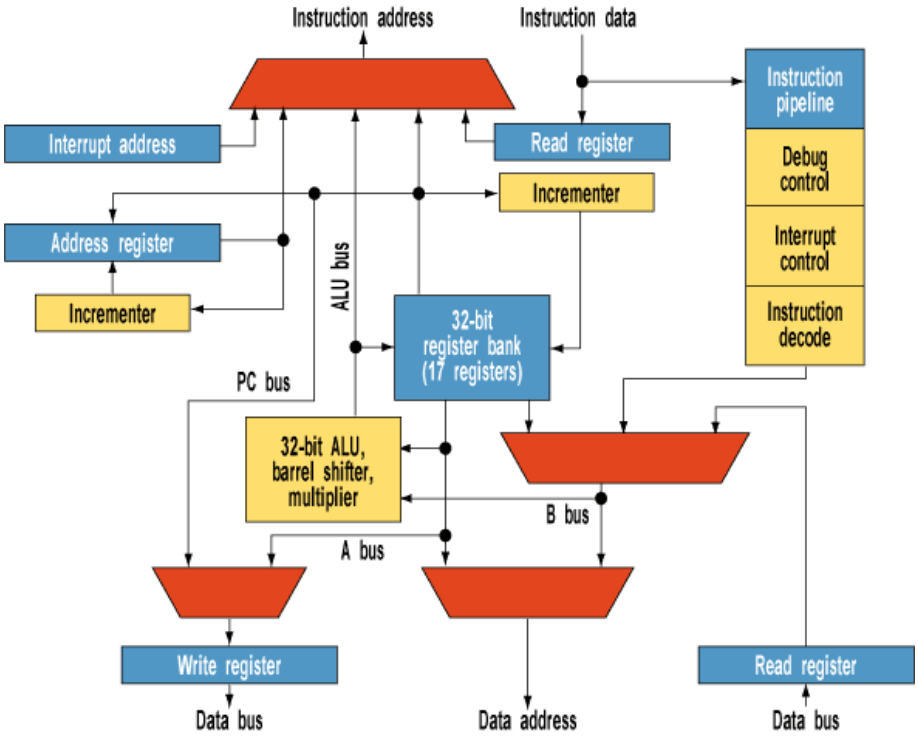
Application examples:

- Simple control: front panel of microwave oven, etc.
- Canon EOS 3 has three microprocessors: 32-bit RISC CPU runs auto-focus and eye control systems.
- Digital TV: programmable CPUs + hardwired logic.
- Smart phone: keyboard, communications, games, “applications”, etc.
- Vehicle control: automotive, aerospace, etc.
- Nuclear power plant control

How do we implement them:

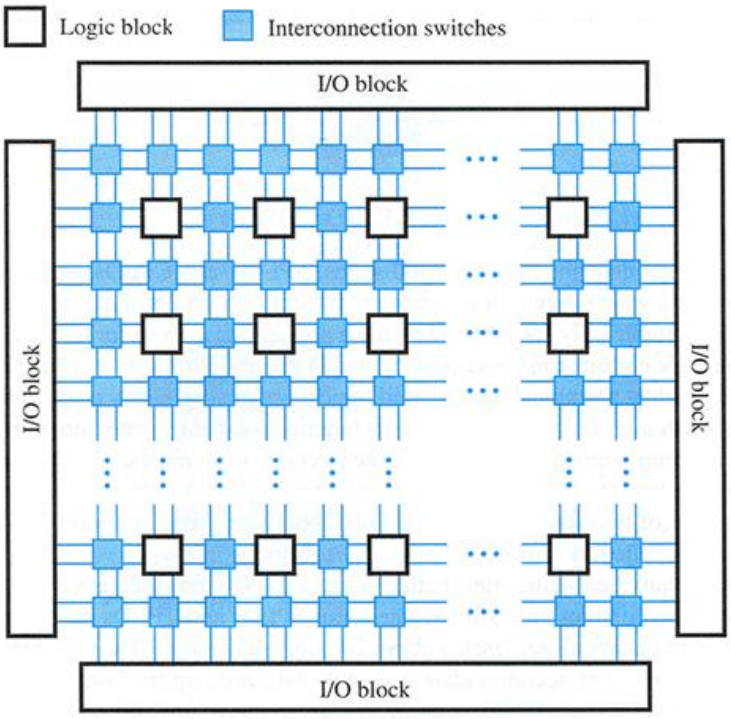
- By using microcontrollers (or microprocessor based systems): Slower (technology dependent), only software is needed to be updated.
- By using digital circuits (Field Programmable Gate Array - FPGA): Faster (only propagation delay), functions they perform can’t be changed easily.

MPU:



The Cortex M3's Thumbnail architecture looks like a conventional Arm processor. The differences are found in the Harvard architecture and the instruction decode that handles only Thumb and Thumb 2 instructions.

FPGA:



General structure of an FPGA

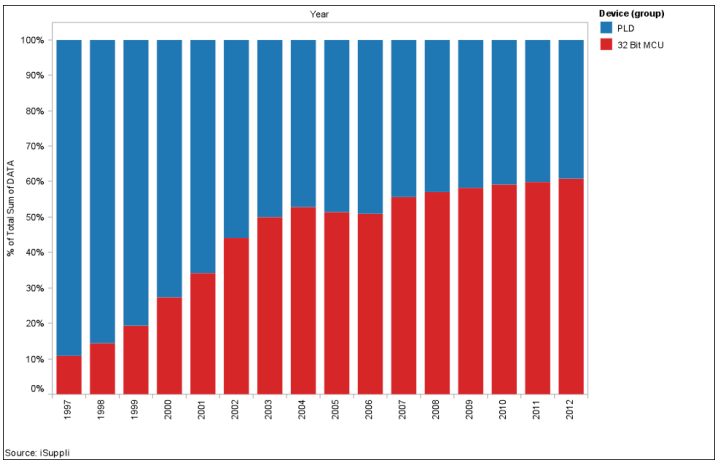


Figure 2-1. Usage of Programmable Logic Design and Microcontroller unit through time.

2.1. What is a Microcontroller?

It is a full computer system on a chip, even if its resources are far more limited than of a desktop personal computer. It is designed for stand alone applications.

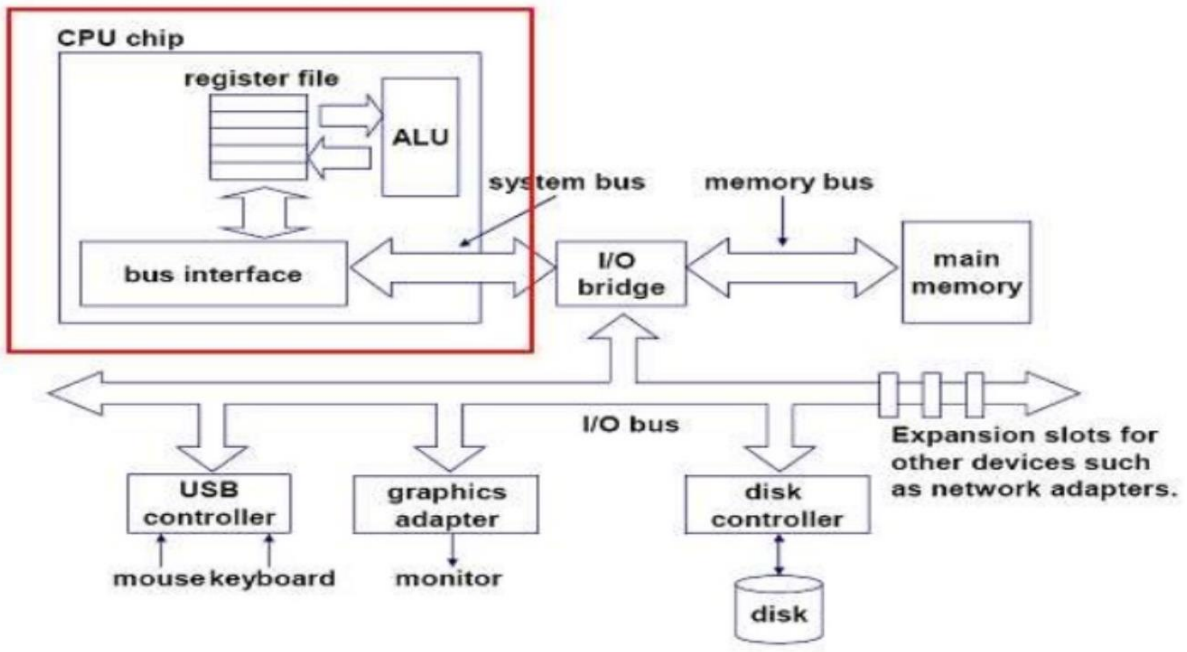


Figure 2-2 Microcontroller components and peripheral input/output devices.

The basic components:

–Processor (or central processing unit - CPU) with its associated temporary memory (registers and cache if available) for code execution,
–Main memory and secondary memory where code and data are temporarily and permanently stored,
–Input and output modules that provide interfaces between the processor and the user.

All these components are connected through an interface bus that consists of Address, Data, and Control signals.

2.2. What is a microprocessor?

A microprocessor is a component that performs the instructions and tasks involved in computer processing. A microprocessor integrated in a single ship with memory, I/O, and other necessary functionalities is a microcontroller. A microprocessor can be a single or multicore. A Multi-core processor is a single computing component that has two or more independent cores or processing units. These cores are the ones that read and perform your program's instructions on your CPU. Being a multi-core processing unit, it can execute multiple instructions at the same time. A multi-core processor is intended to lessen the heat coming off your CPU and to increase the speed of processing these instructions.

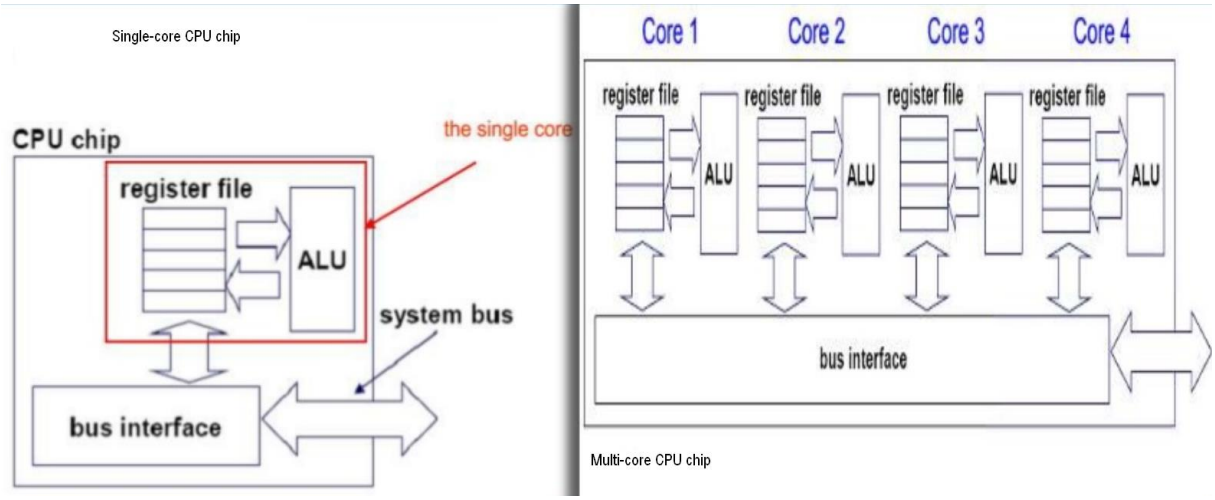


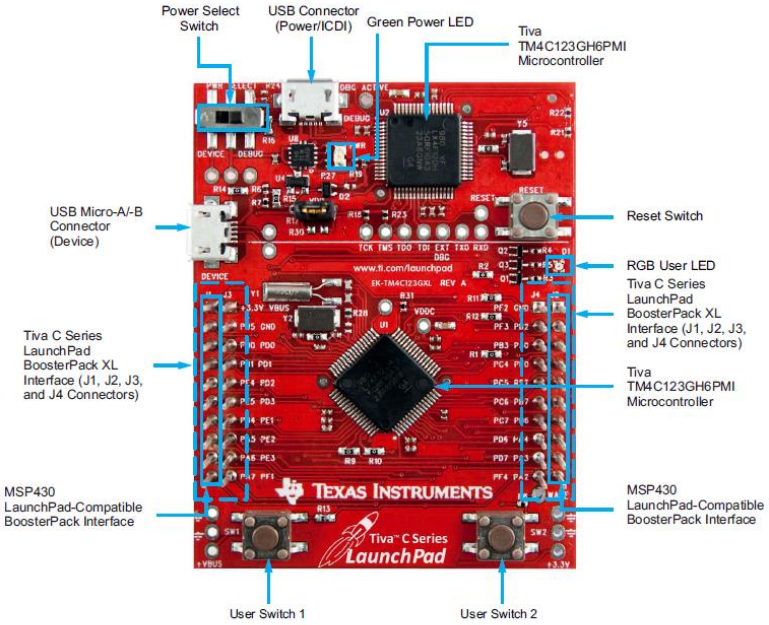
Figure 2-3 Single core and multicore chips

The basic components of a microprocessor are:

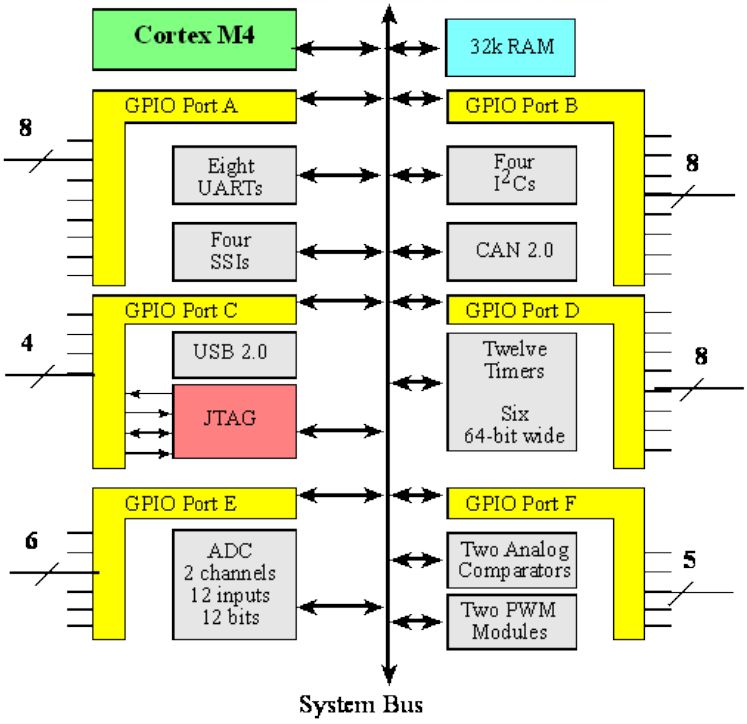
- ALU: Arithmetic Logic Unit is a circuitry which is capable of doing various operations such as ADD, SHIFT, AND, OR on certain on-chip regsiters.
- Control Unit: It directs operation of the processor. It tells the computer's memory, arithmetic/logic unit and input and output devices how to respond to a program's instructions.
- Registers: A register is a small amount of storage available as part of a microprocessor.
- System Bus: A single computer bus that connects the major components of a computer system, combining the functions of a data bus to carry information, an address bus to determine where it should be sent, and a control bus to determine its operation such as reading or writing.

Although many microcontroller vendors use the same processor as their choice of CPU, the memory system, memory map, peripherals, and operation characteristics (e.g., clock speed and voltage) can be completed differently from one product to another. This allows microcontroller manufacturers to add additional features in their products and differentiate their products from others on the market. For details of the complete microcontroller system design, such as peripheral details, memory map, and I/O pin assignments, you still need to read the reference manuals provided by the microcontroller vendor.

Below is the development board of Texas instruments Tiva C Series TM4C123G board including ARM Cortex 4 microprocessor and other components.



Tiva C Series TM4C123G LaunchPad Evaluation Board



2.3. I/O Ports

The external devices attached to the microcontroller provide functionality for the system. An input port is hardware on the microcontroller that allows information about the external world to be entered into the computer. The microcontroller also has hardware called an output port to send information out to the external world. An interface is defined as the collection of the I/O port, external electronics, physical devices, and the software, which combine to allow

the computer to communicate with the external world. In general, we can classify I/O interfaces into four categories

- Parallel - binary data are available simultaneously on a group of lines
- Serial - binary data are available one bit at a time on a single line
- Analog - data are encoded as an electrical voltage, current, or power
- Time - data are encoded as a period, frequency, pulse width, or phase shift

For the Tiva C series board, the following I/O ports are available:

- UART (Universal asynchronous receiver/transmitter)
- SSI (Synhronous serial interface)
- I2C (Inter-integrated circuit)
- Timer (Periodic interrupts, input capture, output compare)
- PWM (Pulse width modulation)
- ADC (Analog to digital converter)
- Analog comparator (Copare two analog signals)
- QEI (Quadrature encoder interface)
- USB (Universal serial bus)
- Ethernet (High speed network)
- CAN (Controller area network)

2.4. Memory

Random access memory (RAM): same amount of time is required to access any location on the same chip. Read/write.

Read-only memory (ROM): can only be read; cannot be written to directly by the processor.

Dynamic Random Access Memory (DRAM): periodic refresh is required to maintain the contents of a DRAM chip.

Static Random Access Memory (SRAM): no periodic refresh is required. Always more predictable and usually faster than DRAM.

Synchronous DRAM (SDRAM): SDRAM has a synchronous interface, meaning that a clock signal must be received before it responds to the control inputs. The memory is divided into several independent sections of memory called *banks*, allowing the device to operate on several memory access commands at a time, provided the commands are independent of each other (in an interleaved fashion). This allows SDRAMs to achieve greater concurrency and higher data transfer rates than asynchronous DRAM.

Mask-programmed read-only memory (MROM): programmed when being manufactured.

Programmable read-only memory (PROM): the memory chip can be programmed by the end user.

Erasable Programmable ROM (EPROM): electrically programmable many times, erased by ultraviolet light (through a window), erasable in bulk (whole chip in one erasure operation).

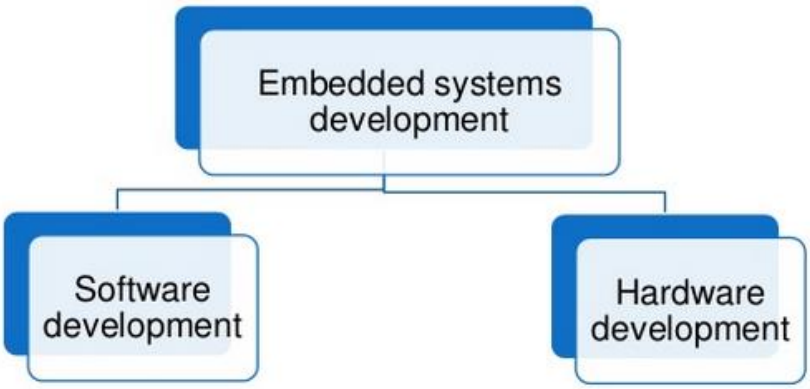
Electrically Erasable Programmable ROM (EEPROM): electrically programmable many times, electrically erasable many times, can be erased one location, one row, or whole chip in one operation

Flash Memory: electrically programmable many times, electrically erasable many times, can only be erased in bulk.

Tiva C Series TM4C123G microcontroller has 256KB of flash memory, 32KB of RAM, 2KB of EEPROM.

2.5. Developing an embedded application using a microcontroller unit

Embedded system development requires synhronous hardware and software development.



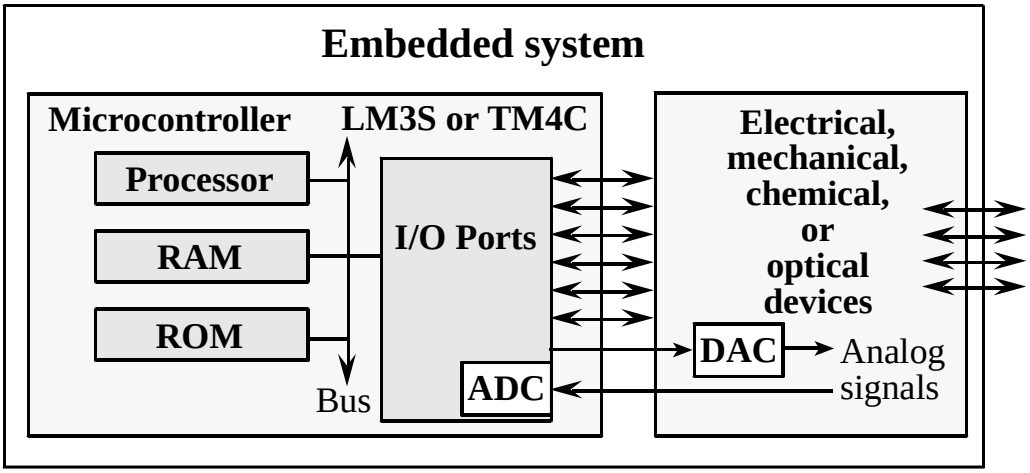


Figure 2-4 Embedded system hardware.

Hardware development

Factors to be considered when selecting a microcontroller device for a product:

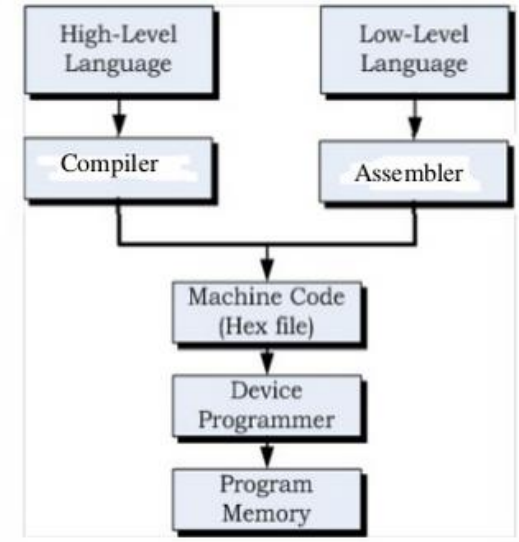
- Peripherals and interface features
  - Number of I/O ports
  - Serial communication modules
  - Peripherals like timer, ADC, PWM, etc
- Memory size requirements of the application
- Processing speed requirements
- Low power requirements
- Performance and maximum frequency
- Chip package

- Operation conditions (voltage, temperature, electromagnetic interference)
- Cost and availability
- Software development tool support and development kits
- Future upgradability
- Firmware packages and firmware security
- Availability of application notes, design examples, and support

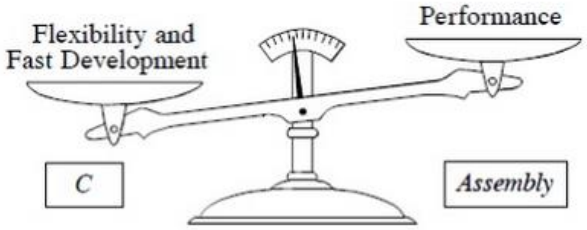
Software development

A program, which is actually a sequence of instructions, is known as software.

- Machine Instruction:
    - A sequence of binary digits that can be executed by the processor. Hard to understand, program, and debug for human being.
    - E.g. in the ARM Cortex M4, instruction 0010 0010 0110 0100 (in binary, 0x2264 in hexadecimal) puts the number 100 in register #2.
- Assembly Language:
    - Defined by machine instructions. An assembly instruction is a mnemonic representation of a machine instruction.
    - E.g. ADD represents a register add, ADD R2 R1 R0 means R1 and R0 should be added and the result should be saved to R2.
    - Assembly programs must be translated to binary before it can be executed: Assembler does translation from ADD R2 R1 R0 to 0001100 010 001 000 = 0x1888
    - In this course we'll be mostly dealing with Assembly Language.
- High-level Language
    - Syntax of a high-level language is similar to English.
    - A translator is required to translate the program written in a high-level language. This is done by a compiler.
    - High-level languages allow the user to work on the program logic at a more conceptual level (e.g. you don't have to worry about what's in register R2).
    - In this course we will see some C language examples, which are helpful if you understand C, but is not necessary to this course.

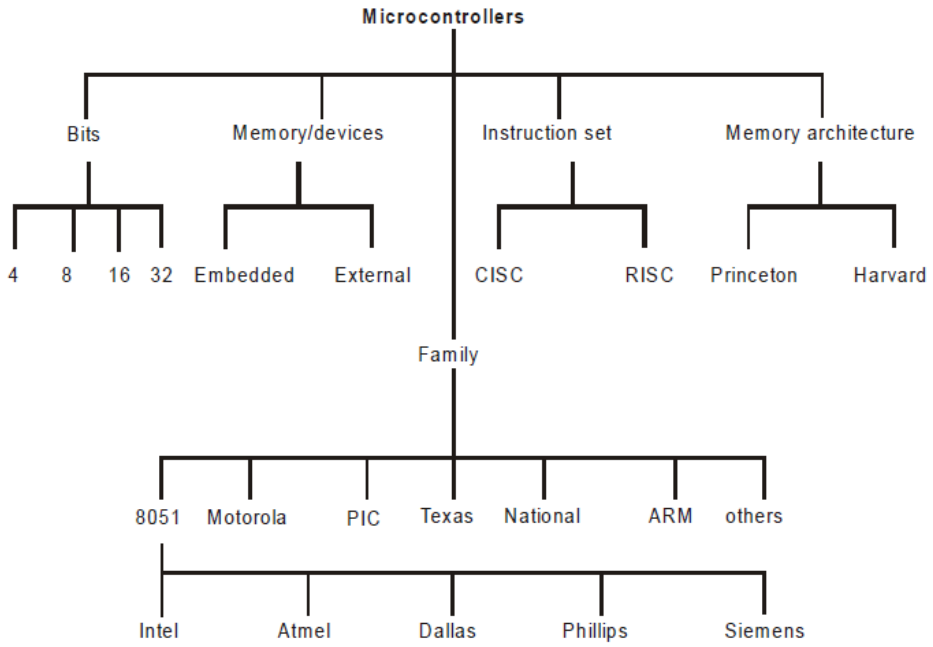


Assembly programs are optimized more than C programs, but to develop more complicated programs using C is more practical and also efficient.



3. Basics of Microcontroller Architecture

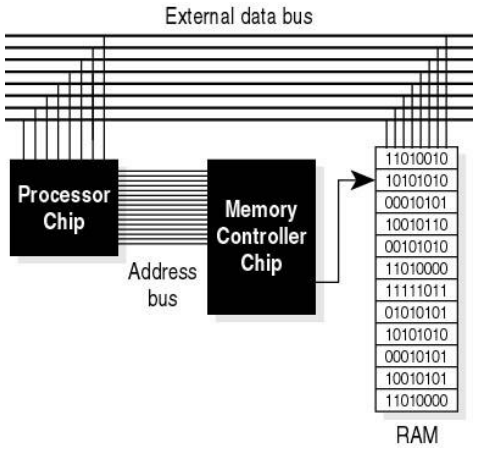
Microcontrollers produced by various vendors (Motorola, Intel, Texas, ...) can have different word sizes (4, 8, 16, 32, ..), memory types (registers, RAM, ROM, ...) and sizes (e.g. 64 K, 4 G, ...), instructions sets and memory architectures.



3.1. Address space

Memory consists of a sequence of directly addressable locations. A location is referred to as an information unit. A memory location can be used to store data and instructions. A memory location has two components: an address and its contents.

Data transfers between the CPU and the memory are done over the common buses: address bus and data bus.



The address space of a processor depends on its address decoding mechanism: Its size will depend on the number of address bits used.

Depending on the processor design, there may be different approaches for address space usage:

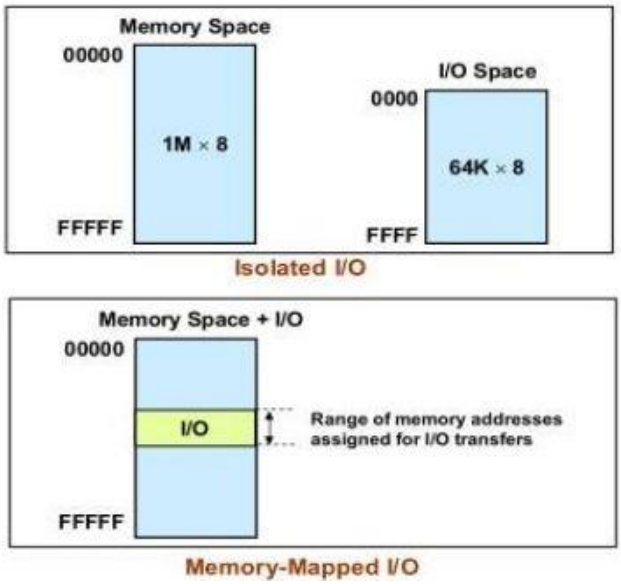
1. Isolated I/O (E.g. Intel):

- one space is used by normal memory access
- another space is reserved for I/O peripheral registers (control, status, and data)

- need extra control signal or special means of accessing these alternate address space

2. Memory mapped I/O (E.g. ARM, Motorola)

- utilize only one address space for both memory and I/O devices



Cortex-M4 processor has 32 bits for addressing and this means that it is supporting 4GB of memory space. The memory space is used by the program code, data, peripherals (I/O devices and memory), and some of the debug support components inside the processors.

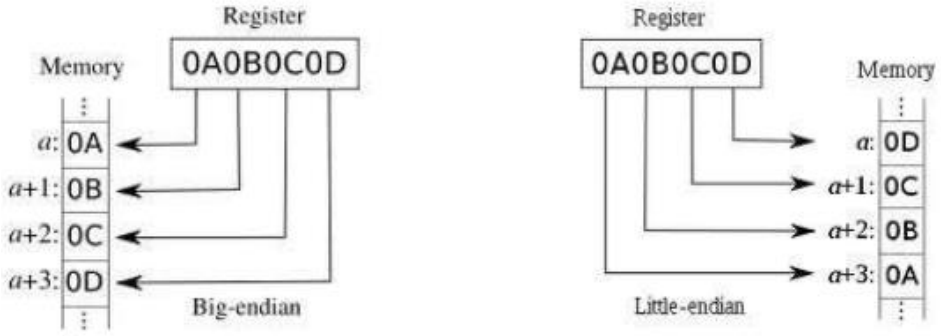
The processor enables a memory chip or I/O device only when it wants to address it. This is done by an address decoding. Details about address decoding is given in Appendix 1.

3.2. Data Organization in Memory

Typically, memory contains a storage locations that can store data of a certain fixed size (most commonly of the 8-bit (byte) size). Each location is provided with a unique address. Depending on the data path size of the processor, the memory content is accessible in the size of an 8-bit byte, a 16-bit half word, a 32-bit word, and even a 64-bit double word.

A 32-bit data consists of four bytes of data, which are stored in four successive memory locations but what is the order of the four bytes of data? Depends on the Endianness adopted.

- In the Little Endian format, the least significant byte (LSB) is stored in the lowest address of the memory, with the most significant byte (MSB) stored in the highest address location of the memory. (ARM is Little Endian by default)
- In the Big Endian format, the least significant byte (LSB) is stored in the highest address of the memory, with the most significant byte (MSB) stored in the lowest address location of the memory.



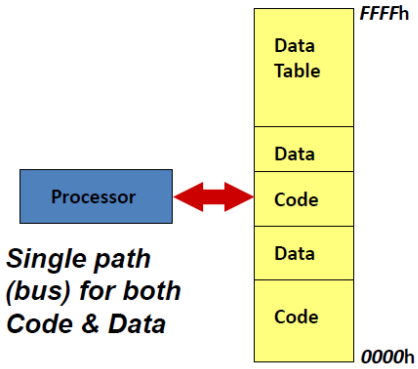
3.3. Memory architecture

Two classes of memory systems have been designed to store the information:

- Von Neumann architecture
- Harvard architecture

Von Neumann architecture:

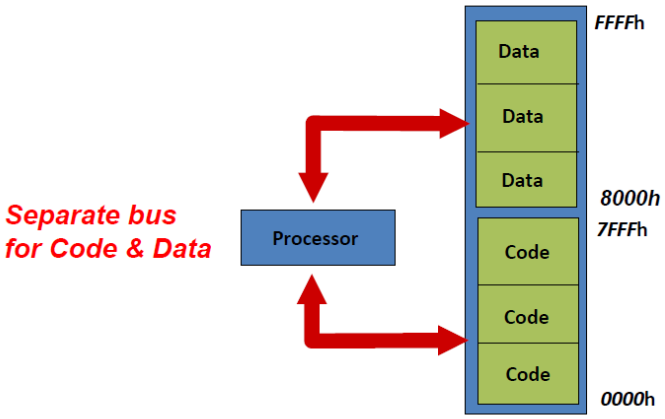
- Data and program can reside in the same memory. More efficient use of memory.
- Single memory interface bus. CPU provides address to get data or instructions. Simplifies the hardware design but bottleneck in code and data transfer.
- More flexible pogramming but data may overwrite code.



Harvard architecture:

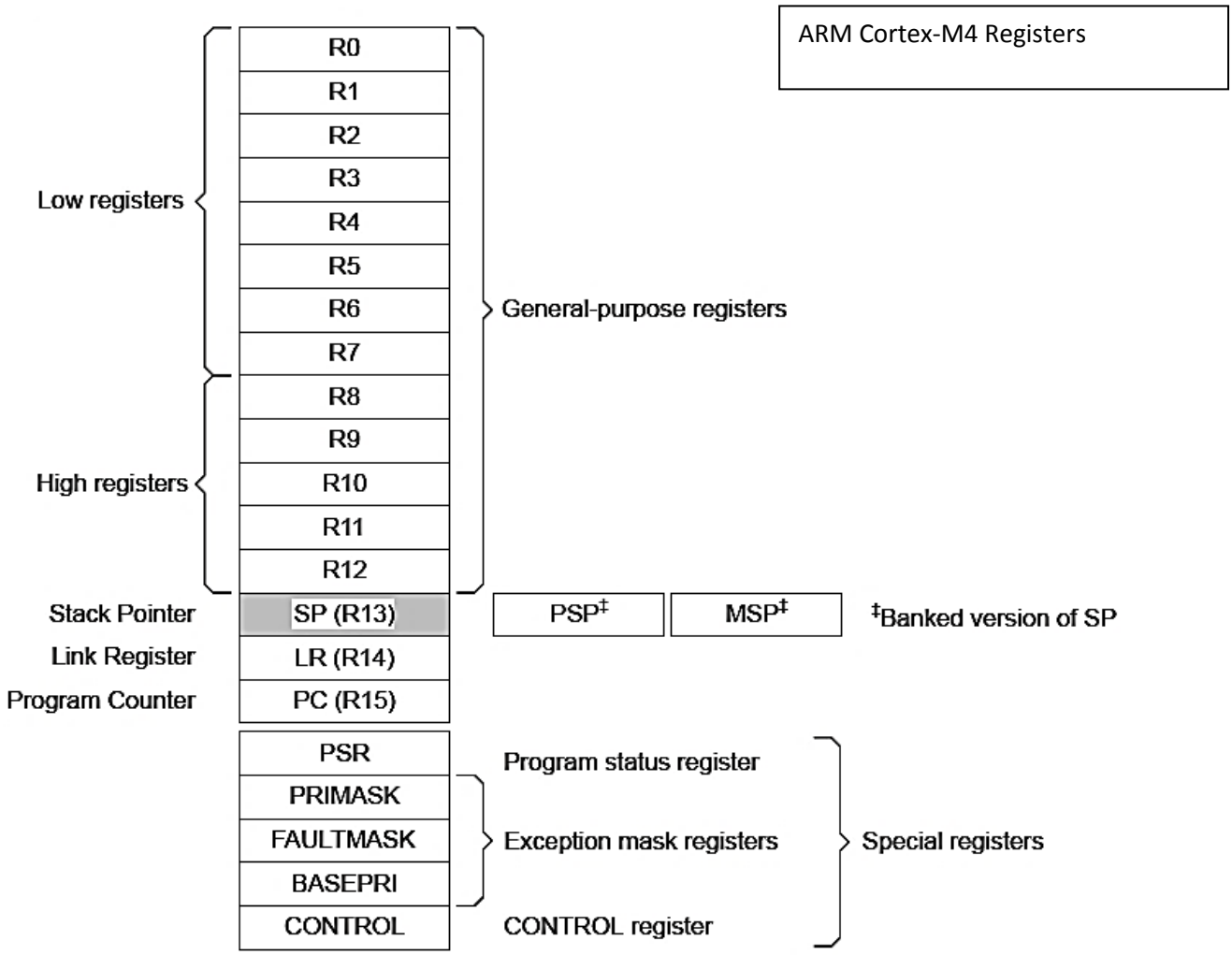
- Seperate instruction bus and data bus (code and data may still share the same memory).
  - allow code and data access at the same time which gives improved performance
  - provide better support for instruction pipeline operations and shorter instruction execution time
  - allow different sizes of data and instructions to be used which results in more flexibility
  - do not incur any code corruption by data which makes the operations more robust
- More sophisticated hardware logic is required to support multiple interface buses.

Cortex M4 is based on Harvard architecture.



3.4. Registers

- The most fundamental storage area in the processor
  - is closely located to the processor
  - provides very fast access, operating at the same frequency as the processor clock
  - but is of limited quantity (typically less than 100)
- Most are of the general purpose type and can store any type of information:
  - data –e.g., timer value, constants
  - address –e.g., ASCII table, stack
- Some are reserved for specific purposes
  - program counter
  - program status register



3.5. Instruction Set Architecture

Instruction Set Architecture (ISA) can be defined as an interface to allow easy communication between the programmer and the hardware. ISA prepares microprocessor to respond to all the user commands like execution of data, copying data, deleting it, editing it and several such and diverse operations.

Two major items in ISA are:

- Instruction Set: It is a group of instructions that can be given to the computer. These instructions direct the computer in terms of data manipulation. A typical instruction consists of two parts: Opcode and Operand.
    - Opcode or operational code is the instruction applied. It can be loading data, storing data etc.
    - Operand is the memory register or data upon which instruction is applied.
    - Ex: ADD R3,R2,R1; R2+R1 -> R3
- Addressing Modes: Addressing modes are the manner how the data is accessed. Depending upon the type of instruction applied, addressing modes are of various types such as direct mode where straight data is accessed or indirect mode where the location of the data is accessed.
    - Ex: ADD R3,R2,R1; R2+R1 -> R3
      ADD R3,R3,#1; R3+ 1 -> R3

There are two prevalent instruction set architectures:

- Complex Instruction Set Architecture (CISC) : The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. For this particular task, a CISC processor would come prepared with a complex instruction such as "MULT" (multiply). In CISC we can use Mult a,b to multiply two numbers in memory. However in RICS first they have to be loaded to registers, multiplied and stored back.
    - Advantages:
        - As each instruction can execute several low level operations: The code size is reduced to save on memory requirements, less main memory access is required and hence processing time is reduced (faster)
        - Backward code compatibility is maintained: Can add new (and more powerful) instructions while retaining the 'old' instruction set for code compatibility (i.e. legacy programs can still run)
        - Easy to program: Direct support of high-level language constructs, complex instructions that fit well with high-level language expressions
    - Disadvantages:
        - A highly encoded instruction set needs to be decoded by hardwired microcode electronic circuitry: More complex hardware design, slower instruction decoding/execution
        - Variable length instructions: Different execution time among instructions, affects pipelined operations

- Reduced Instruction Set Architecture (RISC):
    - RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program. RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" instruction described above could be divided into multiple separate instructions
    - Advantages:
        - Simpler instructions: One clock per instruction gives faster execution than on a CISC processor with the same clock speed
        - Simpler addressing mode: Faster decoding
        - Fixed length instructions: Faster decoding and better pipeline performance
        - Simpler hardware: Less silicon area, less power consumption
    - Disadvantages:
        - Fewer instructions than CISC: As compared to CISC, RISC needs more instructions to execute one task, needs more memory
          No complex instructions: No hardware support for division or floating-point arithmetic operations, needs a more complex compiler and longer compiling time

To reduce memory requirements and cost, ARM provides the 16-bit Thumb instruction set and 16/32-bit Thumb2 instruction set as an option for its RISC processor cores.

3.6. Instruction Execution

Multiple stages are involved in executing an instruction hence multiple processor clock cycles are needed to execute one single instruction.

- Fetching the instruction code

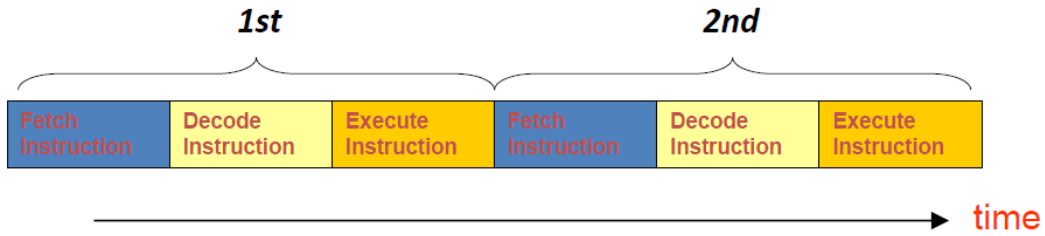
- Decoding the instruction code
- Executing the instruction code

Instruction Execution Example:

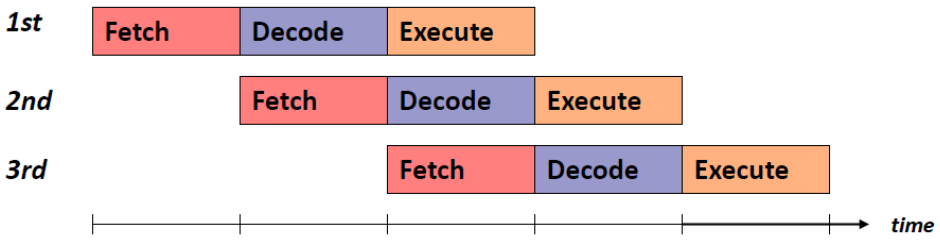
Execute ADD R4,R3,R0 and assume this instruction begins at address 0x00001000. This instruction means: add contents of register R0 to the contents of register R3, and store the results into register R4.

1.Fetch instruction (read) from the memory addresses pointed by the PC. Assume that PC= 0x00001000. Then fetch from 0x00001000 through 0x00001003. The contents of these are the computer's bit pattern for 'ADD R4,R3,R0'. Store these 32 bits in instruction register, IR.
2.Decode the instruction in IR. Determine that it is the ADD instruction.
3. Connect register R0 to input 1 of the ALU.
4. Connect register R3 to input 2 of the ALU.
5. Set controls to ALU to do a 32-bit add operation.
6. Store the outputs of the ALU into register R4.
7.Increment the program counter and store that value back in the program counter: [PC] + 4 -> PC. This operation can actually occur after step 1, in parallel with steps 2-6.
8.Go back to step 1, using the new value of PC.

This fetching-decoding-execution can continue as successive cycles or in a pipelined manner.

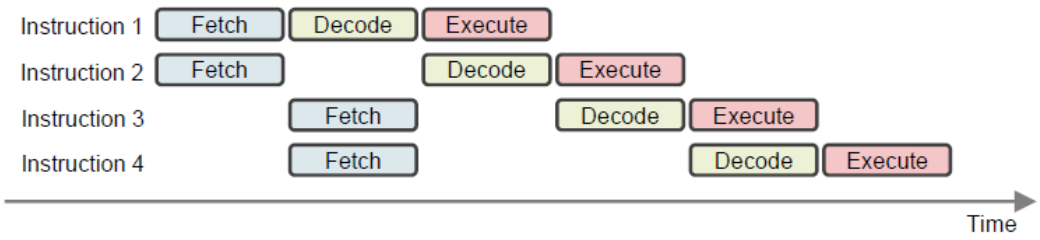


OR



Cortex M4 uses 3 stage pipeline.

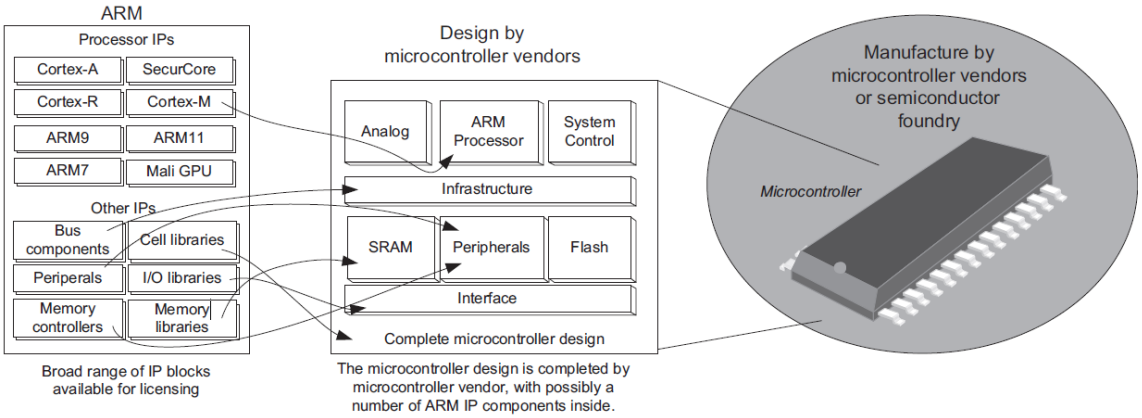
Ex:



4. ARM architecture

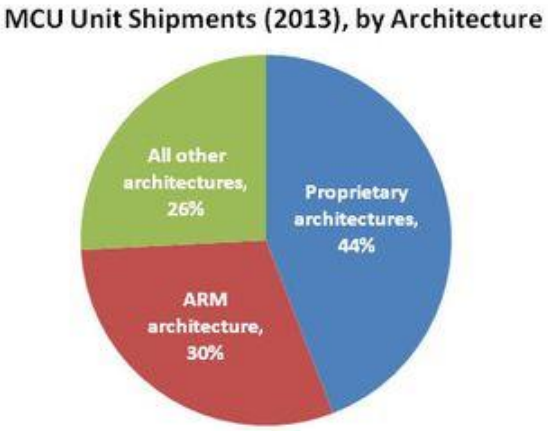
4.1. ARM history, cores, families

The ARM architecture was originally developed by Acorn Computers in the 1980s. ARM Ltd spun out of Acorn Computers in 1990. They license ARM core designs to semiconductor partners who fabricate and sell to their customers.



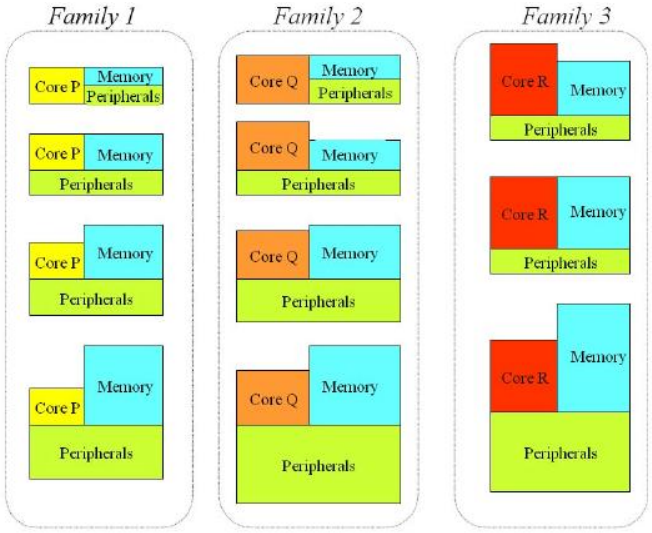
An ARM processor is one of a family of CPUs based on the RISC (reduced instruction set computer) architecture developed by Advanced RISC Machines (ARM). ARM processors are extensively used in consumer electronic devices such as smartphones, tablets, multimedia players and other mobile devices, such as wearables. Because of their reduced instruction set, they require fewer transistors, which enables a smaller die size for the integrated circuitry (IC). The ARM processor's smaller size, reduced complexity and lower power consumption makes them suitable for increasingly miniaturized devices.

For microcontrollers (MCUs) used in embedded devices, ARM is the clear market leader as shown in a recent forecast for VDC Research's report "The Global Market for Embedded Processors.

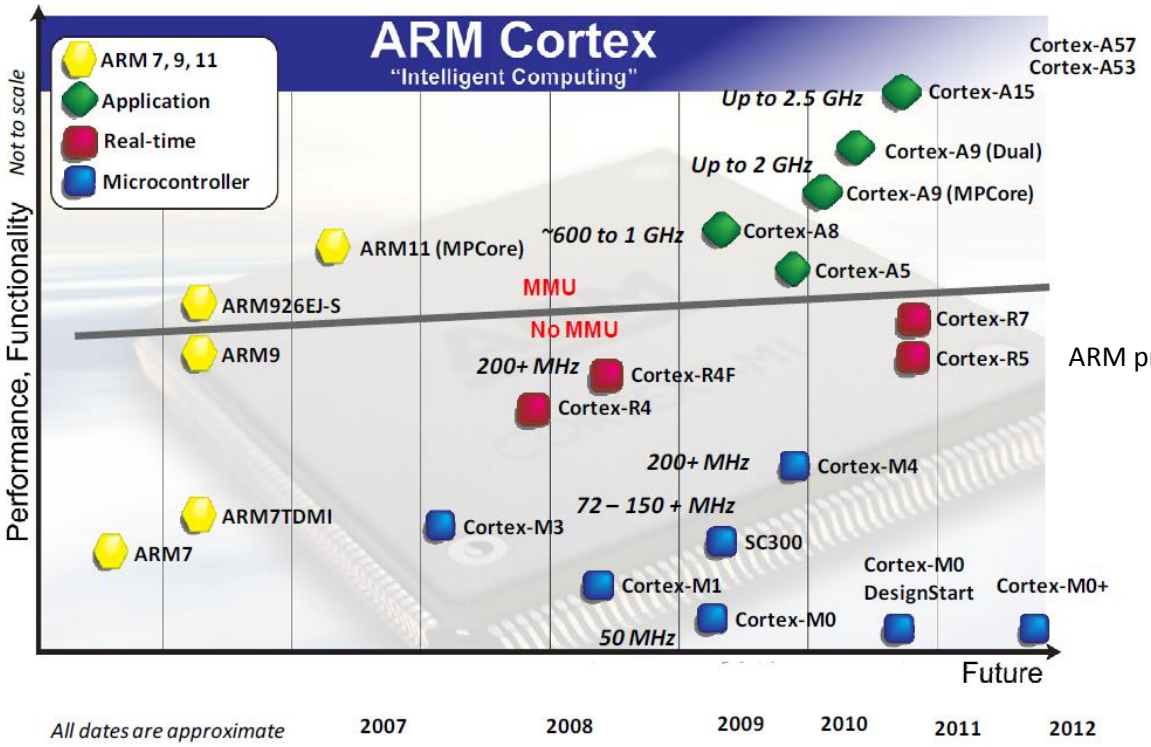


ARM has several designs which can be grouped in different architectures, families and cores. The architectures are the specifications, i.e. the set of registers, instructions and operation modes that should be supported by implementations of the architecture. A family is

a specific detailed implementation of an architecture, i.e. the actual hardware details needed to create an ARM core. Finally a core is a specific implementation of an architecture, i.e. the actual blue-print of the transistors and other discrete parts needed to create a ARM CPU.



The ARM architectures are modular so families may implement only some features of the architecture and not others. ARM11 and ARM Cortex-M are different families implementing features in the ARMv6 architecture with different markets in mind: ARM-11 smartphones and tablets and Cortex-M micro-controllers.



ARM processor families

ARM processor architectures

Architecture	Bit width	Cores designed by ARM Holdings	Cores designed by 3rd parties	Cortex profile
ARMv1	32/26	ARM1		
ARMv2	32/26	ARM2, ARM3	Amber	
ARMv3	32	ARM6, ARM7		
ARMv4	32	ARM8	StrongARM, FA526	
ARMv4T	32	ARM7TDMI, ARM9TDMI		
ARMv5	32	ARM7EJ, ARM9E, ARM10E	XScale, FA626TE, Feroceon, PJ1/Mohawk	
ARMv6	32	ARM11		
ARMv6-M	32	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1		Microcontroller
ARMv7-M	32	ARM Cortex-M3		Microcontroller
ARMv7E-M	32	ARM Cortex-M4		Microcontroller
ARMv7-R	32	ARM Cortex-R4, ARM Cortex-R5, ARM Cortex-R7		Real-time
ARMv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15	Krait, Scorpion, PJ4/Sheeva, Apple A6/A6X (Swift)	Application
ARMv8-A	64/32	ARM Cortex-A53, ARM Cortex-A57	X-Gene, Denver, Apple A7 (Cyclone)	Application
ARMv8-R	32	No announcements yet		Real-time

- ARMv4T architecture: Introduced the 16-bit Thumb instruction set alongside the 32-bit ARM instruction set.
- ARMv5TEJ architecture: Introduced arithmetic support for digital signal processing (DSP) algorithms.
- ARMv6 architecture: Introduced an array of new features including the Single Instruction Multiple Data (SIMD) operations.
- ARMv7 architecture: Implements Thumb-2 technology.
    - Cortex-A: Implements a virtual memory system architecture based on an MMU, an optional NEON processing unit for multimedia applications and advanced hardware Floating Point.
    - Cortex-R: Implements a protected memory system architecture based on an MPU (memory protection unit).
    - Cortex-M: Microcontroller profile designed for fast interrupt processing.

ARM Processor Licences

- ARMv8-A: NVIDIA, Applied Micro, Cavium, AMD, Broadcom, Calxeda, HiSilicon, Samsung and STMicroelectronics
- Cortex-A15: ST-Ericson, TI, Samsung, nVIDIA
- Cortex-A9: NEC, nVIDIA, STMicroelectronics, TI, Toshiba …
- Cortex-A8: Broadcom, Freescale, Matsushita, Samsung, STMicroelectronics, Texas Instruments, PMC-Sierra
- Cortex-A5: AMD ---
- Cortex-R4(F): Broadcom, Texas Instruments, Toshiba, Inf
- Cortex-M4: Freescale, NXP, Atmel, ST

- Cortex-M3: Actel, Broadcom, Energy Micro, Luminary, Micro, NXP, STMicroelectronics, TI, Toshiba, Zilog, …
- Cortex-M0: Austria-microsystems, Chungbuk Technopark, NXP, Triad Semiconductor, Melfas
- Cortex-M0+: Freescale, NXP

Where to use?

ARM 7 – Introduced in 1994, used for simple 32-bir devices.
ARM9 – Was most popular ARM family (over 5 billion sold), used for smarthphones, HDD controllers, set top box, etc.
ARM11 - Extreme low power, many of today's smartphones.
Cortex M-series: Cost-sensitive solutions for microcontroller applications, system clock<200MHz

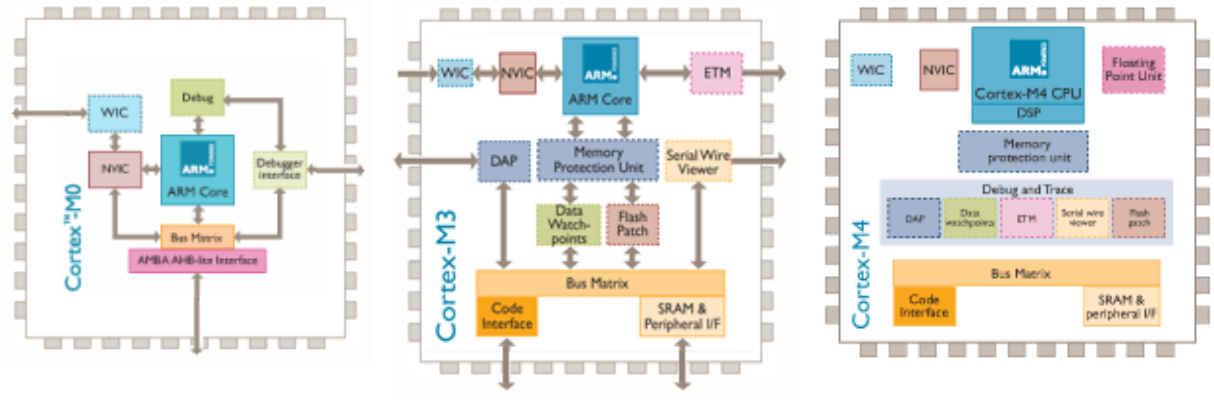
Cortex M0 – Ultra low-power, oltra low gate count,
Cortex M1 – First ARM processor designed specifically for implementation in FPGAs.
Cortex M3 – High performance and energy efficiency. Microcontroller applications.
Cortex M4 – Embedded processor for DSP
Cortex M7 – 2x Performance of M4

Cortex R-series: Exceptional performance for real-time applications, system clock<600MHz

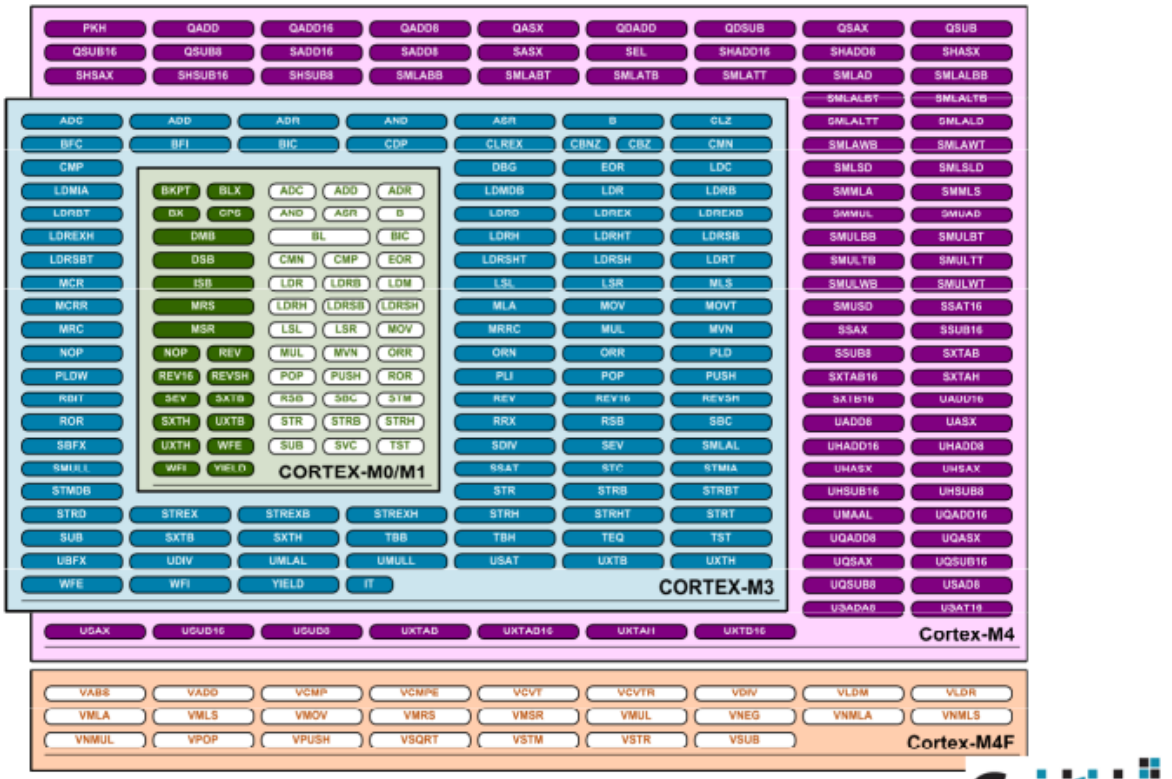
Cortex R4 - First embedded real-time processor based on the ARMv7-R architecture for high-volume deeply-embedded System-on-Chip applications such as hard disk, wireless baseband processors, electronic control units for automotive systems.
Cortex R5 – Extends the feature set of Cortex-R4, incereased efficiency and reliability.
Cortex R7 – High-performance dual core

Cortex A-series: High performance processors for open Operating sys, system clock>1GHz

Cortex A5 – Power and cost sensitive applications, smartphones.
Cortex A8 – Suitable for high-end phones, printers, DTVs
Cortex A9 – 1 to 4 cores. High performance-low power devices
Cortex A15 – Ultra low-power. Suitable for mobile computing, wireless infrastructure



Instruction set concerning M0, M1, M3, M4 and M4F



Cortex
Low-Power Leadership from ARM

In summary, ARM is a Reduced Instruction Set Computer (RISC), as it incorporates these typical RISC architecture features:

- a large uniform register file
- a load/store architecture, where data-processing operations only operate on register contents, not directly on memory contents
- simple addressing modes, with all load/store addresses being determined from register contents and instruction fields only
- uniform and fixed-length instruction fields, to simplify instruction decode.

In addition, the ARM architecture provides:

- Variable cycle execution for certain instructions (Load/store instructions executed in multiple cycles depending on the number of registers being transferred)
- Inline barrel shifter leading to more complex instructions (hardware component that preprocesses input registers before it is used by instruction which improve the performance)
- Thumb 16-bit instruction set (higher code density)
- Conditional execution (instructions executed only when specific conditions satisfied so increase the performance by reducing the brach instructions)
- DSP like enhanced instructions.

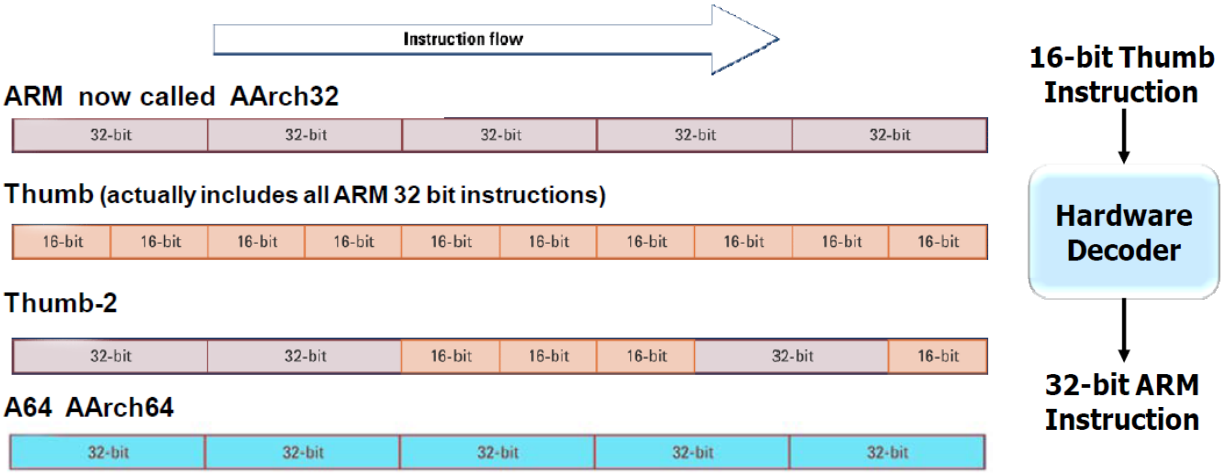
These enhancements to a basic RISC architecture allow ARM processors to achieve a good balance of high performance, small code size, low power consumption, and small silicon area. Even though these are common to all ARM architectures, depending on the families there are differences as stated before.

Throughout this course, we will concentrate on Cortex-M4.

4.2. Features of Cortex-M4

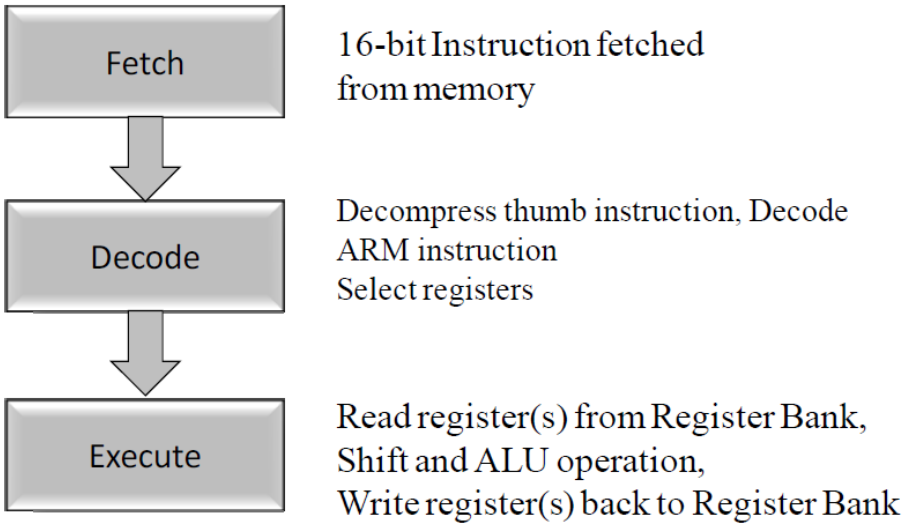
Cortex processors are the new generation of ARM processors. The CortexM3 processor was released by ARM in 2005. The Cortex-M4 processor was released in 2010. Cortex-M4 processor is to deliver higher performance in DSP applications, and to support floating point operations. As a result, some of the instructions available on both processors can be executed in fewer clock cycles on the Cortex-M4.

Processors use a **32-bit architecture**. Internal registers in the register bank, the data path, and the bus interfaces are all 32 bits wide. The Instruction Set Architecture (ISA) in the Cortex-M processors is called the Thumb ISA and is based on Thumb-2 Technology which supports a mixture of 16-bit and 32-bit instructions. This enables very high code density. (In general the processor may choose to operate in ARM mode or THUMB mode using a special instruction. However ARM instructions cannot be executed by the Cortex-M4 processor.)

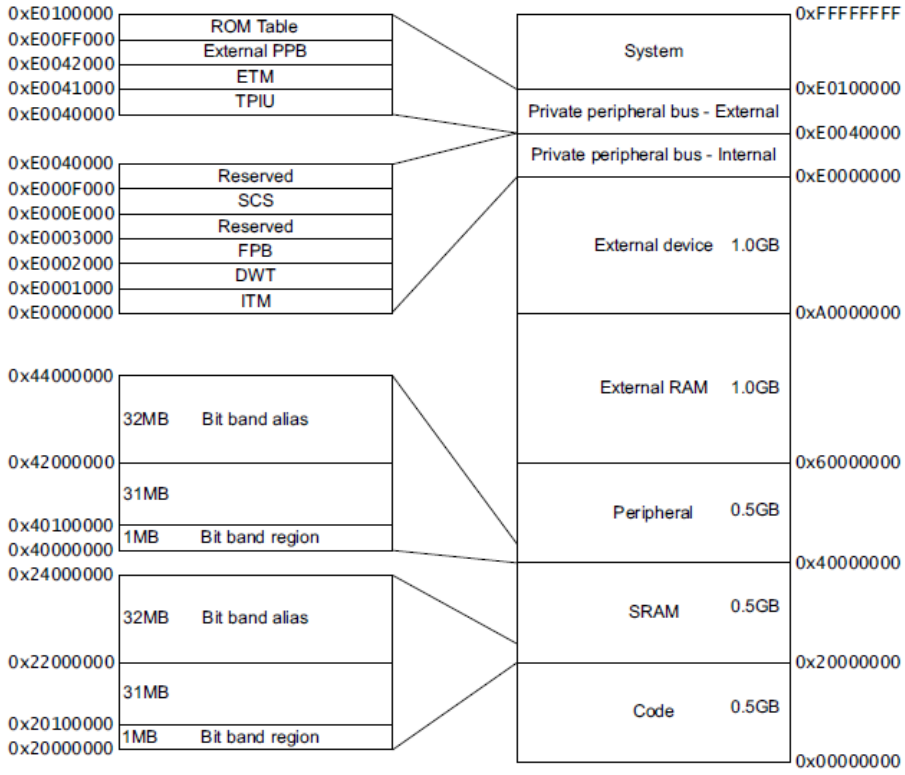


The Cortex-M4 processor (ARMv7-M architecture) have:

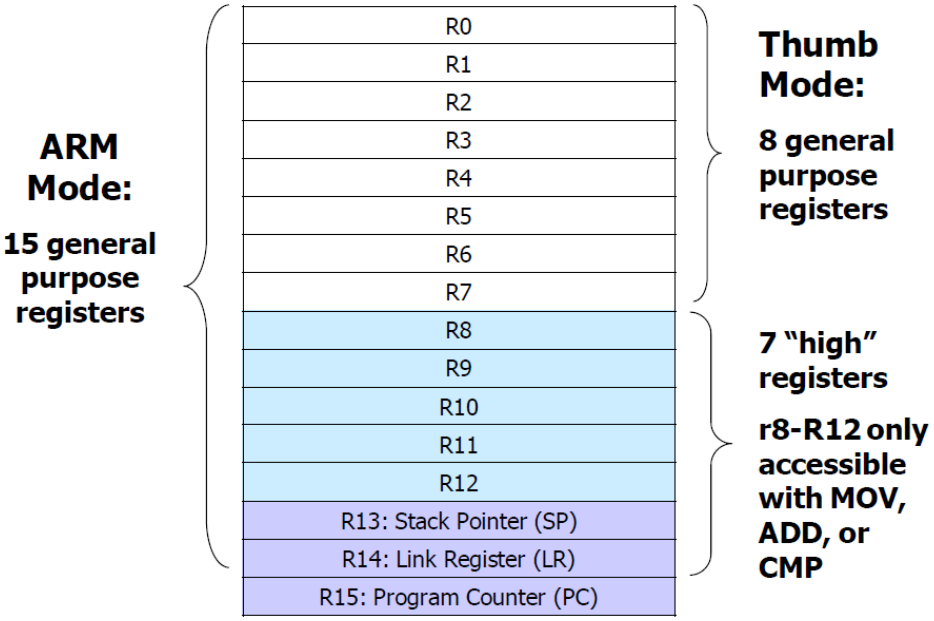
- Three-stage (instruction fetch, decode, and execution) pipeline design



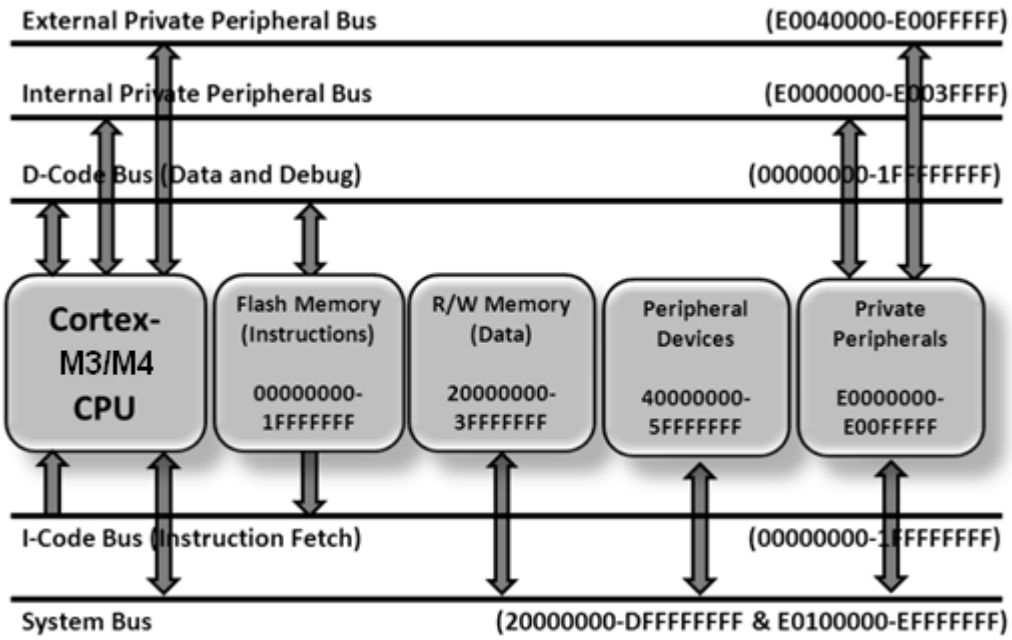
- Harvard bus architecture with unified memory space: instructions and data use the same address space, i.e., simultaneous instruction fetches and data accesses.
- 32-bit addressing, supporting 4GB of memory space. The memory map is unified, which means that although there can be multiple bus interfaces, there is only one 4GB memory space. The memory space is used by the program code, data, peripherals, and some of the debug support components inside the processors.



- 32-bit registers



- 32-bit internal data path
- 32-bit bus interface



- In addition to 32-bit data, the Cortex-M processors (as well as any other ARM processors) can also handle 8-bit, and 16-bit data efficiently. They also support a number of operations involving 64-bit data.
- On-chip bus interfaces based on ARM AMBA (Advanced Microcontroller Bus Architecture) Technology, which allow pipelined bus operations for higherthroughput
- 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)
- 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
- Support for bit-data accesses in two specific memory regions using a feature called Bit Band
- The option of being used in single processor or multi-processor designs

The ISA used in Cortex-M4 processors provides a wide range of instructions:

- General data processing, including hardware divide instructions
- Memory access instructions supporting 8-bit, 16-bit, 32-bit, and 64-bit data, as well as instructions for transferring multiple 32-bit data
- Instructions for bit field processing
- Multiply Accumulate (MAC) and saturate instructions
- Instructions for branches, conditional branches and function calls
- Instructions for system control, OS support, etc.

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)

Operating states:

The processor can operate in one of two operating states:

- Thumb state. This is normal execution running 16-bit and 32-bit halfword aligned Thumb instructions.
- Debug State. This is the state when the processor is in halting debug. The debug state is used for debugging operations only. This state is entered by a halt request from the debugger.

Operating modes:

The processor knows whether it is running in the foreground (i.e. the main program) or in the background (i.e., an interrupt service routine). ARM processors define the foreground as the Thread mode and the background as the Handler mode and support these two modes of operation:

- The processor enters Thread mode on Reset, or as a result of an exception return. Privileged and Unprivileged code can run in Thread mode.
- The processor enters Handler mode as a result of an interrupt (exception). All code is privileged in Handler mode.

Note: Code can execute as privileged or unprivileged. Unprivileged execution limits or excludes access to some resources. Privileged execution has access to all resources.

CHAPTER 2

Programming model

To be able to program a MCU, assembly language programmer should know in depth the following features: CPU registers, instruction set (operations, operands, data sizes and types), addressing modes, memory organization.

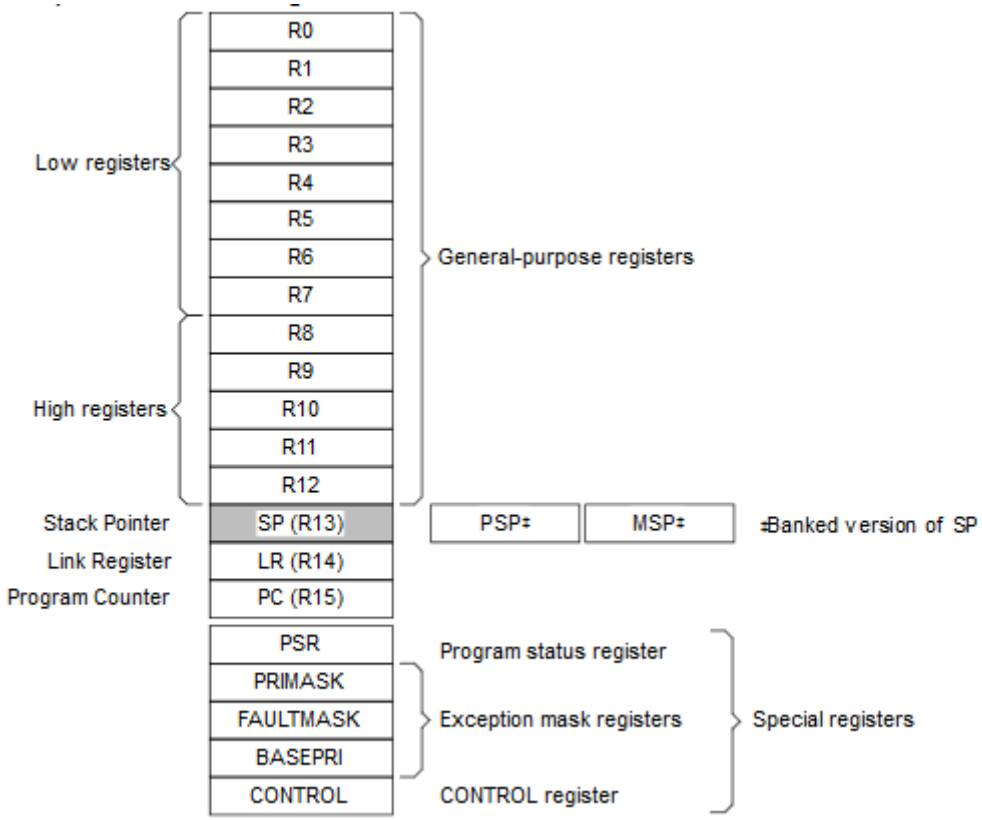
1. Registers

Each data processing instruction specifies the operation required, the source register(s), and the destination register(s) if applicable. In the ARM architecture, if data in memory is to be processed, it has to be loaded from the memory to registers in the register bank, processed inside the processor, and then written back to the memory, if needed. This is called a "load-store architecture." By having a sufficient number of registers in the register bank, this arrangement is easy to use, and allows efficient program code to be generated using C compilers. For instance, a number of data variables can be stored in the register bank for a short period of time while other data processing takes place, without the need to be updated to the system memory and read back every time they are used.

Cortex-M4 has the following 32-bit registers:

- 13 general-purpose registers, r0-r12
- Stack Pointer (SP) alias of banked registers, SP-process (PSP) and SP-main (MSP), r13
- Link Register (LR), r14
- Program Counter (PC), r15
- Special registers
    - Special-purpose Program Status Registers, (xPSR) (Application PSR, Exacution PSR, Interrupt PSR))
    - Exception mask registers (PRIMASK, FAULTMASK, BASEPRI)
    - Control register

Registers are arranged in partially overlapping banks, with the current processor mode controlling which bank is available. Each mode can access a particular set of r0-r12 registers, a particular r13 (the stack pointer, sp) and r14 (the link register, lr), the program counter, r15 (pc) and the current program status register, cpsr.



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. The initial values of R0 to R12 are undefined.

R13, stack pointer (SP)

A stack is a specialized memory which stores data from the top down. As new requests come in, they "push down" the older ones. The most recently entered request always resides at the top of the stack, and the program always takes requests from the top. R13 is used for accessing the stack memory via PUSH and POP operations. Physically there are two different Stack Pointers: the Main Stack Pointer (MSP) 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). The PSP can only be used in Thread Mode. The selection of Stack Pointer is determined by a special register called CONTROL. For 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.

R14, link register (LR)

R14 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. During exception handling, the LR is also updated automatically to a special value.

R15, program counter (PC)
R15 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 ARM7TDM 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.

Special registers
These registers contain the processor status and define the operation states and interrupt/exception masking.

The Program Status Register is composed of three status registers that can also be accessed as one combined register,

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

	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						

The APSR contains the current state of the condition flags from previous instruction executions.

Bits	Name	Function
[31]	N	Negative flag
[30]	Z	Zero flag
[29]	C	Carry or borrow flag
[28]	V	Overflow flag
[27]	Q	DSP overflow and saturation flag
[26:20]	-	Reserved
[19:16]	GE[3:0]	Greater than or Equal flags.
[15:0]	-	Reserved

there are four GE flags, one for each byte.

The IPSR contains the exception type number of the current *Interrupt Service Routine* (ISR).

Bits	Name	Function
[31:9]	-	Reserved
[8:0]	ISR_NUMBER	This is the number of the current exception: 0 = Thread mode 1 = Reserved 2 = NMI 3 = HardFault 4 = MemManage 5 = BusFault 6 = UsageFault 7-10 = Reserved 11 = SVCall 12 = Reserved for Debug 13 = Reserved 14 = PendSV 15 = SysTick 16 = IRQ0. . . .

EPSR is the Execution Program Status Register:

- T bit = 1 if CPU in “Thumb mode” (always for Cortex-M4), 0 in “ARM mode”
- IT field
    - If/Then block information
    - ICI field– Interruptible-Continuable Instruction information

The PRIMASK, FAULTMASK, and BASEPRI registers are all used for exception or interrupt masking. Each exception (including interrupts) has a priority level where a smaller number is a higher priority and a larger number is a lower priority. These special registers are used to mask exceptions based on priority levels. They can only be accessed in the privileged access level (in unprivileged state writes to these registers are ignored and reads return zero). By default, they are all zero, which means the masking (disabling of exception/interrupt) is not active. The PRIMASK register is a 1-bit wide interrupt mask register. When set, it blocks all exceptions (including interrupts) apart from the Non-Maskable Interrupt (NMI) and the HardFault exception. The most common usage for PRIMASK is to disable all interrupts for a time critical process. After the time critical process is completed, the PRIMASK needs to be cleared to re-enable interrupts. The FAULTMASK register is very similar to PRIMASK, but it also blocks the HardFault exception. The BASPRI register defines the priority of the executing software. It prevents interrupts with lower or equal priority.

The CONTROL register controls the stack used and the privilege level for software execution when the processor is in Thread mode and, if implemented, indicates whether the FPU state is active.

Bits	Name	Function
[31:3]	-	Reserved.
[2]	FPCA	When floating-point is implemented this bit indicates whether context floating-point is currently active: 0 = no floating-point context active 1 = floating-point context active. The Cortex-M4 uses this bit to determine whether to preserve floating-point state when processing an exception.
[1]	SPSEL	Defines the currently active stack pointer: In Handler mode this bit reads as zero and ignores writes. The Cortex-M4 updates this bit automatically on exception return: 0 = MSP is the current stack pointer 1 = PSP is the current stack pointer.
[0]	nPRIV	Defines the Thread mode privilege level: 0 = privileged 1 = unprivileged.

2. ARM instruction format

Assembly language instructions may have four fields.

Label **mnemonic {Rd}, {Rn}, {operand2} ; Comments**
(opcode)

```
Loc     ADD   R3,   R2,   R1        ;R3 = R2+R1
        ADD   R3,   R2,   #5        ;R3 = R2+5
        LDR   R3,   [R2]            ;R3 = value pointed by R2
```

- The “label” is used as a reference to an address location. It is optional; some instructions might have a label in front of them so that the address of the instruction can be obtained by using the label. Labels can also be used to reference data addresses.
- Mnemonic represents the operation to be performed.
- The number of operands varies, depending on each specific instruction. Some instructions have no operands at all.
    - If exists, Rd is typically the destination register
    - If exists, Rn is typically the source register.
        - operand2 is the flexible second operand that can be either a register (Rm), shifted register or a constant (immed_8r), i.e.
            - immed_8r: is an expression evaluating to a numeric constant.The constant must correspond to an 8-bit pattern rotated by an even number of bits within a 32-bit word
            - Rm: is the ARM register holding the data for the second operand. The bit pattern in the register can be shifted or rotated in various ways.
                - Shift: is an optional shift to be applied to Rm such as
                    - ASR n: arithmetic shift right n bits. n =1.. 32. (A shift amount of 32 is encoded as shift_imm == 0. Other shift amounts are encoded directly.)

        - LSL n: logical shift left n bits. n = 0..31. (Encoded directly in the shift_imm field.)
- Everything after the semicolon is a comment, which is ignored by the assembler.

2.1. Data type supported

- Integer ALU operations are performed only on 32-bit data
    - Signed or unsigned integers
- Data sizes in memory:
    - Byte (8-bit), Half-word (16-bit), Word (32-bit), Double Word (64-bit)

{type}	*Data type*	*Meaning*
	32-bit word	0 to 4,294,967,295 or -2,147,483,648 to +2,147,483,647
B	Unsigned 8-bit byte	0 to 255, Zero pad to 32 bits on load
SB	Signed 8-bit byte	-128 to +127, Sign extend to 32 bits on load
H	Unsigned 16-bit halfword	0 to 65535, Zero pad to 32 bits on load
SH	Signed 16-bit halfword	-32768 to +32768, Sign extend to 32 bits on load
D	64-bit data	Uses two registers

-
- Bytes/half-words are converted to 32-bits when moved into a register
    - Signed numbers –extend sign bit to upper bits of a 32-bit register
    - Unsigned numbers –fill upper bits of a 32-bit register with 0's
    - Examples:
        - 255 (unsigned byte) 0xFF=>0x000000FF (fill upper 24 bits with 0)
        - -1 (signed byte) 0xFF=>0xFFFFFFFF (fill upper 24 bits with sign bit 1)
        - +1 (signed byte) 0x01=>0x00000001 (fill upper 24 bits with sign bit 0)
        - -32768 (signed half-word) 0x8000=>0xFFFF8000 (sign bit = 1)
        - 32768 (unsigned half-word) 0x8000=>0x00008000
        - +32767 (signed half-word) 0x7FFF=>0x00007FFF (sign bit = 0)
- Cortex-M4F also supports single and double-precision IEEE floating-point data

3. Addressing modes

The addressing mode is the format the instruction uses to specify the memory location to read or write data. Generic types of addressing modes are immediate, direct, indirect or indexed or relative. ARM does not use direct addressing mode since 32-bit address can not be included in a 32-bit inctruction.

3.1. Immediate addressing mode

The data itself is contained in the instruction. Once the instruction is fetched no additional memory access cycles are needed to get the data. It is only used to get, load or read data. Is never used with an instruction that stores to memory.

```
Opcode      Rd, #constant

MOV         R0, #100      ; R0=100
ADD         R0, #0x64     ; R0=R0+100
```

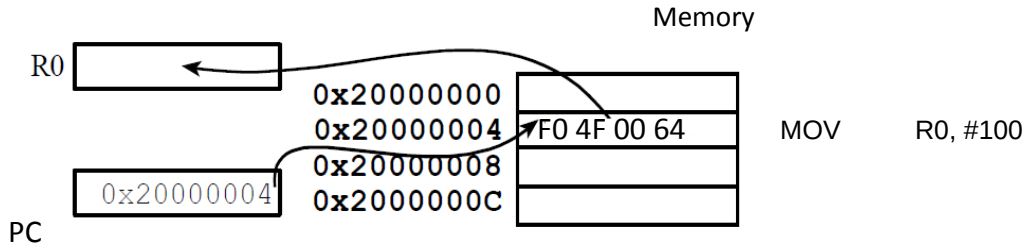
where constant can be (X and Y are hexadecimal digits):

- Any constant that can be produced by shifting an 8-bit value left by any number of bits within a 32-bit word.
- Any constant of the form 0x00XY00XY.
- Any constant of the form 0xXY00XY00.
- Any constant of the form 0xXYXYXYXY.

- For decimal representation no prefixes

Ex:

```
        MOV   R0, #100        ; R0=100
; the location of the instruction in the memory is 0x20000004
; machine code of the instruction is 0xF04F0064
```



After the execution of the instruction: R0=0x64

3.2. Indexed addresing mode

The data is in memory and a register will contain a pointer to the data. Once the instruction is fetched, one or more additional memory access cycles are required to read or write the data.

- Can include an offset from the index address
- Can include updating index register with offset (pre- or post- access)

Opcode R1,[R2, optional offset, optional shift], optional offset

Several forms of indexed addressing:

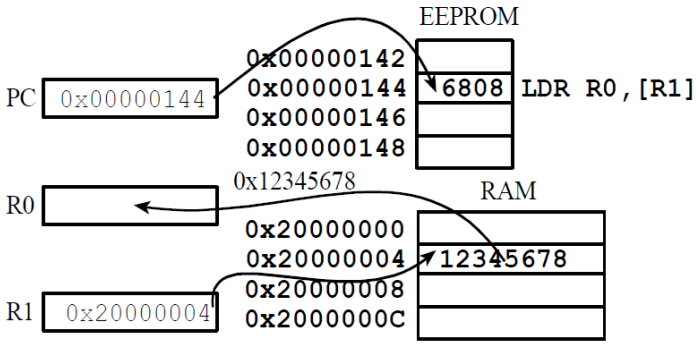
```
LDR{type}   Rd,[Rn]              ;load memory at [Rn] to Rd
STR{type}   Rt,[Rn]              ;store Rt to memory at [Rn]
LDR{type}   Rd,[Rn, #n]          ;load memory at [Rn+n] to Rd, Rn unchanged
STR{type}   Rt,[Rn, #n]          ;store Rt to memory [Rn+n], Rn unchanged
LDR{type}   Rd,[Rn,Rm,LSL #n1]   ;load [Rn+Rm<<n1] to Rd, Rn unchanged
STR{type}   Rt,[Rn,Rm,LSL #n1]   ;store Rt to [Rn+Rm<<n1], Rn unchanged
LDR{type}   Rd,[Rn, #n]!         ;load memory at [Rn+n] to Rd, update Rn (Rn=Rn+n)
STR{type}   Rt,[Rn, #n]!         ;store Rt to memory [Rn+n], update Rn  (Rn=Rn+n)
LDR{type}   Rd,[Rn], #n          ;load memory at [Rn] to Rd, update Rn  (Rn=Rn+n)
STR{type}   Rt,[Rn], #n          ;store Rt to memory [Rn], update Rn  (Rn=Rn+n)
```

{type}: none is word, B is byte, H is halfword.

Offset range n: -255 to 255, n1: 0 to 3

Ex: LDR R0,[R1] ; R0 = value pointed to by R1

```
; the location of the instruction in the memory is 0x00000144
; machine code of the instruction is 0x6808
; R1 = 0x20000004
```



After the execution, R0= 0x12345678

Pre-Index: LDR r1, [r0, #4]

After the execution, R1= 0x88796A5B; R0=0x20008000.

Post-Index: LDR r1, [r0], #4

After the execution, R1= 0x4C3D2E1F; R0=0x20008004.

3.3. PC-relative addressing mode

It is indexed addressing using PC as the pointer. It is represented in the instruction as the PC value plus or minus a numeric offset. The assembler calculates the required offset from

the label and the address of the current instruction. If the offset is too big, the assembler produces an error.

This addressing mode is used for branching, calling functions, etc.

Ex: **PC**

```
0x00001000          ADR R5, label        ;load address to R5
        .
        .
        .
0x0000100C  label   LDR   R0,[R1]
```

Here R5=PC+$offset, where **$**offset is from PC to data (pointed by label), in this example $0C.

ADR instruction generates an address, within a certain range, without performing a data load, i.e. Can only load values that are addresses (relative to PC).

The label used with ADR must be within the same code section. Values of label must be within the range -4095 to 4095 from the address in the PC. In a 32 bit instruction this range could be specified. In a 16 bit instruction offset is 8 bits. The assembler faults references to labels that are out of range in the same section.

PC-relative addressing mode is also used in pseudo-instructions. The assembler uses ADR.W as the pseudo-instruction. This is explained below.

3.4. Pseudo Instructions

The ARM assembler supports a number of pseudo-instructions (pseudo-operation, pseudo-op) that are translated into the appropriate combination of one or more actual ARM or Thumb instructions at assembly time.

In loading data

- Load any constant using LDR as a pseudo instruction.
- Load a 32-bit constant (data, address, etc .) into a register. Cannot embed 32-bit value in a 32-bit instruction. Use a “pseudo-op” to load the constant: Either LDR with relative addressing or MOV32.

Ex:

Source Program	Debug Disassembly
MOV32 r0,#0x12345678	MOVW r0,#0x5678
	MOVT r0,#0x1234

Ex:

LDR Rd,=val is a pseudo-op that gets assembled into either a **MOV Rd,#val** (if an appropriate immediate constant can be found) or a constant in ROM with **LDR Rd,[pc,#offs]** using PC-relative addressing mode. “val” can be any constant (number/value) or address

Ex:

Source Program		Debug Disassembly
LDR r3,=0x55	=>	MOV r3,#0x55

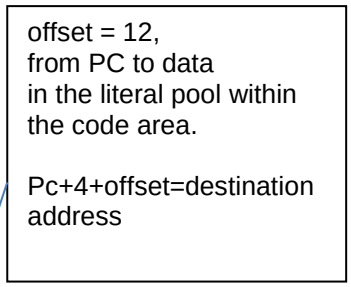
Ex:

LDR r3,=0x12345678

This pseudo-op requires two 32-bit words:

- One word for the LDR instruction
- One word for the 32-bit constant in the literal pool

32-bit constant is placed in the “literal pool”.
Literal pool: set of constants stored after program in code area.
Constant loaded from literal pool address [PC,#offset]



PC	Source Program		Debug Disassembly	
0x0000050C	LDR	r3,=0x12345678	LDR r3,[PC,#offset]	;occupies one word
		.....		
0x0000051C	dcw 0x5678		;in literal pool following the code	
0x0000051E	dcw 0x1234			

Flash Memory content:

0x50C	4B
0x50D	03
0x50E	.
.	.
0x51C	78
0x51D	56
0x51E	34
0x51F	12

LDR R3,[PC,#offset]

Please note that, usually the assembler groups various literal data (e.g., DCD 0x12345678 in the above example) together into data blocks called literal pools. Since the value of the offset in the LDR instruction is limited, a program will often need a number of literal pools so that the LDR instruction can access the literal data. Therefore we need to insert assembler directives like LTORG (or .pool) to tell the assembler where it can insert literal pools. Otherwise the assembler will try to put all the literal data after the end of the program code, which might be too far away for the LDR instruction to access it.

Ex:

```
PC                  Source Program          Debug Disassembly
0x0000028C  label   LDR    r3,=label        LDR r3,[PC,#offset]

                           ...              ;offset=164=0xA4

                           ...              ;destination address= 0x28C+0xA4=0x330

0x00000330          dcw    0x0000
0x00000334          dcw    0x028C
```

Note about instruction width:
There are many instructions that can generate either a 16-bit encoding or a 32-bit encoding depending on the operands and destination register specified. For some of these instructions, you can force a specific instruction size by using an instruction width suffix. The .W suffix forces a 32-bit instruction encoding. The .N suffix forces a 16-bit instruction encoding.
If you specify an instruction width suffix and the assembler cannot generate an instruction encoding of the requested width, it generates an error.
In some cases it might be necessary to specify the .W suffix, for example if the operand is the label of an instruction or literal data, as in the case of branch instructions. This is because the assembler might not automatically generate the right size encoding.

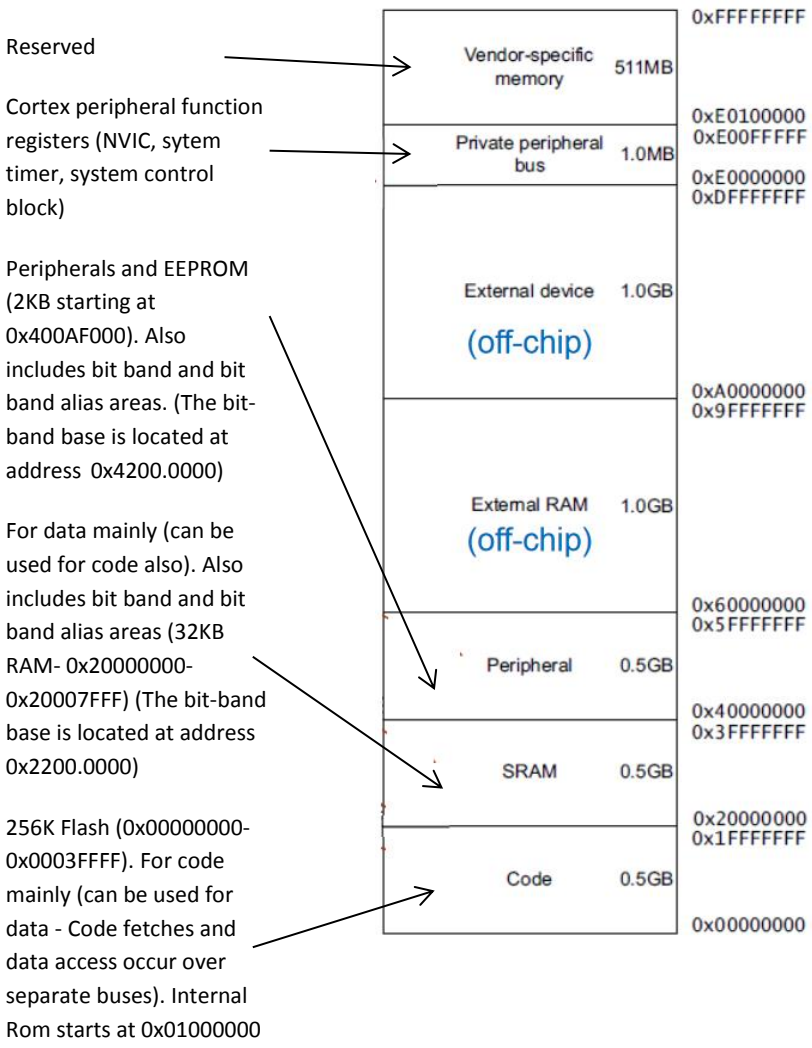
Ex: LDR, LDR.W

4. Memory and Bus Structure

Cortex-M4 uses a memory mapped architecture. 4 GB memory is divided into 8x512MB segments.

Memory is needed as

- Code memory (normally read-only memory)
  - Program instructions
  - Constant data
- Data memory (normally read/write memory – RAM)
  - Variable data/operands
- Stack (located in data memory)
  - Special Last-In/First-Out (LIFO) data structure
    - Save information temporarily and retrieve it later
    - Save return addresses for subroutines and interrupt/exception handlers
    - Data to be passed to/from a subroutine/function can be saved
  - Stack Pointer register (r13/sp) points to last item placed on the stack
- Peripheral addresses
  - Used to access registers in “peripheral functions” (timers, ADCs, communication modules, etc.) outside the CPU



Memory organization - Memory map

The Cortex-M3 and M4 processors themselves do not include memories (i.e.,they do not have program memory, SRAM, or cache). Instead, they come with a generic on-chip bus interface, so microcontroller vendors can add their own memory system to their design. In this way, different microcontroller products can have different memory configurations, different memory sizes and types, and different peripherals.

The bus interfaces on the Cortex-M processors are 32-bit, and based on the Advanced Microcontroller Bus Architecture (AMBA) standard. AMBA contains a collection of several bus protocol specifications. The AMBA specifications can be downloaded from the ARM website, and any silicon designers can freely use these protocol standards. The main bus interface protocol used by the Cortex-M3 and M4 processors is the AHB Lite (Advanced High-performance Bus), which is used in program memory and system bus interfaces. The Bus Matrix partitions memory access via the AHB and PPB (Private Peripheral Bus) buses.

The processor contains four external *Advanced High-performance Bus* (AHB)-Lite bus interfaces:
ICode memory interface: Instruction fetches from Code memory space, 0x00000000 to 0x1FFFFFFF, are performed over this 32-bit AHB-Lite bus.

The Debugger cannot access this interface. All fetches are word-wide. The number of instructions fetched per word depends on the code running and the alignment of the code in memory.

DCode memory interface: Data and debug accesses to Code memory space, 0x00000000 to 0x1FFFFFFF, are performed over this 32-bit AHB-Lite bus. Core data accesses have a higher priority than debug accesses on this bus. This means that debug accesses are waited until core accesses have completed when there are simultaneous core and debug access to this bus.

Control logic in this interface converts unaligned data and debug accesses into two or three aligned accesses, depending on the size and alignment of the unaligned access. This stalls any subsequent data or debug access until the unaligned access has completed.

Note that DCode has a higher priority than ICode.

System interface: Instruction fetches, and data and debug accesses, to address ranges 0x20000000 to 0xDFFFFFFF and 0xE0100000 to 0xFFFFFFFF are performed over this 32-bit AHB-Lite bus.

For simultaneous accesses to this bus, the arbitration order in decreasing priority is:

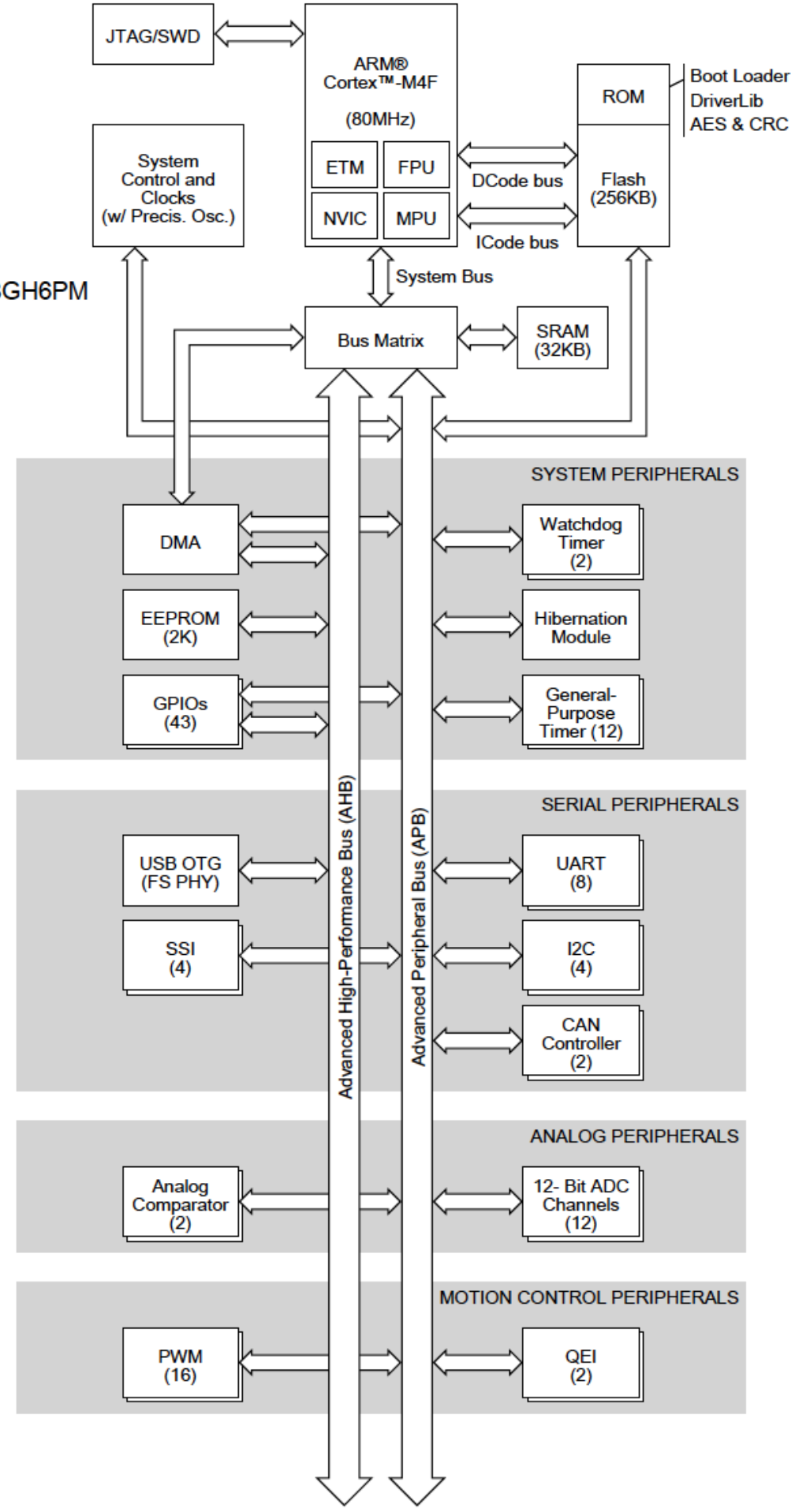
- data accesses
- instruction and vector fetches
- debug.

The system bus interface contains control logic to handle unaligned accesses, FPB remapped accesses, bit-band accesses, and pipelined instruction fetches.

Private Peripheral Bus (PPB): Data and debug accesses to external PPB space, 0xE0040000 to 0xE00FFFFF, are performed over this 32-bit *Advanced Peripheral Bus* (APB) bus. The *Trace Port Interface Unit* (TPIU) and vendor specific peripherals are on this bus.

Core data accesses have higher priority than debug accesses, so debug accesses are waited until core accesses have completed when there are simultaneous core and debug access to this bus. Only the address bits necessary to decode the External PPB space are supported on this interface.

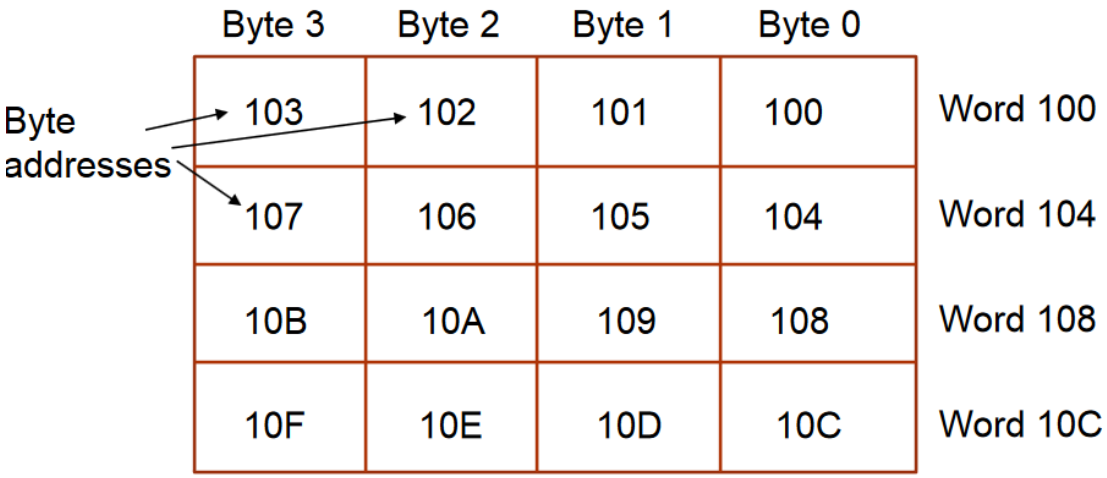
TM4C123GH6PM



Physical memory organization

Physical memory may be organized as N bytes per addressable word

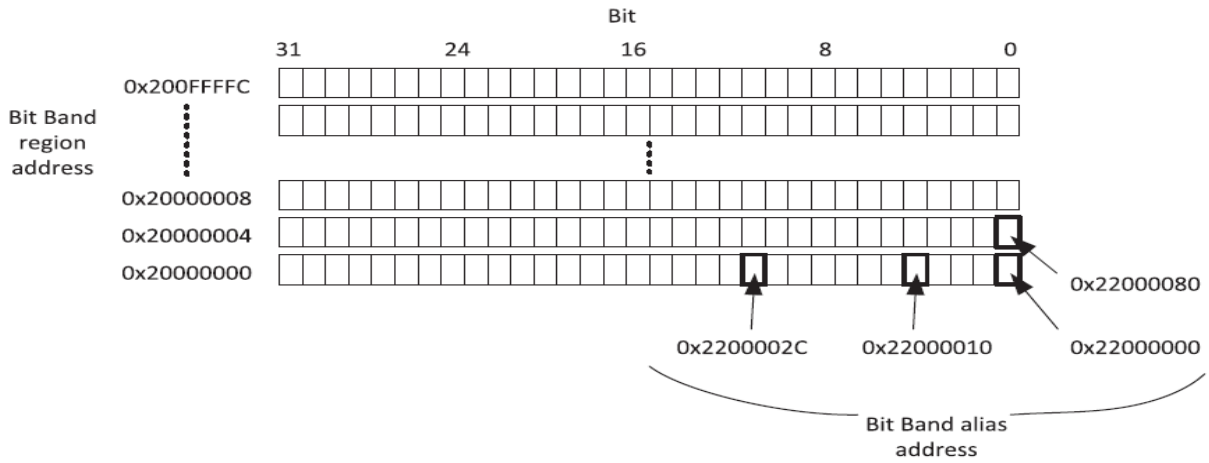
- ARM memories normally 4-bytes wide
- "Align" 32-bit data to a Word boundary (address that is a multiple of 4)
    - All bytes of a word must be accessible with one memory read/write



Bit banding

Bit-banding is a term that ARM uses to describe a feature that is available on the Cortex M3 and M4 CPU cores. Basically, the device takes a region of memory (the Bit-band region) and maps each bit in that region to an entire word in a second memory region (the Bit-band Alias Region). The benefit of Bit-banding is that a write to a word in the alias region performs a write to the corresponding bit in the Bit-band region. Also, reading a word in the alias region will return the value of the corresponding bit in the Bit-band region. This is especially useful for interacting with peripheral registers where it is often necessary to set and clear individual bits.

The figure below shows the bit-band alias address for individual bits in the bit-band region.



Ex: To set bit 0 in word data in address 0x20000000, instead of using three instructions to read the data, set the bit, and then write back the result, this task can be carried out by a single instruction.

Without bit-band

```
LDR  r0,=0x20000000          ; set-up address
LDR  r1,[r0]                 ; read
ORR.W  r1, #0x1              ; modify bit
STR  r1,[r0]                 ; write back
```

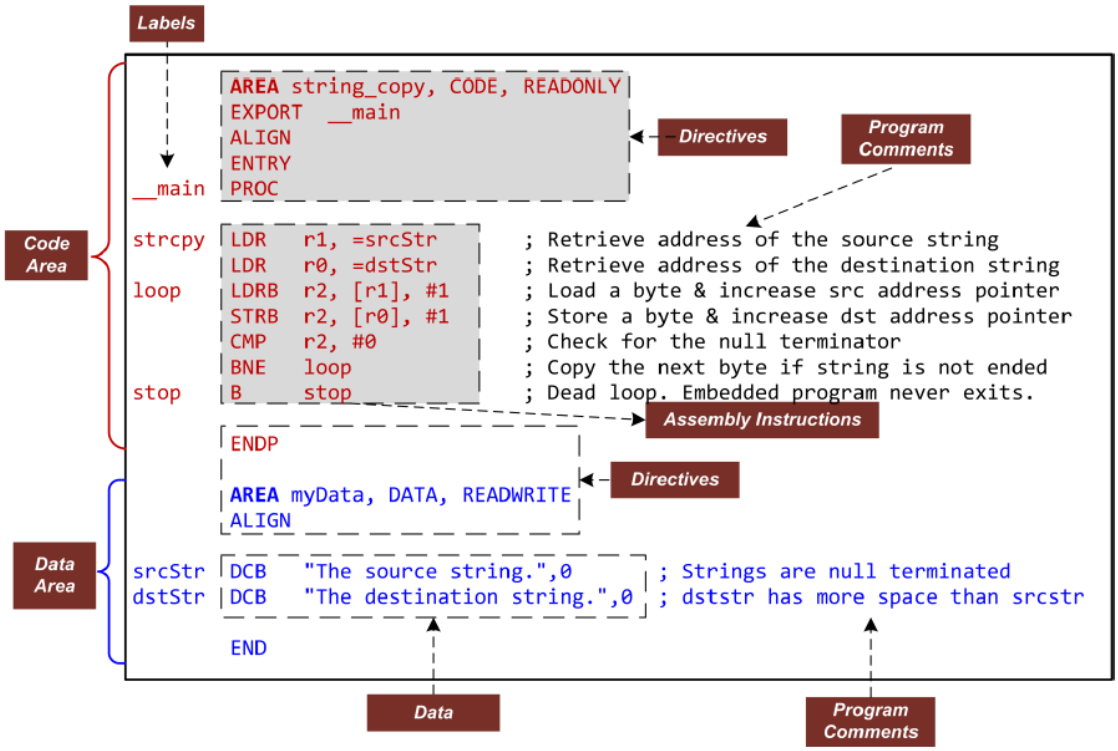
With bit-band

```
LDR     r0,=0x22000000       ; set-up address
MOV     r1,#1                ; modify bit
STR     R1,[R0]
```

5. Microprocessor directives

An assembler supports statements to define data, to organise segments, to control procedure, to define macros. It consists of two types of statements: instructions and directives. The instructions are translated to the machine code by the assembler whereas directives are not translated to the machine codes. The directives that you can use are (key directives are in bold):

AREA	Make a new block of data or code
ENTRY	Declare an entry point where the program execution starts
ALIGN	Align data or code to a particular memory boundary
DCB	Allocate one or more bytes (8 bits) of data
DCW	Allocate one or more half-words (16 bits) of data
DCD	Allocate one or more words (32 bits) of data
SPACE	Allocate a zeroed block of memory with a particular size
FILL	Allocate a block of memory and fill with a given value.
EQU	Give a symbol name to a numeric constant
RN	Give a symbol name to a register
EXPORT	Declare a symbol and make it referable by other source files
IMPORT	Provide a symbol defined outside the current source file
INCLUDE/GET	Include a separate source file within the current source file
PROC	Declare the start of a procedure
ENDP	Designate the end of a procedure
END	Designate the end of a source file



AREA directive indicates to the assembler the start of a new data or code section.

- Areas are the basic independent and indivisible unit processed by the **linker**.
- Each area is identified by a name. Areas within the same source file cannot share the same name.
- An assembly program must have at least one code area.
- A code area can only be read (read-only). A data area may be read from and written to (read-write).

ENTRY directive marks the first instruction to be executed within an application program

- There must be exactly one ENTRY directive in an *application program*, no matter how many source files the application has.

END directive indicates the end of a source file. Any lines of information after this are ignored by the assembler.

- Each assembly source file must end with this directive.

PROC and ENDP are to mark the start and end of a function (also called subroutine or procedure).

- A single source file can contain multiple functions, with each of them defined by a pair of PROC and ENDP.
- PROC and ENDP cannot be nested. We cannot define a function within another function – more on this later.

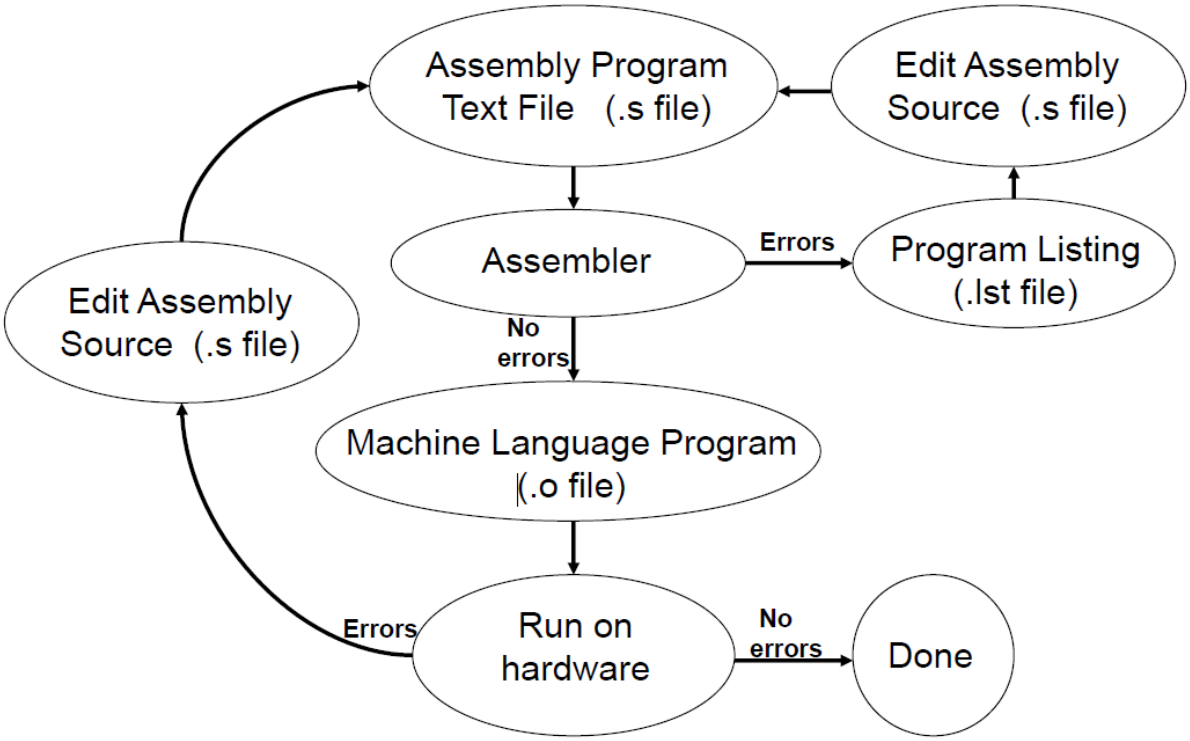
EXPORT declares a symbol and makes this symbol visible to the linker.

IMPORT gives the assembler a symbol that is not defined locally in the current assembly file but which must be defined in another file to be used by the linker.

DCB	Define Constant Byte	Reserve 8-bit values
DCW	Define Constant Half-word	Reserve 16-bit values
DCD	Define Constant Word	Reserve 32-bit values (word=32 bits)
DCQ	Define Constant	Reserve 64-bit values
SPACE	Define Zeroed Bytes	Reserve a number of zeroed bytes
FILL	Define Initialized Bytes	Reserve and fill each byte with a value

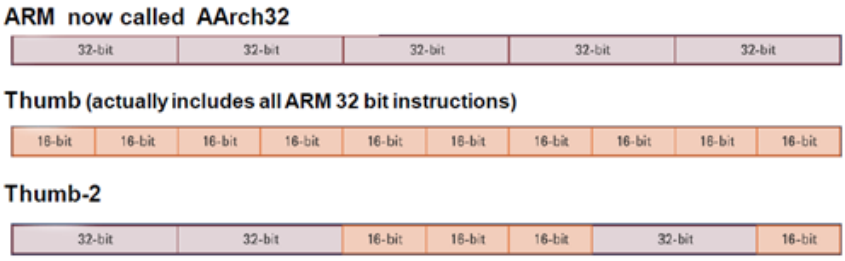
With ARM, in many cases instructions and data must begin on 2, 4, or 8 byte boundaries. The **ALIGN** directive instructs the Assembler to cause this to happen.

5.1. Assembly process



CHAPTER 3

Assembly Programming Details



Thumb-2 is a major enhancement to the Thumb Instruction Set Architecture (ISA). It introduces 32-bit instructions that can be intermixed freely with the older 16-bit Thumb instructions. These new 32-bit instructions cover almost all the functionality of the ARM instruction set.

Differences

- Thumb-2 delivers overall code density comparable with Thumb, together with the performance levels associated with the ARM ISA. Before Thumb-2, developers had to choose between Thumb for size, or ARM for performance.
- Thumb-2 enforces 16-bit alignment on all instructions. This means that 32-bit instructions are treated as two halfwords, hw1 and hw2, with hw1 at the lower address. In instruction encoding diagrams, hw1 is shown to the left of hw2. This results in the encoding diagrams reading more naturally, and in a close correspondence between the ARM and Thumb encoding diagrams in some cases, particularly coprocessor instructions. However, it also makes the byte order of a 32-bit Thumb instruction differ from that of an ARM instruction.

ARM 32-bit instruction order in memory

32-bit ARM instruction							
31	24	23	16	15	8	7	0
Byte at Address A+3		Byte at Address A+2		Byte at Address A+1		Byte at Address A	

Thumb 32-bit instruction order in memory

32-bit Thumb instruction, hw1				32-bit Thumb instruction, hw2			
31	24	23	16	15	8	7	0
Byte at Address A+1		Byte at Address A		Byte at Address A+3		Byte at Address A+2	

- Thumb2 instructions are either 16-bit or 32-bit. Bits[15:11] of the halfword that the PC points to determine whether it is a 16-bit instruction, or whether the following halfword is the second part of a 32-bit instruction. A 16-bit instruction can be converted to 32-bit by adding ".W" to the opcode. ".N" forces a 16-bit instruction decoding. See the table below and "Instruction set overview" given below for the Thumb 2 instruction set coding.

hw1[15:11]	Function
0b11100	Thumb 16-bit unconditional branch instruction, defined in all Thumb architectures.
0b111xx	Thumb 32-bit instructions, defined in Thumb-2.
0bxxxxx	Thumb 16-bit instructions.

1. Thumb Instruction Set

Most instructions are 16 bits long, some are 32 bits. Most 16-bit instructions can only access low registers (R0-R7), but some can access high registers (R8-R15). Instructions are half-word aligned.

1.1. Instruction Types

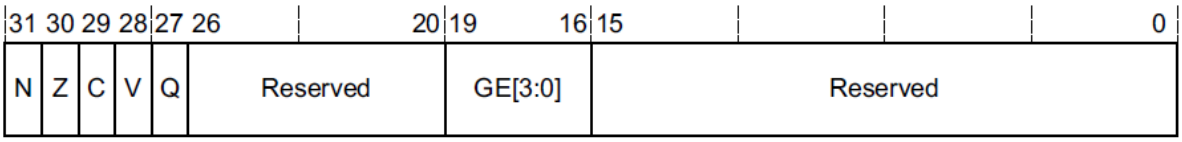
- Data movement operations
  - memory-to-register and register-to-memory
    - includes different memory “addressing” options
    - “memory” includes peripheral function registers
  - register-to-register
  - constant-to-register (or to memory in some CPUs)
- Arithmetic operations
  - add/subtract/multiply/divide
  - multi-precision operations (more than 32 bits)
- Logical operations
  - and/or/exclusive-or/complement (between operand bits)
  - shift/rotate
  - bit test/set/reset
- Flow control operations
  - branch to a location (conditionally or unconditionally)
  - branch to a subroutine/function
  - return from a subroutine/function
- Miscellaneous
  - Wait for events
  - Interrupts
  - Others

A summary of instruction types are given below with their mnemonics.

Instruction Type	Instructions
Move	MOV
Load/Store	LDR, LDRB, LDRH, LDRSH, LDRSB, LDM, STR, STRB, STRH, STM
Add, Subtract, Multiply	ADD, ADDS, ADCS, ADR, SUB, SUBS, SBCS, RSBS, MULS
Compare	CMP, CMN
Logical	ANDS, EORS, ORRS, BICS, MVNS, TST
Shift and Rotate	LSLS, LSRS, ASRS, RORS
Stack	PUSH, POP
Conditional branch	IT, B, BL, B{cond}, BX, BLX
Extend	SXTH, SXTB, UXTH, UXTB
Reverse	REV, REV16, REVSH
Processor State	SVC, CPSID, CPSIE, BKPT
No Operation	NOP
Hint	SEV, WFE, WFI

APSR register (Application Program Status Register):

APSR is an 32 bit register, where most significant 5 bits hold some flags (N, Z, C, V, Q) which could be updated by the latest instruction that has been executed by the CPU.



Bits	Name	Function
[31]	N	Negative flag
[30]	Z	Zero flag
[29]	C	Carry or borrow flag
[28]	V	Overflow flag
[27]	Q	DSP overflow and saturation flag
[26:20]	-	Reserved
[19:16]	GE[3:0]	Greater than or Equal flags.
[15:0]	-	Reserved

- "S" suffix in an instruction mnemonic (some examples are given below) indicates that the instruction updates APSR
  - ADD vs. ADDS
  - ADC vs. ADCS
  - SUB vs. SUBS
  - MOV vs. MOVS
- There are some instructions that update the APSR without explicitly adding S to them since their basic functions are to update the APSR
  - CMP
  - TST

Carry occurs

- if the result of an addition is greater than or equal to 2^{32}
- if the result of a subtraction is positive (not negative!) or zero
- as the result of an inline barrel shifter operation in a move or logical instruction.

Overflow occurs when the sign of the result, in bit[31], does not match the sign of the result had the operation been performed at infinite precision, for example:

- if adding two negative values results in a positive value
- if adding two positive values results in a negative value
- if subtracting a positive value from a negative value generates a positive value
- if subtracting a negative value from a positive value generates a negative value.

1.2. Instruction set overview

Machine code encoding of instructions are given below.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Shift by immediate, move register	0	0	0	opcode [1]		imm5					Rm			Rd		
Add/subtract register	0	0	0	1	1	0	opc		Rm		Rn			Rd		
Add/subtract immediate	0	0	0	1	1	1	opc		imm3		Rn			Rd		
Add/subtract/compare/move immediate	0	0	1	opcode		Rdn			imm8							
Data-processing register	0	1	0	0	0	0	opcode				Rm			Rdn		
Special data processing	0	1	0	0	0	1	opcode [1]		DN		Rm			Rdn		
Branch/exchange instruction set	0	1	0	0	0	1	1	1	L		Rm			(0)	(0)	(0)
Load from literal pool	0	1	0	0	1	Rd			PC-relative imm8							
Load/store register offset	0	1	0	1	opcode				Rm		Rn			Rd		
Load/store word/byte immediate offset	0	1	1	B	L	imm5					Rn			Rd		
Load/store halfword immediate offset	1	0	0	0	L	imm5					Rn			Rd		
Load from or store to stack	1	0	0	1	L	Rd			SP-relative imm8							
Add to SP or PC	1	0	1	0	SP	Rd			imm8							
Miscellaneous:	1	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x
Load/store multiple	1	1	0	0	L	Rn			register list							
Conditional branch	1	1	0	1	cond [2]				imm8							
Undefined instruction	1	1	0	1	1	1	1	0	x	x	x	x	x	x	x	x
Service (system) call	1	1	0	1	1	1	1	1	imm8							
Unconditional branch	1	1	1	0	0	imm11										
32-bit instruction	1	1	1	0	1	x	x	x	x	x	x	x	x	x	x	x
32-bit instruction	1	1	1	1	x	x	x	x	x	x	x	x	x	x	x	x

[1] opcode != 0b11. [2] cond != 0b111x.

Miscellaneous instructions

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Adjust stack pointer	1	0	1	1	0	0	0	0	opc	imm7						
Sign/zero extend	1	0	1	1	0	0	1	0	opc		Rm			Rd		
Compare and Branch on (Non-)Zero	1	0	1	1	N	0	i	1	imm5					Rn		
Push/pop register list	1	0	1	1	L	1	0	R	register list							
UNPREDICTABLE	1	0	1	1	0	1	1	0	0	1	0	0	x	x	x	x
Set Endianness	1	0	1	1	0	1	1	0	0	1	0	1	E	(0)	(0)	(0)
Change Processor State	1	0	1	1	0	1	1	0	0	1	1	im	0	A	I	F
UNPREDICTABLE	1	0	1	1	0	1	1	0	0	1	1	x	1	x	x	x
Reverse bytes	1	0	1	1	1	0	1	0	opc		Rn			Rd		
Software breakpoint	1	0	1	1	1	1	1	0	imm8							
If-Then instructions	1	0	1	1	1	1	1	1	cond				mask (!= 0b0000)			
NOP-compatible hints	1	0	1	1	1	1	1	1	hint				0	0	0	0

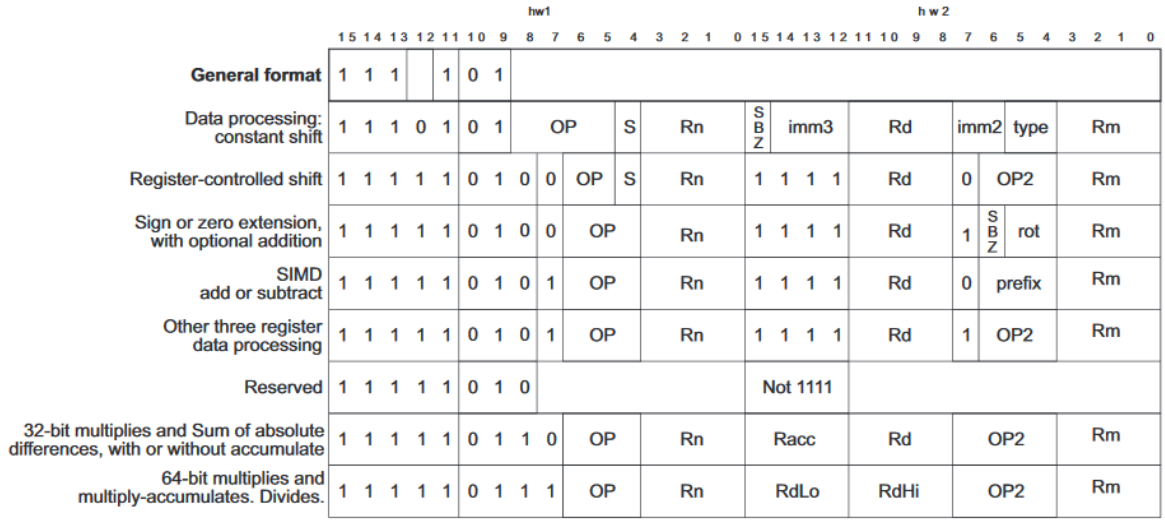
32-bit instructions

	hw1 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	hw2 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data processing: immediate, including bitfield, and saturate	1	1	1	1	0												0															
Data processing, no immediate operand	1	1	1			1	0	1																								
Load and store single data item, memory hints	1	1	1	1	1	0	0																									
Load and Store, Double and Exclusive, and Table Branch	1	1	1	0	1	0	0				1																					
Load and Store Multiple, RFE and SRS	1	1	1	0	1	0	0				0																					
Branches, miscellaneous control	1	1	1	1	0												1															
Coprocessor	1	1	1			1	1	1	1																							

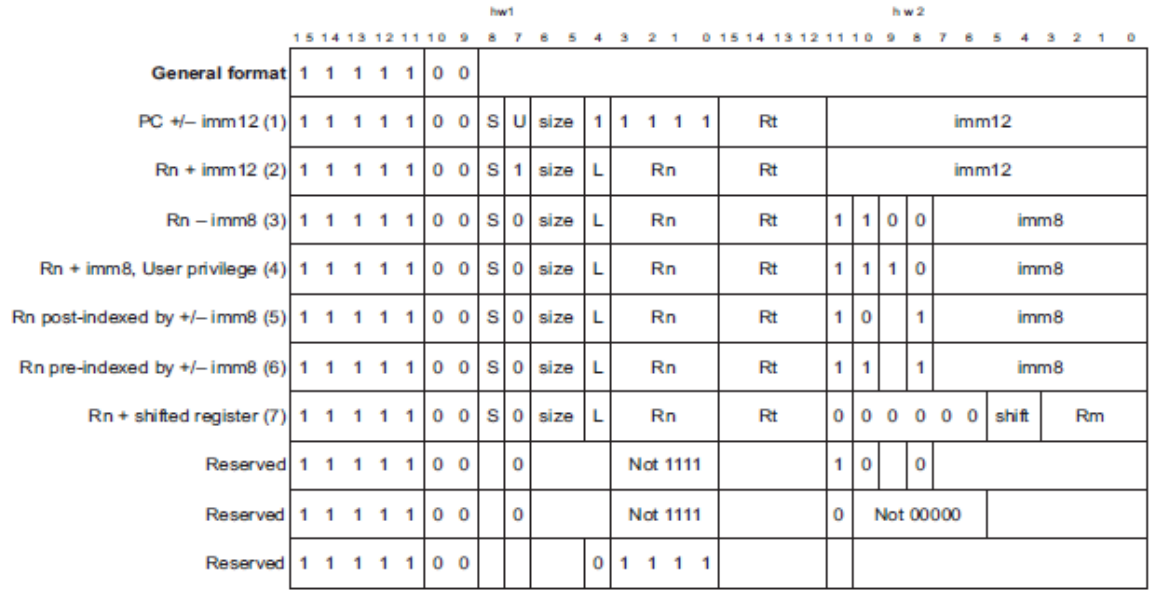
32-bit data processing instructions (immediate)

	hw1 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	hw2 15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
General format	1	1	1	1	0												0															
Data processing, modified 12-bit immediate	1	1	1	1	0	i	0	OP				S	Rn				0	imm3			Rd				imm8							
Add, Subtract, plain 12-bit immediate	1	1	1	1	0	i	1	0	OP		0	OP2			Rn		0	imm3			Rd				imm8							
Move, plain 16-bit immediate	1	1	1	1	0	i	1	0	OP		1	OP2			imm4		0	imm3			Rd				imm8							
Bit field operations, Saturation with shift	1	1	1	1	0	SBZ	1	1	OP			0	Rn				0	imm3			Rd			imm2	SBZ	imm5						
Reserved	1	1	1	1	0		1	1				1					0															

32-bit data processing instructions (non-immediate)



Load-store 32-bit



Some examples of instruction codes are given below:

Assume that the current value of PC is 0x00000290 and the memory content is as follows:

0x00000290	4F
0x00000291	F0
0x00000292	01
0x00000293	03
0x00000294	19

0x00000295	68
0x00000296	59
0x00000297	68
0x00000298	D2
0x00000299	F8
0x0000029a	FF
0x0000029b	10
0x0000029c	53
0x0000029d	F8
0x0000029e	04
0x0000029f	1B

Inst1: Pc Instruction Machine code

0x00000290 mov r3,#1 0xF04F0301

31					16					7 0
11110	0	0	0010	0	1111		0	000	0011	00000001

Move immediate (12 bit) Rd=3 constant=1

Inst2: 0x00000294 ldr r1,[r3] 0x6819

15					0
011	0	1	00000	011	001

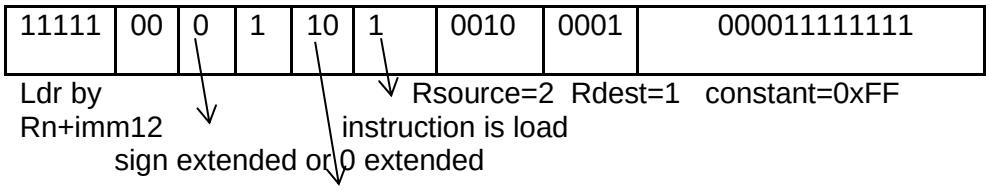
Ldr offset Rn=3 Rdest=1

Inst3: 0x00000296 ldr r1,[r3,#4] 0x6859

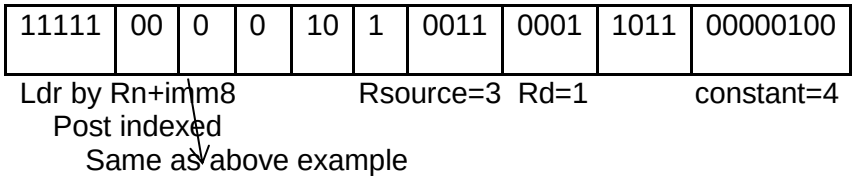
15					0
011	0	1	00001	011	001

Ldr offset=4 Rsource=3 Rdest=1

Inst4: 0x00000298 ldr r1,[r2,#0xFF] 0xF8D210FF



Inst5: 0x0000029c ldr r1,[r3],#4 0xF8531B04



2. Instruction Types

2.1. Data movement operations

ARM is a load/store architecture, so must process data in registers, not memory. General load/store instruction format is as follows:

Load and strore instructions

```
LDR{type}   Rd,[Rn]              ;load memory at [Rn] to Rd
STR{type}   Rt,[Rn]              ;store Rt to memory at [Rn]
LDR{type}   Rd,[Rn, #n]          ;load memory at [Rn+n] to Rd, Rn unchanged
STR{type}   Rt,[Rn, #n]          ;store Rt to memory [Rn+n], Rn unchanged
LDR{type}   Rd,[Rn,Rm,LSL #n1]   ;load [Rn+Rm<<n1] to Rd, Rn unchanged
STR{type}   Rt,[Rn,Rm,LSL #n1]   ;store Rt to [Rn+Rm<<n1], Rn unchanged
LDR{type}   Rd,[Rn, #n]!         ;load memory at [Rn+n] to Rd, update Rn
STR{type}   Rt,[Rn, #n]!         ;store Rt to memory [Rn+n], update Rn
LDR{type}   Rd,[Rn], #n          ;load memory at [Rn] to Rd, update Rn
STR{type}   Rt,[Rn], #n          ;store Rt to memory [Rn], update Rn
```

{type} : none is word; B is byte; H is half-word
Offset range n: -255 to 255, n1: 0 to 3

LDRD, STRD is used for multiple register operations.

Pseudo Instructions

```
LDR     Rd, =value      ; puts address or 32 bit data into Rd
ADR     Rd, LABEL       ; loads Rd with address of LABEL (only valid for
labels defined in a CODE AREA)
```

Move data

```
MOV Rd, #im16    ; puts the number im16 in the range of  0 to 65535 to Rd
MOV R0, R1       ; copy r1 to r0
```

Push-POP

PUSH {reglist} ;push 32-bit values from registers onto stack
POP {reglist} ; pop 32-bit values from stack into registers

- Decrements SP by 4 bytes for each register saved
- Increements SP by 4 bytes for each register restored

Ex:

```
MOV   r1, #64          ; copy 64 (0x00000040) to r1
MOV   r0, r1           ; copy r1 to r0. Now r0=0x00000040, r1 doesn't change
MOVT  r0, #0x1234      ; # -> r0[31:16](move to Top). Now r0=0x12340040
MVN   r0, #0x13        ; copy NOT(0x00000013). r0=0xFFFFFFEC
MOV32 r0, #0x20008000  ; r0=0x20008000
```

Ex: Assume r1=0x76543210 and r0= 0x20008000 and the memory content is as follows.

	Before	After
Memory Address	Memory Data	Memory Data
0x20008007	0x00	0x00
0x20008006	0x00	0x00
0x20008005	0x00	0x00
0x20008004	0x00	0x00
0x20008003	0x00	0x76
0x20008002	0x00	0x54
0x20008001	0x00	0x32
0x20008000	0x00	0x10

```
STR   r1, [r0], #2    ;r1 same, r0=0x20008002

LDR   r1,[r0]         ; r1=0x00007654

LDRH  r1,[r0]         ; r1=0x00007654

LDRB  r1,[r0]         ; r1=0x00000054

MOV   r3,#2;          ; r3=0x00000002

LDRB  r1,[r0,r3]      ; r1=[0x20008002 + 0x2] = 0x00000000

LDRSB r1,[r0]         ; r1=0X00000054

LDR   r1,[r0,r3,LSL #2] ; address=(r0)+(r3 x 4), i.e. r1=[0x20008002 + 0x8]
```

Ex: A simple programming example

```
int p, k;      //signed integer (32-bit) variables
int w[10];     //array of 10 (32-bit) integers
```

```
int *ps;          //32-bit pointer

p = k;             //copy k to p
p = w[k];         //copy  kth element of array w to p
ps = &k;          //ps = address of (pointer to) variable k
p = *ps;          //p=value of the variable pointed by ps (value of k)
```

```
        ldr     r0,=k   ;r0 = address of variable k
        ldr     r1,[r0] ;read value of k from memory and put -> r1
        ldr     r0,=p   ;r0 = address of p
        str     r1,[r0] ;write value in r1 to variable at memory address p

        ldr     r0,=k   ;address of variable k
        ldr     r1,[r0] ;load value of k
        ldr     r0,=w   ;base address of array w
        ldr     r2,[r0,r1,LSL #2]       ;value of w[k] (scale index k×4 for 32-bit words)
        ldr     r0,=p   ;address of variable p
        str     r2,[r0] ;write to variable p

        ldr     r0,=k   ;address of k -> r0
        ldr     r1,=ps  ;address of ps -> r1
        str     r0,[r1] ;store address of k into pointer variable ps

        ldr     r2,=p   ;address of p -> r2
        ldr     r3,[r0] ;value of variable k -> r3 (r0 = address of k)
        str     r3,[r2] ;store value of variable k -> variable p
```

2.2. Arithmetic operations

Example Add/subtract instructions

```
ADD{S} {Rd,} Rn, Op2          ; Rd=Rn+Op2 or Rn=Rn+Op2
ADD{S} {Rd,} Rn, #imm12       ; Rd=Rn+imm12 or Rn= Rn+imm12
```

- 32 bit addition
- N, Z, C, and V bits of PSR are updated if suffix S is appended to the instruction, e.g. ADDS
- imm12 is any 12 bit number and is part of the instruction

```
SUB{S} {Rd,} Rn, Op2                  ; Rd=Rn-Op2 or Rn=Rn-Op2
RSB{S} {Rd,} Rn, Op2          ; Rd=Op2-Rn or Rn=Op2-Rn
SBC{S} Rdn, Rn                ; Rdn=Rdn-Rn-Not(C)
```

- 32-bit subtraction
- N, Z, C, and V bits of PSR are updated if suffix S is appended to the instruction, e.g. SUBS
- For subtraction operation, if there is borrow than carry bit is set to 0.

Ex: Assume R3= 1000, R2=4000, R4=3000 then

```
        ADDS R3, #250         ; R3=1250, N=Z=C=V=0
        ADDS R1, R2, R4       ;results in R1=7000, N=Z=C=V=0
```

Ex: Assume R3= 1000, R2=250, R4=3000 then

```
SUBS R3, R2        ; R3=750, N=Z= V=0 , C=1
SUBS R1, R4, R2    ; R1=2750, N=Z= V=0 , C=1
```

Example Multiply instructions

```
MUL{S} Rd, Rn, Rs        ;Rd = Least Significant 32 bit of(Rn * Rs) (Signed multiply)
UMUL{S} Rd, Rn, Rs       ;Rd = Least Significant 32 bit of(Rn * Rs) (unsigned multiply)
MLA Rd, Rn, Rs, Rm       ;Rd = Least Significant 32 bit of(Rm +Rn * Rs):Multiply and
                         ;accumulate
MLS Rd, Rn, Rs, Rm       ;Rd = Least Significant 32 bit of(Rm-Rn * Rs):Multiply and
subtract
```

Long multiply

```
UMULL Rdl, Rdh, Rn, Rs   ; Rdh:Rdl = Rn *Rs, Rdh = MSB bits, Rdl = LSB bits (unsigned)
SMULL Rdl, Rdh, Rn, Rs   ; Rdh:Rdl = Rn *Rs, Rdh = MSB bits, Rdl = LSB bits (signed)
UMLAL Rdl, Rdh, Rn, Rs   ; Rdh:Rdl = Rdh:Rdl +Rn *Rs (unsigned)
SMLAL Rdl, Rdh, Rn, Rs   ; Rdh:Rdl = Rdh:Rdl +Rn *Rs (signed)
```

Ex: Assume R3= 0x05, R2=0x50 then

```
MUL r1,r2,r3     ;  r1=0x0190
UMUL r1,r2,r3    ;  r1=0x0190
```

Example Divide instructions

- 32-bit Unsigned or signed divide
- These instructions do NOT save the remainder
- These instructions do not change any flags

```
UDIV{cond} {Rd,} Rn, Rm
SDIV{cond} {Rd,} Rn, Rm
```

- Rd is quotient if specified, otherwise Rn is quotient
- Rn is dividend
- Rm is divisor

Ex: Assume R3= 0x05, R2=0x50 then

```
UDIV r1,r2,r3    ;  r1=0x10
SDIV r1,r2,r3    ;  r1=0x10
```

2.3. Logical Operations

If the S flag is used then the flags are updated.

```
AND Rd, Rn,              ; Rd = Rd &Rn                                  Bitwise AND
AND Rd, Rn,#immed        ; Rd = Rn & #immed
AND Rd, Rn, Rm           ; Rd = Rn & Rm
ORR Rd, Rn               ; Rd = Rd | Rn                                 Bitwise OR
ORR Rd, Rn,#immed        ; Rd = Rn | #immed
ORR Rd, Rn, Rm           ; Rd = Rn | Rm
BIC Rd, Rn               ; Rd = Rd & (~Rn)                  Bit clear
BIC Rd, Rn,#immed        ; Rd = Rn &(~#immed)
BIC Rd, Rn, Rm           ; Rd = Rn &(~Rm)
BFC Rd, #lsb, #width     ; Rd(lsb+width-1 :lsb)=0           Bit field clear
BFI Rd, Rn, #lsb, #width ; Rd(lsb+width-1: lsb)=Rn(width-1:0)      Bit field insert
ORN Rd, Rn,#immed        ; Rd = Rn |(~#immed)                      Bitwise OR NOT
ORN Rd, Rn, Rm           ; Rd = Rn | (~Rm)
EOR Rd, Rn               ; Rd = Rd ^ Rn                    Bitwise Exclusive OR
EOR Rd, Rn,#immed        ; Rd = Rn | #immed
EOR Rd, Rn, Rm           ; Rd = Rn | Rm
```

#immed is a constant and must correspond to an 8-bit pattern rotated by an even number of bits within a 32-bit word

Ex:

Assume [r0]=0x0F = 0b00001111 = 15
[r4]=0xF0=0b11110000
[r1]=0xAD=0b10101101

```
AND r0,r0,#5  ; perform AND;  r0=0b00000101
ORR r4,r0,r4  ; perform OR; r4=0xFF
BFI  r4,r0,#8,#4; r4=0b0000010111111111
BFC r4,#1,#5  ; r4=0b0000000011000001
ORN r4,r0,r1  ; r4=0xFFFFFF57: OR r0 and complement of r1
```

Comparison instructions

```
CMP Rn, op2          ; Rn – op2 sets the NZVC bits
CMN Rn, op2          ; Rn - (-op2) sets the NZVC bits
TEQ  Rn, op2         ; Rn & op2 sets the NZVC bits
TST  Rn, op2         ; Rn .XOR. op2 sets the NZVC bits
```

Ex: Assume [r0]=0x0F, [r1]=0xAD

```
CMP r0,r1     ; compute r0 - r1 and update NZVC
BLE fblock    ; if r0 <= r1, branch to fblock: if N=1 or Z=1
```

2.4. Shift and Rotate

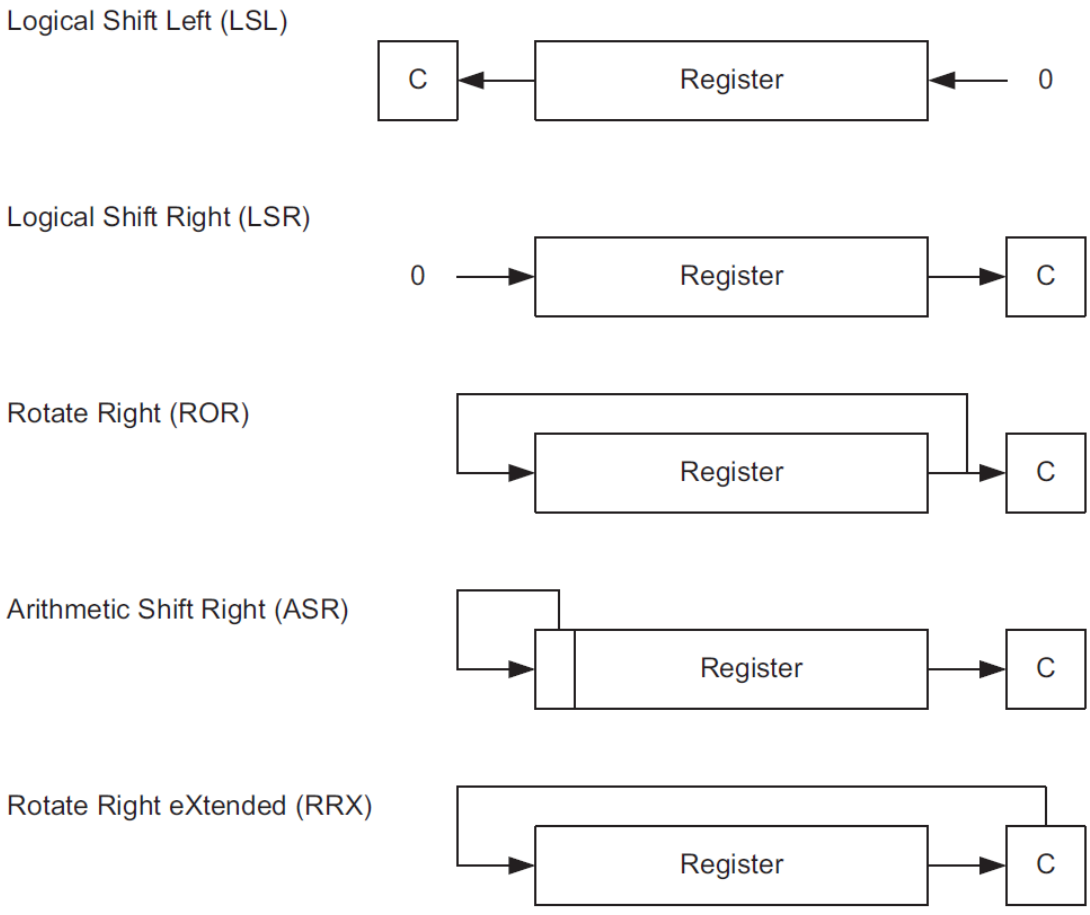
- All of these instructions update APSR condition flags
- Shift/rotate amount (in number of bits) specified by last operand

```
ASR Rd, Rn,#immed    ; Rd = Rn >> immed          Arithmetic shift right
ASR Rd, Rn           ; Rd = Rd >> Rn
```

```
ASR Rd, Rn, Rm          ; Rd = Rn >> Rm
LSL Rd, Rn,#immed       ; Rd = Rn << immed          Logical shift left
LSL Rd, Rn              ; Rd = Rd << Rn
LSL Rd, Rn, Rm          ; Rd = Rn << Rm
LSR Rd, Rn,#immed       ; Rd = Rn >> immed          Logical shift right
LSR Rd, Rn              ; Rd = Rd >> Rn
LSR Rd, Rn, Rm          ; Rd = Rn >> Rm
ROR Rd, Rn              ; Rd rotate by Rn           Rotate right
ROR Rd, Rn, Rm          ; Rd = Rn rotate by Rm
RRX Rd, Rn               ; {C, Rd} = {Rn, C}        Rotate right extended
```

İmmed is the shift length. The range of shift length depends on the instruction:

ASR: Shift length from 1 to 32.
LSL: Shift length from 0 to 31.
LSR: Shift length from 1 to 32.
ROR: Shift length from 1 to 31.

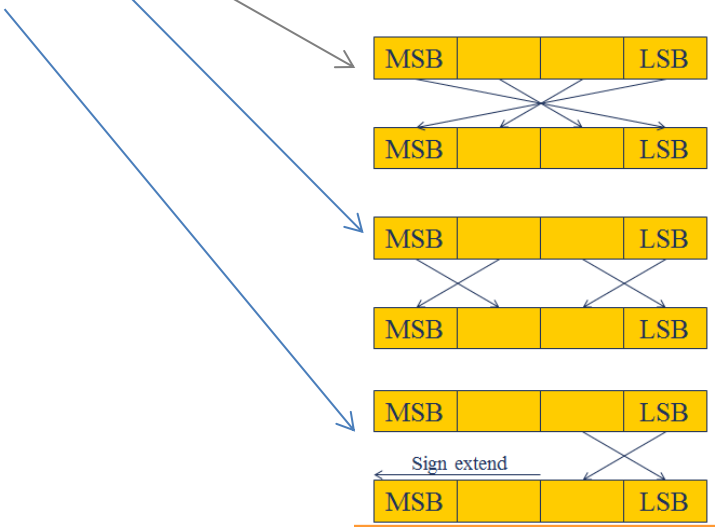


Ex: Assume [r0] = 0x0F = 0b00001111 = 15
[r4] = 0xF0 = 0b11110000 = 240

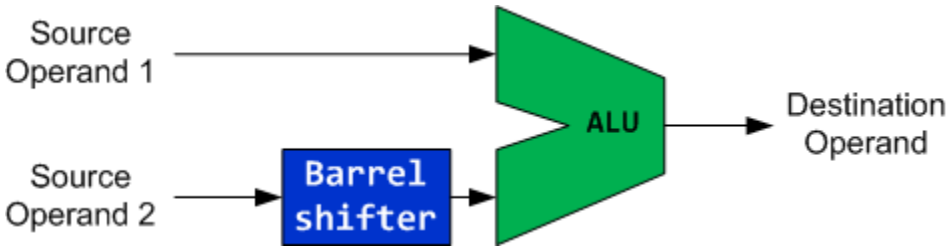
```
LSL r1, r0, #3          ; r1 = 0b01111000 = 0x78 = 120 = 15*2^3
LSR r3, r4, #2          ; r3 = 0b00111100 = 0x3C = 60 = 240/2^2
ASR r3, r4, #2          ; r3 = 0b00111100 = 0x3C = 60 = 240/2^2
```

Reversing bytes

```
REV Rd, Rn      ; Rd = rev(Rn) Reverse bytes in word
REV16 Rd, Rn    ; Rd = rev16(Rn) Reverse bytes in each half-word
REVSH Rd, Rn    ; Rd = revsh(Rn) Reverse bytes in bottom half-word and sign extend
                ;result
```

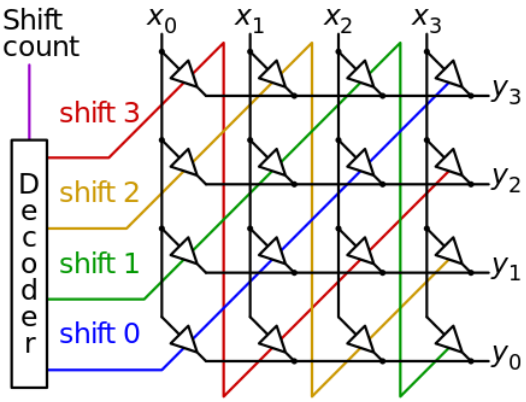


Using Barrel Shifter in arithmetic



The second operand of ALU has a special hardware called Barrel shifter which allows more powerful instruction when shifting is combined with math operations.

A Barrel shifter example:



Ex:

```
MOV r0, r0, LSL #1        ;Multiply R0 by two.
MOV r1, r1, LSR #2        ;Divide R1 by four (unsigned).
MOV r2, r2, ASR #2        ;Divide R2 by four (signed).
MOV r3, r3, ROR #16       ;Swap the top and bottom halves of R3.
ADD r4, r4, r4, LSL #4    ;Multiply R4 by 17. (N = N + N * 16)
```

2.5. Flow control operations

Unconditional Branches:

- Branch
  B label
  - Target address must be within 2 KB of branch instruction (-2048 B to +2046 B)
- Branch with link
  BL label
  - Call subroutine at <label>
  - PC-relative, range limited to PC+/-16MB
  - Save return address in LR
- Branch with link and exchange
  BLX Rd
  - Call subroutine at address in register Rd
  - Supports full 4GB address range
  - Save return address in LR
- Branch and exchange
  BX Rd
  - Branch to address specified by <Rd>
  - Supports full 4 GB address space
  - BX LR - Return from subroutine

Conditional Instructions:

Most data processing instructions can optionally update the condition flags in the Application Program Status Register (APSR) according to the result of the operation. Some instructions update all flags, and some only update a subset. If a flag is not updated, the original value is preserved.

You can execute an instruction conditionally, based on the condition flags set in another instruction, either:

- immediately after the instruction that updated the flags
- after any number of intervening instructions that have not updated the flags.

Conditional execution is available by using conditional branches or by adding condition code suffixes to instructions.The condition code suffix enables the processor to test a condition based on the flags. If the condition test of a conditional instruction fails, the instruction:

- does not execute
- does not write any value to its destination register
- does not affect any of the flags
- does not generate any exception.

Conditional instructions, except for conditional branches, must be inside an If-Then instruction block.

The condition flags

The APSR contains the following condition flags:

N Set to 1 when the result of the operation was negative, cleared to 0 otherwise.
Z Set to 1 when the result of the operation was zero, cleared to 0 otherwise.
C Set to 1 when the operation resulted in a carry, cleared to 0 otherwise.
V Set to 1 when the operation caused overflow, cleared to 0 otherwise.

A carry occurs:

- if the result of an addition is greater than or equal to 2^{32}
- if the result of a subtraction is positive or zero
- as the result of an inline barrel shifter operation in a move or logical instruction.

Overflow occurs when the sign of the result, in bit[31], does not match the sign of the result had the operation been performed at infinite precision, for example:

- if adding two negative values results in a positive value
- if adding two positive values results in a negative value
- if subtracting a positive value from a negative value generates a positive value
- if subtracting a negative value from a positive value generates a negative value.

Most instructions update the status flags only if the S suffix is specified.

Condition code suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned
CC or LO	C = 0	Lower, unsigned
MI	N = 1	Negative
PL	N = 0	Positive or zero

Suffix	Flags	Meaning
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned
LS	C = 0 or Z = 1	Lower or same, unsigned
GE	N = V	Greater than or equal, signed
LT	N != V	Less than, signed
GT	Z = 0 and N = V	Greater than, signed
LE	Z = 1 and N != V	Less than or equal, signed
AL	Can have any value	Always. This is the default when no suffix is specified.

Conditional branch:

- B{cond} label
- BL{cond} label
- BX{cond} Rm
- BLX{cond} Rm

Where {cond} Is an optional condition code

- Compare and branch (Thumb2)

```
CBZ r0,label        ;branch if r0 == 0
CBNZ r0,label       ;branch if r0 != 0
```

The Compare operations are identical to subtracting, for CMP, or adding, for CMN, except that the result is discarded.

If-Then conditional:

- IT{x{y{z}}} cond

where:
x Specifies the condition switch for the second instruction in the IT block.
y Specifies the condition switch for the third instruction in the IT block.
z Specifies the condition switch for the fourth instruction in the IT block.
cond Specifies the condition for the first instruction in the IT block.

The condition switch for the second, third and fourth instruction in the IT block can be either:
T Then. Applies the condition cond to the instruction.
E Else. Applies the inverse condition of cond to the instruction.

Ex:

```
ITTE NE                     ; Next 3 instructions are conditional
ANDNE R0, R0, R1            ; ANDNE does not update condition flags
ADDSNE R2, R2, #1           ; ADDSNE updates condition flags
```

MOVEQ R2, R3 ; Conditional move

2.6. Miscellenous

- No Operation - does nothing!
  NOP
- Breakpoint - causes hard fault or debug halt - used to implement software breakpoints
  BKPT #imm8
- Wait for interrupt - Pause program, enter low-power state until a WFI wake-up event occurs (e.g. an interrupt)
  WFI

Sample questions:
State whether the instruction is correct or not. Why?

```
ADD     r3,r7,#1020
AND     r0,r5,r2
SUB     r11,r12,r3,ASR #5
MOVS    r4,r4, LSR #32
SUB     r11,r12,r3,LSL #32
MOVS    r4,r4,RRX #3
```

Chapter 4

Programming Examples

Example 1: (simple addition) Write a program which adds the contents of memory locations 0x00000040 and 0x00000044 and stores the sum to memory location 0x00000048.

```
Label   mnemonic                comments
        LDR R0, =0x00000040     ;Set the operand's address to R0/ set R0 as address
                                ;pointer
        LDR R1, [R0], #4        ;Load the first operand to R1 and increment pointer
        LDR R2, [R0], #4        ;Load the second operand to R2 and increment pointer
        ADD R1, R2              ;R1=R1+R2
        STR R1, [R0]            ;Store R1 to address 0x00000048
Done    B Done                  ;Loop at this instruction
```

Example 2: (nibble separation) Write a program which takes the 32 bits stored starting at the location 0x20000040 and stores the four least significant bits (LSBs) of the least significant byte in location 0x20000044, i.e., separate these bits, and store the four most significant bits (MSB) of the least significant byte in location 0x20000045 as the least significant bits.

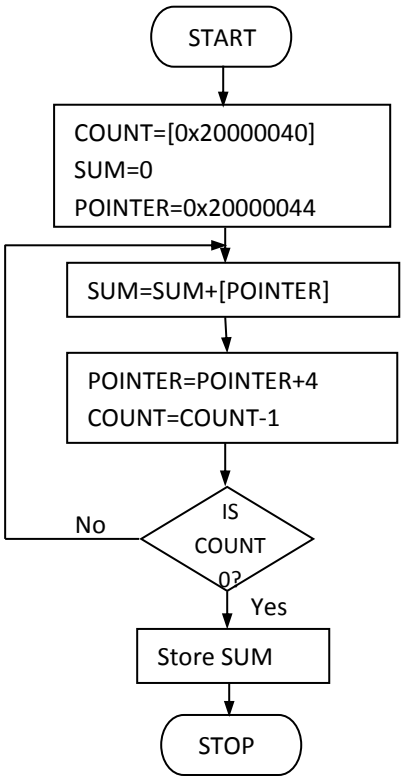
```
Label       mnemonic                comments
            LDR R0, =0x20000040     ;Set the operand's address to R0
            LDRB R1, [R0]           ;Load the byte at the address R0
            BFC R1, #4, #4          ;Clear bits from 4 to 7 of R1
            STRB R1, [R0, #4]       ;Store the byte to R0+4
            LDRB R1, [R0]           ;Load the byte at the address R0
            LSR R1, R1, #4          ;Shift the four MSB to LSB position
            STRB R1, [R0, #5]       ;Store the byte to R0+5
Done        B Done                  ;Loop at this instruction
```

Example 3: Write a program which places the lower half word of the 4 byte data saved starting from location 0x20000040 in location 0x20000050 and the higher half word in location 0x00000060.

```
Label       mnemonic                comments
            LDR R0, =0x20000040     ; Set the operand's address to R0
```

```
                LDRH R1, [R0]                ; Lower half is loaded to R1 (little endian)
                STRH R1, [R0, #0x10]
                LDRH R1, [R0, #2]            ; Upper half word is loaded to R1
                STRH R1,  [R0, #0x20]
Done            B Done                       ; Loop at this instruction
```

Example 4: (addition of an array of *n* four-byte unsigned numbers) Memory location 0x20000040 contains the length (≠0) of a set of numbers. The set starts at memory location 0x20000044. Write a program that stores the sum of numbers (<2^{64}) starting from memory location 0x20001000.



Label	mnemonic	comments
	LDR R0, =0x20000040	; Set the address of the length to R0,
		; R0 to be used as the pointer later
	LDR R1, [R0], #4	; R1 in COUNT, RO is POINTER
	BFC R2, #0, #32	; R2 and R3 to store SUM, R2 is least significant
	BFC R3, #0, #32	
	LDR R2, [R0], #4	; Load the first number and increment POINTER

```
                SUBS R1, #1             ; Reduce COUNT by 1


                BEQ End                 ; If COUNT is zero, branch to End
Loop            LDR R4, [R0], #4        ; Load the following number
                                        ; and increment POINTER
                ADDS R2, R4             ; Add the number to SUM's least significant word
                BCC Cont                ; If there is no carry, branch to Cont
                ADD R3, #1              ; else increment R3
Cont            SUBS R1, #1             ; Reduce COUNT by 1
                BEQ End                 ; If COUNT is zero, branch to End
                B Loop
End             LDR R0, =0x20001000     ; Set the address of storage to R0
                STRD R2, R3, [R0]       ; Save first R2 and then R3 starting from [R0]
Done            B Done                  ; Loop at this instruction
```

Example 5: (data transfer) The length of the data array is in memory location 0x20000000, the data originally starts in memory location 0x20004000 (Source Address), and the destination area for the data starts in memory location 0x20008000 (Destination Address). Write a program to transfer the data.

```
Label           mnemonic                comments
                LDR R0, =0x20000000
                LDR R1, [R0]            ; R1 holds the array length/counter
                LDR R2, =0x20004000     ; R2 is a pointer to source area
                LDR R3, =0x20008000     ; R3 is a pointer to destination area
Loop            LDR R4, [R2], #4        ; Load the data and increment the source pointer
                STR R4, [R3], #4        ; Store the data and increment the destination
pointer
                SUBS R1, #1             ; Decrement length counter
                BNE Loop                ; If there are more data to read go to Loop
End             B End
```

Example 6: (condition code flags in Program Status Register (PSR))

Assume register R1 contains the following data and register R2 contains 0x1000.0000 before the execution of "SUBS R1, R1, R2" instruction. What is the result in R1 and the N, Z, C, and V bits?

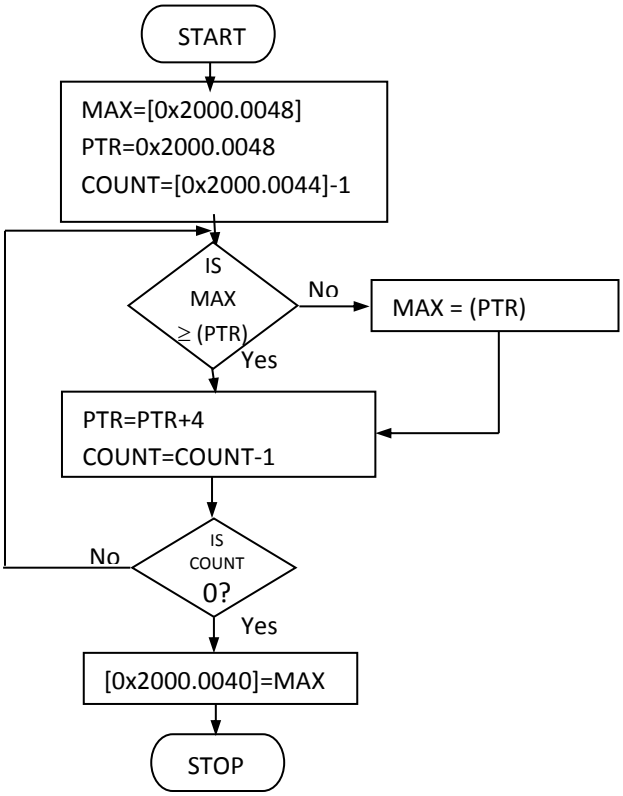
R1: 0x2000.0000;

0x0000.1000;

0x1000.0000

Before	After	
R1	*R1*	*N Z C V*
0x2000.0000	0x1000.0000	0 0 1 0
0x0000.1000	0xF000.1000	1 0 0 0
0x1000.0000	0x0000.0000	0 1 1 0

Example 7: (maximum value) Length (≠0) of a memory array, which contains unsigned numbers, is in 0x2000.0044 and the array starts at 0x2000.0048. Write a program that places the maximum value in the array in 0x2000.0040



label	mnemonic	comment
	LDR R0, =0x20000044	; R0 is the pointer to the length

```
        LDR R4, =0x20000040         ; R4 is the pointer to the max storage
        LDR R1, [R0], #4            ; R1 holds the length/count information
                                    ; R0 updated to point the array
        LDR R2, [R0]            ; R2 holds the maximum
        SUBS R1, #1             ; Decrement counter
        BEQ Final               ; If it is the end of the array, finish
Loop    LDR R3, [R0, #4]!       ; R3 holds the next data
        CMP R2, R3              ; R2 - R3
        BHI Cont                ; If R2>R3, go to Cont
        LDR R2, [R0]            ; Else load the new data to max (R2)
Cont    SUBS R1, #1             ; Decrement counter
        BEQ Final               ; If it is the end of the array, finish
        B Loop                  ;Else go to Loop
Final   STR R2, [R4]            ; Store max
Forever B Forever
```

Assume that the initial values of the locations 0x0000.0040:0x0000.0053 are as given below, a trace example for the above program is as follows:

Location	Content
0x2000.0040	0x0000.0000
0x2000.0044	0x0000.0003
0x2000.0048	0x0000.0005
0x2000.004C	0x0000.0002
0x2000.0050	0x0000.0006

Instruction	R1	R2	R3	R0	CZ	R4	0x 20000040
LDR R0, =0x20000044				0x 20000044			
LDR R4, =0x20000040						0x20000040	
LDR R1, [R0], #4	3			0x 20000048			
LDR R2, [R0]		5					
SUBS R1, #1	2				10		

BEQ Final							
LDR R3, [R0, #4]!			2	0x 2000004C			
CMP R2, R3					10		
BHI Cont							
SUBS R1, #1	1				10		
BEQ Final							
B Loop							
LDR R3, [R0, #4]!			6	0x 20000050			
CMP R2, R3					00		
BHI Cont							
LDR R2, [R0]		6					
SUBS R1, #1	0				01		
BEQ Final							
STR R2, [R4]							6
B Forever							
B Forever							
….							

Example 8: (multiple precision arithmetic) Two n-word (Word is 32 bits) numbers will be added. The number of words (n, which is greater than 0 and less than 256) in the numbers is given in location 0x2000.0040. First number starts (least significant word first) at 0x2000.0044 and the second number at 0x2000.0144. The sum is to replace the first number.

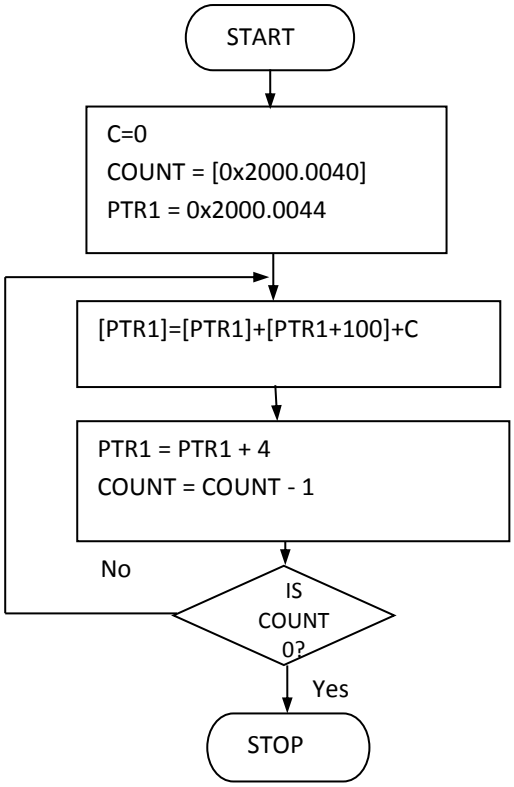
Ex: n1 = 50 34 29

n2 = 28 97 20

+ _____

sum = 79 21 49

Location	Before	After
0x2000.0040	03	
0x2000.0044	29	49
0x2000.0048	34	71
0x2000.004C	50	79
...	...	
0x2000.0144	20	
0x2000.0148	97	
0x2000.014C	28	



label	mnemonic	comment
	LDR R0, =0x20000040	; Pointer to address 0x20000040
	LDRB R2, [R0], #4	; R2 is Length counter, R0 is incremented
	ADDS R2, #0	; Dummy instruction to clear C bit
Loop	LDR R3, [R0]	; First number is read to R3
	LDR R4, [R0, #0x0100]	; Second number to R4
	ADCS R3, R3, R4	; Add second number to first
	STR R3, [R0], #4	; Store R3, increment pointer
	SUB R2, #1	; Decrement counter without changing PSR
	TEQ R2, #0	; Does not change C bit
	BNE Loop	; If counter is not equal to zero go to Loop
Forever	B Forever	

Example 9: (square from a lookup table) Write a program to find the square of a 16-bit binary number from a lookup table. The table starts at SQTAB, location NUM contains the number whose square is required and the result will be saved in location NUM again.

Assume that SQTAB contains 0, 1, 4, 9, 16, 25, 36, 49, ….

```
label       mnemonic                comment
            ADR R0, SQTAB           ; R0 is set as the pointer to the table
            ADR R2, NUM             ; R1 is set as the pointer to the number
            LDR R1, [R2]
            ADD R0, R0, R1, LSL #2  ; R0 is moved to the location of square
            LDR R1, [R0]            ; Square of the number is loaded to R1
            STR R1, [R2]            ; It is stored to NUM
Forever     B Forever
```

Example 10: (character manipulation) Determine the length of a string of ASCII characters (one ASCII character is coded by one byte) starting at ARRAY and ending with period ".". Store the length of the string to location LENGTH. ASCII equivalent of "." is 0x2E.

e.g.

Location	Content	
ARRAY	54	T
ARRAY+1	4F	O
ARRAY+2	4F	O
ARRAY+3	20	
ARRAY+4	4C	L
ARRAY+5	41	A
ARRAY+6	54	T
ARRAY+7	45	E
ARRAY+8	2E	.

```
label       mnemonic              comment
            LDR R0, =ARRAY        ; R0 is pointer to ARRAY
            MOV R1, #0x00         ; R1 holds length counter, initialized to 0
Loop        LDRB R2, [R0], #1     ; Read the data, increment pointer by one
            CMP R2, #0x2E         ; Compare the byte with '.'
            BEQ End               ; If equal go to End
            ADD R1, R1, #1        ; else increment counter and go to Loop
            B Loop
End         LDR R0, =LENGTH       ; Store the length information
            STR R1, [R0]
Forever     B Forever
```

Example 11: (pattern comparison) Two strings of 32 bit words start in memory locations STRING1 and STRING2, respectively. Memory location LENGTH contains the length ($\neq 0$) of the strings. Write a program that compares these two strings. If they are equal, the program will place zeros to location LENGTH, otherwise ones will be placed.

```
Label       mnemonic              comment
            LDR R0, =STRING1      ; Pointer to STRING1
            LDR R1, =STRING2      ; Pointer to STRING2
            LDR R2, =LENGTH       ; Length and final decision
            LDR R3, [R2]          ; R3 keeps the length
            MOV R7, #0xFFFFFFFF   ; not equal = Set R7 bits to 1's
Loop        LDR R5, [R0], #4      ; R5 keeps the element from STRING1
            LDR R6, [R1], #4      ; R6 keeps the element from STRING2
            CMP R5, R6            ; Compare R5 and R6
            BNE End               ; If they are not equal go to End
            SUBS R3, #1           ; else continue checking and decrement counter
            BNE Loop
            BFC R7, #0, #32       ; equal = Set R7 bits to 0's
End         STR R7, [R2]          ; store the decision
Forever     B Forever
```

Example 12: (Finite impulse response filtering) For a causal discrete-time FIR filter of order N, each value of the output sequence is a weighted sum of the most recent input values:

$$y[n] = c_0 x[n] + c_1 x[n-1] + \cdots + c_N x[n-N] = \sum_{i=0}^{N} c_i x[n-i]$$

Write a program which computes output $y[n]$ when c_i's and $x[n-i]$'s are stored starting at memory locations COEFF and INPUT, respectively. The order N of the filter is stored at FORDER. Store the computed value of $y[n]$ to memory location OUTPUT.

Label	mnemonic	comment
	MOV r0,#0	; use r0 for i
	MOV r8,#0	; use separate index for arrays
	LDR r2,=FORDER	; get address for N
	LDR r1,[r2]	; get value of N
	MOV r2,#0	; use r2 for y[n]
	LDR r3, =COEFF	; load r3 with base of c
	LDR r5,=INPUT	; load r5 with base of x
Loop	LDR r4,[r3,r8]	; get c[i]
	LDR r6,[r5,r8]	; get x[n-i]
	MUL r4,r4,r6	; compute c[i]*x[n-i]
	ADD r2,r2,r4	; add into running sum y[n]
	ADD r8,r8,#4	; add word offset to array index
	ADD r0,r0,#1	; add 1 to i
	CMP r0,r1	; exit?
	BLT loop	; if i < N, continue
	LDR r3,=OUTPUT	; else get address for y[n]
	STR r2, [r3]	; store y[n]

CHAPTER 5

Stacks, subroutines, interrupts

1. Stack

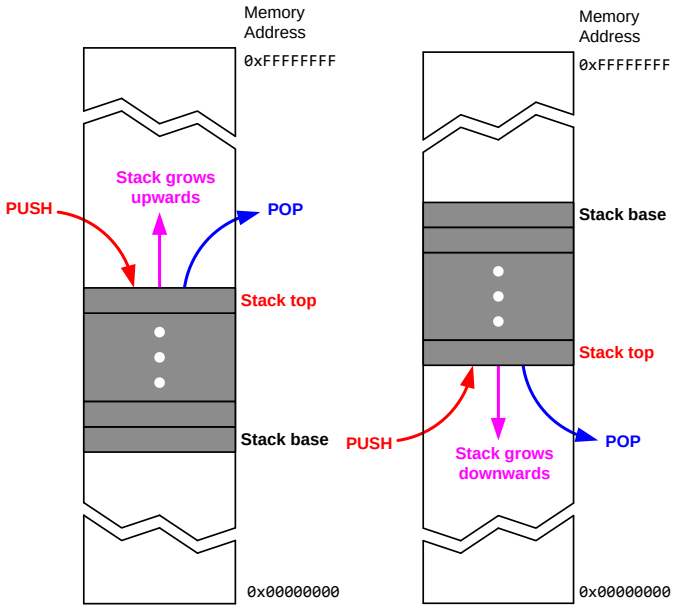
The stack is a data structure, known as last in first out (LIFO). In a stack, items entered at one end leave in the reversed order. Stack can be used for

- Storing original data in registers in subroutine so that the values can be restored at the end of the subroutine
- Passing information to subroutine
- Storing local variables
- Holding processor status and registers when an exception occurs

Stacks in microprocessors are implemented by using a stack pointer to point to the top of the stack in memory. Normally, there are two types of the stacks depending on which way the stack grows.

- *Ascending Stack* - When items are pushed on to the stack, the stack pointer is increasing. That means the stack grows towards higher address.

- *Descending Stack* - When items are pushed on to the stack, the stack pointer is decreasing. That means the stack is growing towards lower address.



Depending on what the stack pointer points to, the stacks can also be categorized into the following two types:

- *Empty Stack* - Stack pointer points to the location in which the next item will be stored.
- *Full Stack* - Stack pointer points to the location in which the last item was stored.

The Cortex-M4 processor uses a full descending stack, i.e. stack pointer holds the address of the last stacked item in memory. When the processor pushes a new item onto the stack, it decrements the stack pointer and then writes the item to the new memory location. When the processor pulls an item from the stack, it reads the item from the stack and increments the stack pointer.

In Cortex-M4, there are two stack pointers.

- Main Stack Pointer (MSP) : Use at start-up and exception handlers, including OS operations.
- Process Stack Pointer (PSP): Typically used by application tasks in a multitasking system.

Both of them are 32-bit registers and can be referenced as R13, but only one is used at one time, depending on bit[1] of the control regsiter and the current mode (Handler or Thread). In Handler mode, the processor always uses the main stack. In Thread mode, bit[1] of the CONTROL register indicates the stack pointer to use:

- 0 = Main Stack Pointer (MSP). This is the reset value.
- 1 = Process Stack Pointer (PSP)

For simple applications, we can use MSP all the time.

1.1. PUSH/POP instructions

PUSH stores multiple registers on the stack, with the lowest numbered register using the lowest memory address and the highest numbered register using the highest memory address, irrespective of their order in the instruction.

POP loads multiple registers from the stack, with the lowest numbered register using the lowest memory address and the highest numbered register using the highest memory address, irrespective of their order in the instruction.

PUSH uses the value in the SP register minus four as the highest memory address, POP uses the value in the SP register as the lowest memory address, implementing a full-descending stack. On completion, PUSH updates the SP register to point to the location of the lowest store value, POP updates the SP register to point to the location above the highest location loaded.

PUSH {*Rd*}

- SP = SP-4 ⟶ descending stack
- (*SP) = *Rd* ⟶ full stack

Push multiple registers

PUSH {r6, r7, r8} ⟷ **PUSH {r8, r7, r6}** ⟷ **PUSH {r8}**
PUSH {r7}
PUSH {r6}

POP {*Rd*}

- *Rd* = (*SP) ⟶ full stack
- SP = SP + 4 ⟶ Stack shrinks

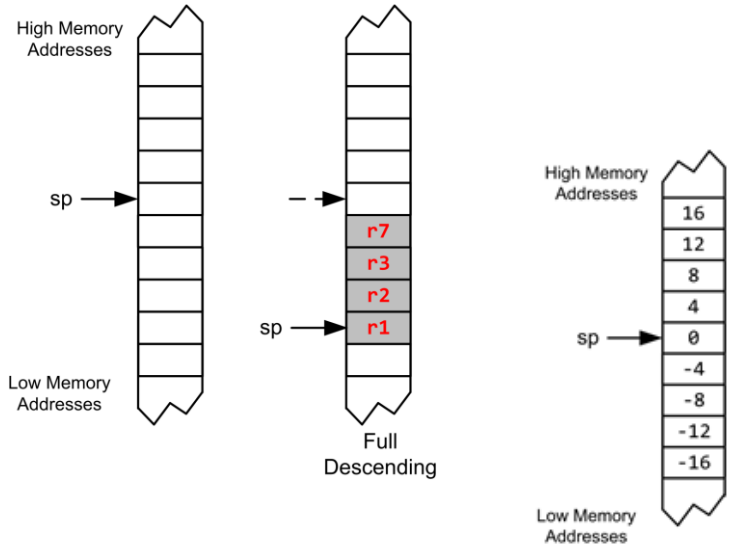
Pop multiple registers

POP {r6, r7, r8} ⟷ **POP {r8, r7, r6}** ⟷ **POP {r6}**
POP {r7}
POP {r8}

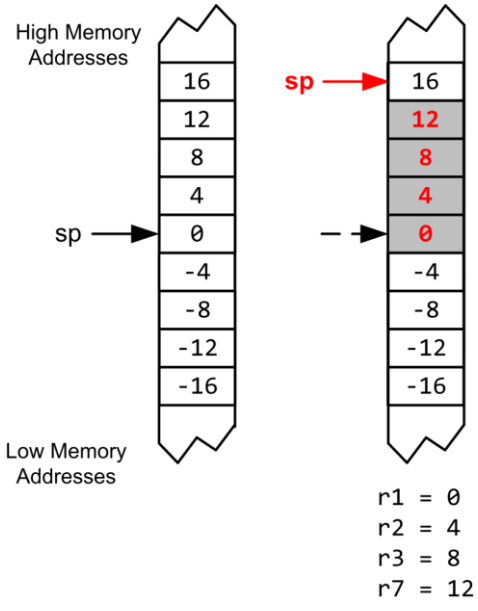
Ex:

Assume r1=0, r2=4, r3=8, r7=12

PUSH {r3, r1, r7, r2}



POP {r3, r1, r7, r2}



Before using the stack, software has to define stack space in the SRAM of the microcontroller and initialize the stack pointer (SP). SP starts at **0x20000200** for TM4C123GXL board. On system reset this value is read from the vector table (adress 0x00000000).

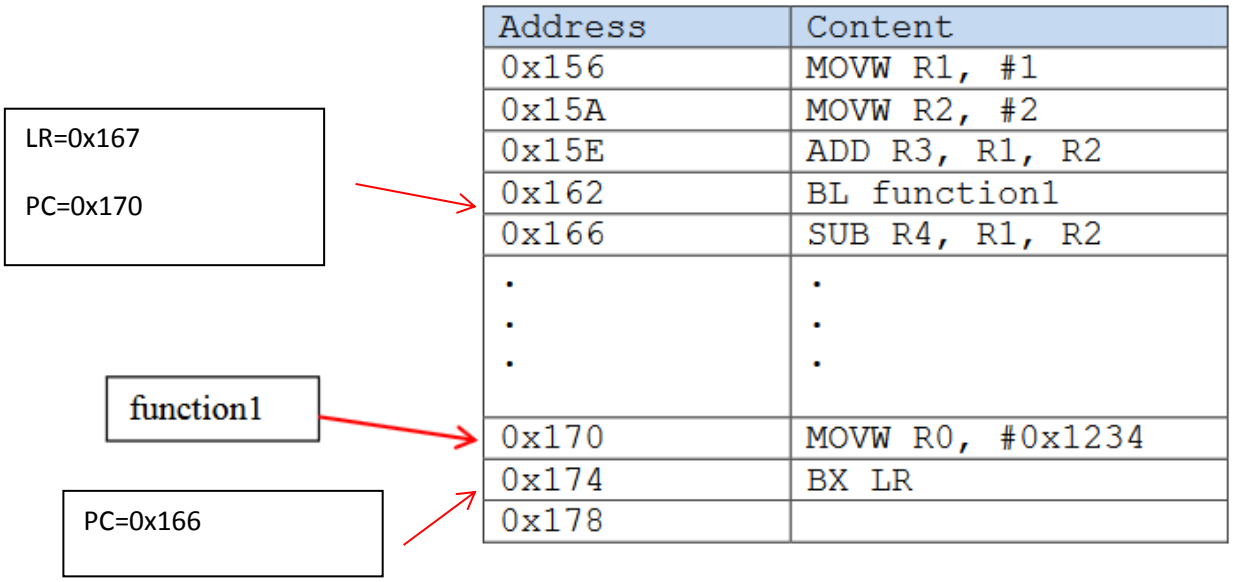
2. Subroutines

Subroutines are code sequences that can be called to perform specific tasks. The use of subroutines allows us to develop modular programs. Modular programming is preferred to develop complex system using simple components.

To call a subroutine, the Branch and Link (BL) instruction or Branch and Link with eXchange (BLX) instruction can be used. These instructions will save the return address to the Link Register (LR or R14) and execute the unconditional branch to the subroutine label.

A subroutine name is identified by the label at the first instruction in the subroutine. The last instruction should be the BX LR instruction which retrieves the return address from LR and return to the instruction immediately after the instruction that performed the subroutine call. Note that although the return addresses are always even (bit 0 = '0' for half-word aligned), the bit 0 in the LR is actually '1' to indicate the Thumb state. This is done automatically.

Ex:



Address	Content
0x156	MOVW R1, #1
0x15A	MOVW R2, #2
0x15E	ADD R3, R1, R2
0x162	BL function1
0x166	SUB R4, R1, R2
. . .	. . .
0x170	MOVW R0, #0x1234
0x174	BX LR
0x178	

You can define the beginning and end of subroutines using PROC – ENDP directives.

Ex:

Caller Program	Subroutine/Callee

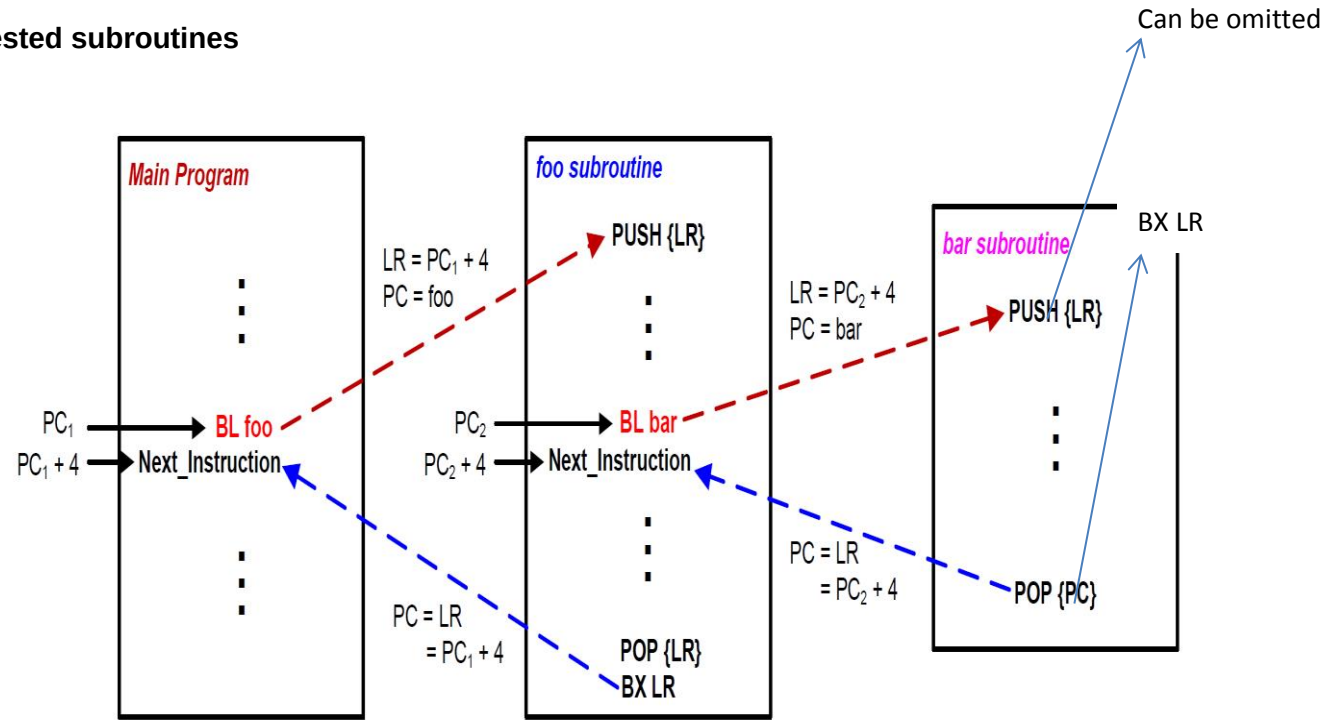
MOV r4, #100
...
BL foo
...
ADD r4, r4, #1 ; r4 = 101, not 11
	foo PROC
	...
	MOV r4, #10 ; foo changes r4
	...
	BX LR
	ENDP

Preserve run time environment

Caller Program	Subroutine/Callee

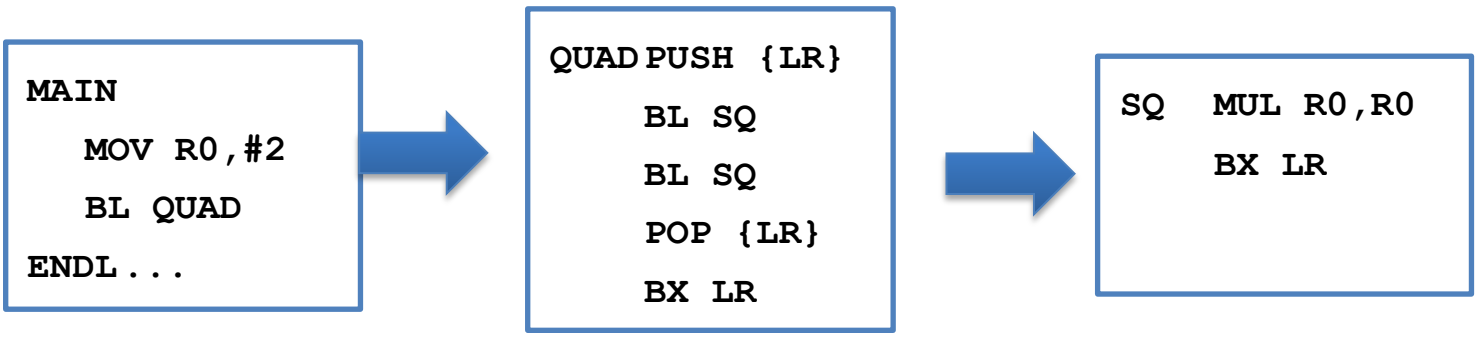
MOV r4, #100
...
BL foo
...
ADD r4, r4, #1 ; r4 = 101, not 11
	foo PROC
	PUSH {r4} ; preserve r4
	...
	MOV r4, #10 ; foo changes r4
	...
	POP {r4} ; Recover r4
	BX LR
	ENDP

Nested subroutines



Ex:

Find R0=R0^4



3. Interrupts

One of the methods to read an input from an I/O port is using the polling technique. Polling is a technique to monitor an I/O port and to trigger an appropriate action when a change is detected. The general idea is that the processor periodically reads the I/O port and determines if there is a change. If there is a change, the processor will execute the subset of code to deal specifically with this change. Otherwise, the processor will continue with normal operations.

Another way to read an input from an I/O port is using an interrupt. Interrupt-based program allows the processor to continue processing the main task without periodically checking for a change at an I/O pin. Interrupt is an exception caused by an explicit request signal from a peripheral or hardware device. An interrupt cause the automatic transfer of software execution outside of the normal programmed sequence (e.g. to provide service to the peripheral). When a peripheral or a hardware device needs service form the processor, typically:

- It asserts an interrupt request to the processor,
- The processor completes the current instruction then it suspends the current task and jumps to an Interrupt Service Routine (ISR) to service the peripheral where ISR is a dedicated section of code executed in response to the interrupt,
- Then, the processor resumes the previously suspended task.

There might also be software interrupts. In general the purpose of an interrupt is to alert the microprocessor that some important event has occurred and the operating state of the application should change to accommodate this event. The event interrupts the normal application flow in order to properly respond to the interrupt.

3.1. Interrupt Service Routine Execution

When an interrupt occurs, how does the interrupt service routine get executed? One common misconception is that the main application calls the interrupt service routine just like it would call any other function. That, however, is not the case. When an interrupt is detected, the microprocessor enters into a special interrupt mode called Handler mode. The microprocessor will save all of the general purpose registers, any status registers, and the program counter to a reserved portion of SRAM. This collection of registers fully describes the current computational state of an application. Saving these registers is called a context

save. The context save allows the processor to halt the execution of an application and resume the application at a later time.

Once the context save is finished, the hardware will automatically modify the program counter (PC) and load it with the pre-defined address of an interrupt service routine. The ISR executes and carries out the instructions in the ISR. When the ISR exits, the microprocessor does a context restore by restoring the general purpose registers, PC, and status registers with the values each register held immediately before the interrupt. The final step the microprocessor carries out is to exit Handler Mode. At this point, the PC contains the address of the next instruction in the application and the application continues on as normal. Its important to realize that the ISR does not return to the main routine using the value in the link register. The microprocessor 'returns' to the main application by restoring the value of the PC when the microprocessor exits Handler Mode.

Polled vs Vectored Interrupts

When an interrupt is detected and hardware has saved the context of the application, how does it know which interrupt service vector to execute? There are two common approaches to determining how to execute the correct interrupt service routine.

The first approach is called Polled Interrupts. If a microprocessor uses polled interrupts, the microprocessor will branch to a predefined location that contains a master ISR. Once inside the master ISR, the source of the interrupt is determined by examining interrupt status registers. The interrupt status registers will contain a bit mask that indicates which interrupts are currently active. Once the active interrupt has been determined, the master ISR will then make a function call to a function that acts as the dedicated ISR for the specified device.

The second approach to executing an ISR is called a vectored interrupt handler. A processor that implements a vectored interrupt handler has dedicated memory addresses for each peripheral device. Each of those memory address locations contains the address of the ISR associated with the specified peripheral. When an interrupt occurs, the microprocessor hardware detects which device caused the interrupt and will load the PC with the address specified in the vector table. Vectored interrupts normally result in a more complex interrupt mechanism, but it provides superior performance since software is not responsible for determining the active interrupt source.

The Cortex-M4 processor supports vector interrupts. It provides a Nested Vectored Interrupt Controller (NVIC) (Fig. 5.1) for interrupt handling. NVIC provides deterministic, fast interrupt processing: always 12 cycles, or just 6 cycles with tail-chaining.

For any microcontroller, the NVIC receives interrupt requests from various sources. In addition to interrupt requests, there are some other events that need servicing. Together with interrupts, they are called "exceptions". For Cortex-M4 processor, exceptions include resets, software interrupts and hardware interrupts. Each exception has an associated 32-bit vector that stores the memory location where the start address of the ISR that handles the exception is located. These vectors are stored in ROM at the beginning of memory which is called as the vector table. In addition to the exception vectors, vector table also contains the reset value of the stack pointer. On system reset, the vector table is fixed at address 0x0000.0000. Privileged software can write to the Vector Table Offset (VTABLE) register to relocate the vector table start address to a different memory location, in the range 0x0000.0400 to 0x3FFF.FC00 The least-significant bit of each vector in the vector table must be 1, indicating that the exception handler is Thumb code.The program jumps to this specific memory location indicated by the vector table using Nested Vectored Interrupt Controller (NVIC).

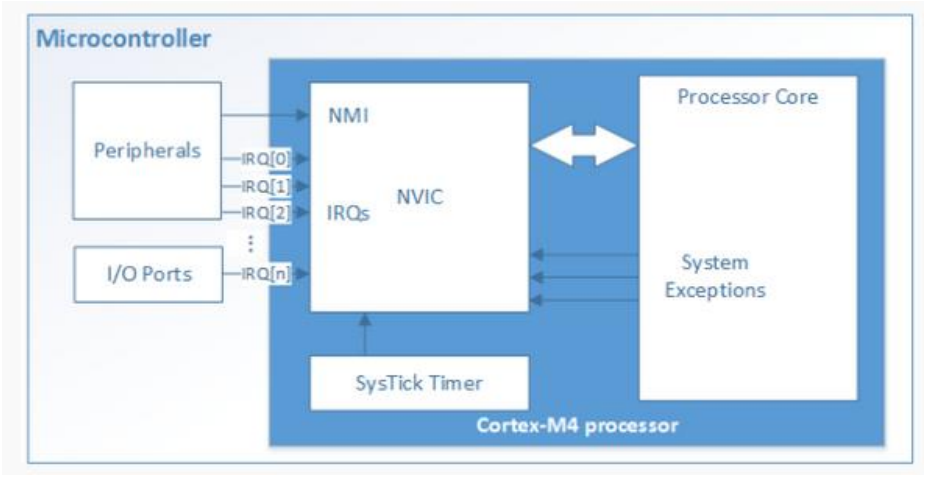


Fig. 5.1. NVIC in ARM Cortex-M4

The Cortex-M4 NVIC supports up to 240 interrupt requests (IRQs), a non-maskable interrupt (NMI), a SysTick timer interrupt and a number of system exceptions. Most of these IRQs are generated by peripherals such as timers, GPIO ports and communication interfaces such as UARTs. As given in Table 5.1, the NVIC in Cortex-M4 assigns the first 15 interrupts for internal use.

Vector Number	Interrupt Type	Priority	Vector Address
0	–	–	0x0000.0000
1	Reset	-3 (highest)	0x0000.0004
2	NMI	-2	0x0000.0008
3	Hard Fault	-1	0x0000.000C
4	Memory Management Fault	programmable	0x0000.0010
5	Bus Fault	programmable	0x0000.0014
6	Usage Fault	programmable	0x0000.0018
7 ~ 10	Reserved	–	0x0000.001C ~ 0x0000.0028
11	SVCall	programmable	0x0000.002C
12	Debug Monitor	Programmable	0x0000.0030
13	Reserved	–	0x0000.0034
14	PendSV	programmable	0x0000.0038
15	SysTick	programmable	0x0000.003C
16 ~ 255	Interrupts	programmable	0x0000.0040 ~ 0x0000.03FC

Table 5.1 Processor exceptions.

Nested Vectored Interrupt Controller (NVIC) prioritize and handle all exceptions in Handler Mode. The processor state is automatically stored to the stack on an exception and automatically restored from the stack at the end of the Interrupt Service Routine (ISR), fault or system handlers. The vector is fetched in parallel to the state saving, enabling efficient interrupt entry. The processor supports tail-chaining, which enables back-to-back interrupts to be performed without the overhead of state saving and restoration.

Table 5.1 and 5.2 list all exception types.

Exception Handlers

The processor handles exceptions using:

- **Interrupt Service Routines (ISRs).** Interrupts (IRQx) are the exceptions handled by ISRs.
- **Fault Handlers.** Hard fault, memory management fault, usage fault, and 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.

Exceptions can be classified into various different types and the handling programs are named differently:

Reset: It is a special kind of exception. When reset is activated, the CPU goes to a known state with all registers loaded with the predefined values. When the device is coming out of reset, the ARM Cortex-M4 loads the program counter from memory location at 0x000.0004. Reset has the highest priority.

NMI: The non-maskable interrupt (NMI) has the second highest priority after Reset. As shown in Figure 5.1, NMI is also an input signal into the CPU. It cannot be masked by software, for this reason, it is called non-maskable interrupt. Whenever it is activated, the CPU will go to address 0x0000.0008 to get the address of its interrupt service routine.

Bus Fault: This kind of exception occurs when there is an error in accessing the buses. Memory access problem during the fetch stage of an instruction or reading and writing to data section of memory can lead to some bus fault exceptions.

Usage Fault: It is an exception that occurs because of a fault related to instruction execution, such as an undefined instruction, invalid state of instruction execution, an error on execution return, unaligned memory access and so on.

Debug Monitor: When executing a program, sometimes there is a need to examine the contents of the CPU's registers and system memory. This is often done by executing the program one instruction at a time and then inspecting registers and memory. ARM microcontroller has a designated interrupt, Debug Monitor, which can be used for debug events like breakpoints, watch points.

SysTick: It is capable of generating exceptions whenever system timer counts down to zero. In the multitasking OS a real time interrupt clock is needed to notify the CPU to the task, which is ready to be executed. The SysTick is designed for this purpose.

Interrupt: An interrupt (IRQ) is an exception signaled by a peripheral or generated by a software request. It can be prioritized and is asynchronous to instruction execution.

Peripherals, which are capable of generating interrupts, include GPIO, UART, SSI, Timers, ADC, and so on. Table 5.2 lists the interrupt assignments in TM4C123GXL LaunchPad.

Vector Number	Interrupt Number (Bit in Interrupt Registers)	Vector Address or Offset	Description
0-15	-	0x0000.0000 - 0x0000.003C	Processor exceptions
16	0	0x0000.0040	GPIO Port A
17	1	0x0000.0044	GPIO Port B
18	2	0x0000.0048	GPIO Port C
19	3	0x0000.004C	GPIO Port D
20	4	0x0000.0050	GPIO Port E
21	5	0x0000.0054	UART0
22	6	0x0000.0058	UART1
23	7	0x0000.005C	SSI0
24	8	0x0000.0060	I^2C0
25	9	0x0000.0064	PWM0 Fault
26	10	0x0000.0068	PWM0 Generator 0
27	11	0x0000.006C	PWM0 Generator 1
28	12	0x0000.0070	PWM0 Generator 2
29	13	0x0000.0074	QEI0
30	14	0x0000.0078	ADC0 Sequence 0
31	15	0x0000.007C	ADC0 Sequence 1
32	16	0x0000.0080	ADC0 Sequence 2
33	17	0x0000.0084	ADC0 Sequence 3
34	18	0x0000.0088	Watchdog Timers 0 and 1
35	19	0x0000.008C	16/32-Bit Timer 0A
36	20	0x0000.0090	16/32-Bit Timer 0B
37	21	0x0000.0094	16/32-Bit Timer 1A
38	22	0x0000.0098	16/32-Bit Timer 1B
39	23	0x0000.009C	16/32-Bit Timer 2A
40	24	0x0000.00A0	16/32-Bit Timer 2B
41	25	0x0000.00A4	Analog Comparator 0
42	26	0x0000.00A8	Analog Comparator 1
43	27	-	Reserved
44	28	0x0000.00B0	System Control

45	29	0x0000.00B4	Flash Memory Control and EEPROM Control
46	30	0x0000.00B8	GPIO Port F
47-48	31-32	-	Reserved
49	33	0x0000.00C4	UART2
50	34	0x0000.00C8	SSI1
51	35	0x0000.00CC	16/32-Bit Timer 3A
52	36	0x0000.00D0	16/32-Bit Timer 3B
53	37	0x0000.00D4	I²C1
54	38	0x0000.00D8	QEI1
55	39	0x0000.00DC	CAN0
56	40	0x0000.00E0	CAN1
57-58	41-42	-	Reserved
59	43	0x0000.00EC	Hibernation Module
60	44	0x0000.00F0	USB
61	45	0x0000.00F4	PWM Generator 3
62	46	0x0000.00F8	µDMA Software
63	47	0x0000.00FC	µDMA Error
64	48	0x0000.0100	ADC1 Sequence 0
65	49	0x0000.0104	ADC1 Sequence 1
66	50	0x0000.0108	ADC1 Sequence 2
67	51	0x0000.010C	ADC1 Sequence 3
68-72	52-56	-	Reserved
73	57	0x0000.0124	SSI2
74	58	0x0000.0128	SSI3
75	59	0x0000.012C	UART3
76	60	0x0000.0130	UART4
77	61	0x0000.0134	UART5
78	62	0x0000.0138	UART6
79	63	0x0000.013C	UART7
80-83	64-67	0x0000.0140 - 0x0000.014C	Reserved
84	68	0x0000.0150	I²C2
85	69	0x0000.0154	I²C3
86	70	0x0000.0158	16/32-Bit Timer 4A
87	71	0x0000.015C	16/32-Bit Timer 4B
88-107	72-91	0x0000.0160 - 0x0000.01AC	Reserved
108	92	0x0000.01B0	16/32-Bit Timer 5A
109	93	0x0000.01B4	16/32-Bit Timer 5B
110	94	0x0000.01B8	32/64-Bit Timer 0A
111	95	0x0000.01BC	32/64-Bit Timer 0B
112	96	0x0000.01C0	32/64-Bit Timer 1A
113	97	0x0000.01C4	32/64-Bit Timer 1B
114	98	0x0000.01C8	32/64-Bit Timer 2A
115	99	0x0000.01CC	32/64-Bit Timer 2B
116	100	0x0000.01D0	32/64-Bit Timer 3A
117	101	0x0000.01D4	32/64-Bit Timer 3B
118	102	0x0000.01D8	32/64-Bit Timer 4A
119	103	0x0000.01DC	32/64-Bit Timer 4B
120	104	0x0000.01E0	32/64-Bit Timer 5A
121	105	0x0000.01E4	32/64-Bit Timer 5B
122	106	0x0000.01E8	System Exception (imprecise)
123-149	107-133	-	Reserved
150	134	0x0000.0258	PWM1 Generator 0
151	135	0x0000.025C	PWM1 Generator 1
152	136	0x0000.0260	PWM1 Generator 2
153	137	0x0000.0264	PWM1 Generator 3
154	138	0x0000.0268	PWM1 Fault

Table 5.2 Interrupts.

Exception States

Each exception is in one of the following states:

- Inactive. The exception is not active and not pending.
- Pending. The exception is waiting to be serviced by the processor. An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.
- Active. An exception that is being serviced by the processor but has not completed. Note: An exception handler can interrupt the execution of another exception handler. In this case, both exceptions are in the active state.
- Active and Pending. The exception is being serviced by the processor, and there is a pending inerrupt.

Interrupt priority

Most embedded systems will have multiple interrupts active. So what happens when the ISR of one peripheral device is being executed and another interrupt occurs? The answer to that depends on the priority of each ISR. In Cortex-M4 processors, the NVIC supports up to 240 interrupt inputs, with 8 up to 256 programmable priority levels. The lower the interrupt number, the higher priority. If an ISR is already running and a higher priority interrupt occurs, the lower priority ISR is interrupted and the higher priority ISR begins to execute. When the higher priority ISR finishes, the remainder of the lower ISR will then complete. If the second interrupt is of the same priority or lower priority, the active ISR will complete and then the second interrupt's ISR is executed. Notice that the second ISR is not executed immediately, but it does in fact get executed.

Interrupt latency

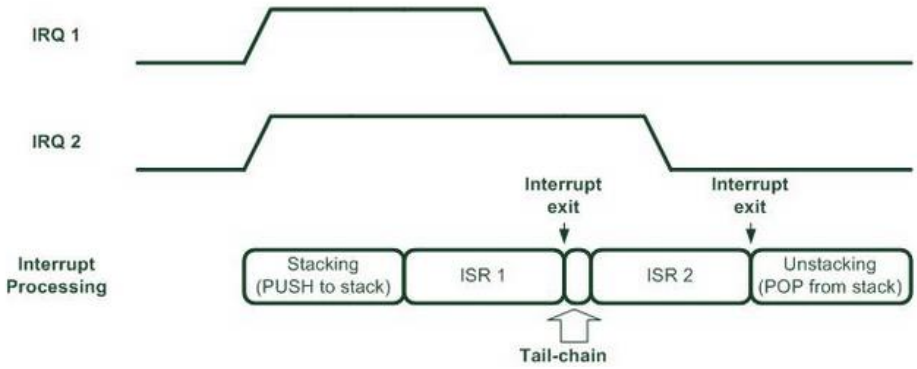
Interrupt latency is one of the key characteristics of a microcontroller and is crucial for many applications with real time requirements. It refers to the number of clock cycles required for a processor to respond to an interrupt request, this is typically a measure based on the number of clock cycles between the assertion of the interrupt request up to the cycle where the first instruction of the interrupt handler is executed. In many processors, the exact interrupt latency depends on what the processor is executing at the time the interrupt occurs. For example, in many processor architectures, the processor starts to respond to an interrupt request only when the current executing instruction completes, which can add a number of extra clock cycles. As a result, the interrupt latency value can contain a best case and a worst case value. This variation can results in jitters of interrupt responses, which could be problematic in certain applications like audio processing (with the introduction of signal distortions) and motor control (which can result in harmonics or vibrations).

Ideally, a processor should have the following characteristics:

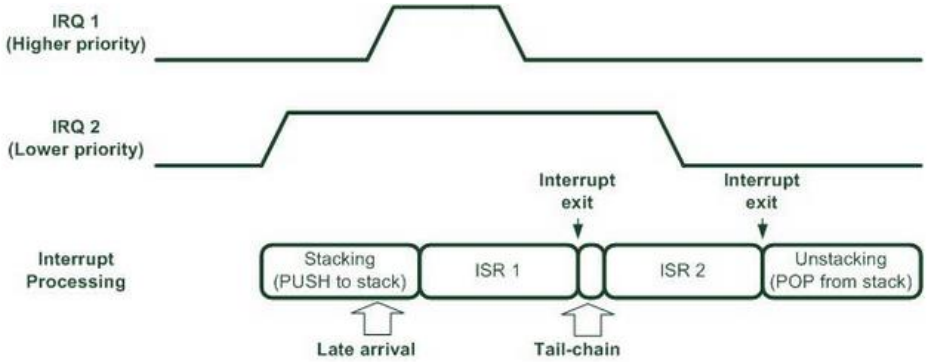
- The interrupt latency should be low
- The interrupt response is determinstic and low jitter
- The interrupt handler takes as short a time to execute as possible

The interrupt latency of all of the Cortex-M processors is extremely low. For Cortex-M4, the latency count, i.e. the exact number of cycles from the assertion of the interrupt request up to the cycle where the first instruction of the interrupt handler is ready to be executed is 12 cycles. The Cortex-M processors also incorporate some additional optimizations during interrupt handling to reduce overheads even further:

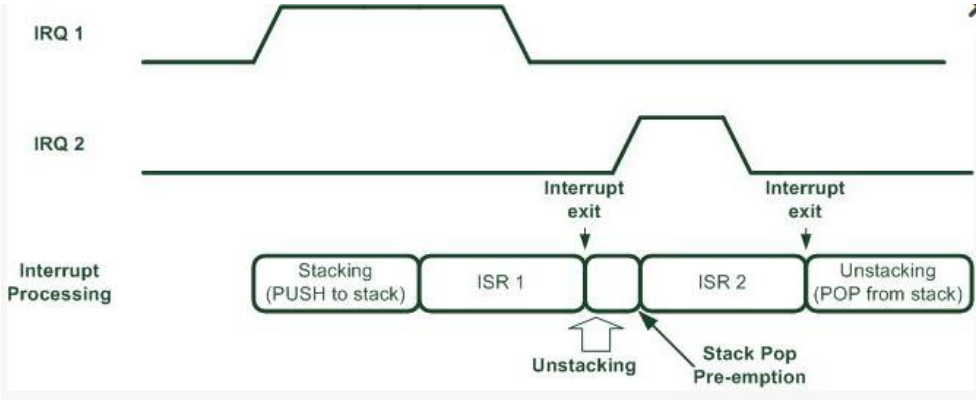
- **Tail chaining:** When an ISR is completed, and if there is another ISR waiting to be served, the processor will switch to the other ISR as soon as possible by skipping some of the un-stacking and stacking operations which are normally needed. This is called Tail Chaining, and can be just 6 cycles in the Cortex-M4 processors. This also makes the processor much more energy efficient by avoiding unnecessary memory accesses.



- **Late Arrival:**If a high priority interrupt request arrives during the stacking stage of a lower priority interrupt, the high priority interrupt will always be serviced first. This ensures high priority interrupts are serviced quickly, and avoids another level of stacking operation during the nested interrupt handling process. In addition this will save energy on power consumption (due to less access to memory) and less stack space too.



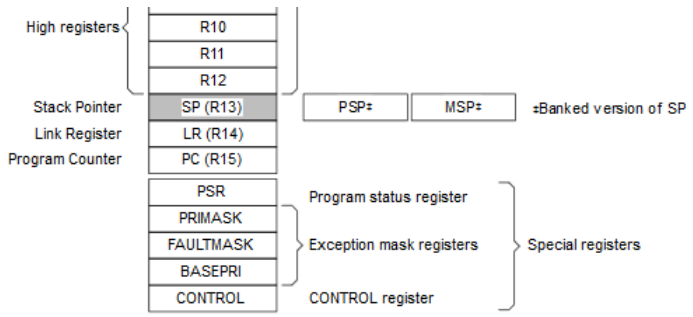
- **Pop pre-emption:** If an interrupt request arrives just as another ISR exiting and the un-stacking process is underway, the un-stacking sequence is stopped and the ISR for the new interrupt is entered as soon as possible. Again, this avoids unnecessary un-stacking and stacking, and reduces power consumption and latency.



3.2. Processing interrupts in ARM Cortex-M4

In order for the processor to recognize an interrupt request form a peripheral, the peripheral and processor must be initialized and configured properly to enable the interrupt triggering mechanism. In general, the following conditions must be true for an interrupt to occur:

- The peripheral is configured properly. This step varies based on the peripheral.
- The interrupt enable bit for the peripheral in the Interrupt Set-Enable Registers (ISERnregisters) is set. By default, all interrupt enable bits are cleared (disabled).
- The priority level for the peripheral is configured properly in the Interrupt Priority Registers (IPRn registers). In order for this interrupt to occur, this priority level must be higher than or the same as the priority level set in BASEPRI register. By default, the priority level is 0 in BASEPRI register. Also, by default the priority level is zero for all peripherals.It means that default values for these registers are ok to use in your program without any modifications.
- The global interrupt bit is enabled (bit 0 in PRIMASK register= 0). By default bit 0 of PRIMASK is 0 (enabled).
- The peripheral asserts the interrupt request which sets the interrupt flag.



The BASEPRI register (b0-b7) defines the minimum priority for exception processing. When BASEPRI is set to a nonzero value, it prevents the activation of all exceptions with the same or lower priority level as the BASEPRI value.

The PRIMASK register prevents activation of all exceptions with configurable priority.

Figure 5.3. Exception mask registers.

Address	Name	Type	Required privilege	Reset value	Description
0xE000E100 - 0xE000E11C	NVIC_ISER0-NVIC_ISER7	RW	Privileged	0x00000000	*Interrupt Set-enable Registers*
0XE000E180-0xE000E19C	NVIC_ICER0-NVIC_ICER7	RW	Privileged	0x00000000	*Interrupt Clear-enable Registers*
0XE000E200-0xE000E21C	NVIC_ISPR0-NVIC_ISPR7	RW	Privileged	0x00000000	*Interrupt Set-pending Registers*
0XE000E280-0xE000E29C	NVIC_ICPR0-NVIC_ICPR7	RW	Privileged	0x00000000	*Interrupt Clear-pending Registers*
0xE000E300-0xE000E31C	NVIC_IABR0-NVIC_IABR7	RW	Privileged	0x00000000	*Interrupt Active Bit Registers*
0xE000E400-0xE000E4EF	NVIC_IPR0-NVIC_IPR59	RW	Privileged	0x00000000	*Interrupt Priority Registers*

Figure 5.4. Interrupt control and status registers.

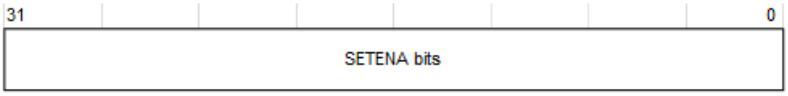
Interrupt set enable register:

Address	Name	Type	Reset Value	Description
0xE000E100	SETENA0	R/W	0	Enable for external interrupt #0–31 bit[0] for interrupt #0 (exception #16) bit[1] for interrupt #1 (exception #17) ... bit[31] for inte Write 1 to set t Read value ind
0xE000E104	SETENA1	R/W	0	Enable for exte Write 1 to set t Read value ind
0xE000E108	SETENA2	R/W	0	Enable for exte Write 1 to set t Read value ind
...	-	-	-	-

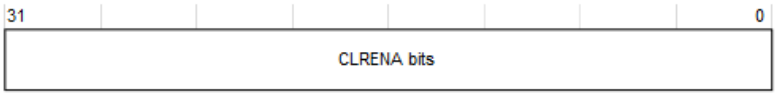
Interrupt clear enable register:

0xE000E180	CLRENA0	R/W	0	Clear enable for external interrupt #0–31 bit[0] for interrupt #0 bit[1] for interrupt #1 ... bit[31] for interrupt #31 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current enable status
0xE000E184	CLRENA1	R/W	0	Clear Enable for external interrupt #32–63 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current enable status
0xE000E188	CLRENA2	R/W	0	Clear enable for external interrupt #64–95 Write 1 to clear bit to 0; write 0 has no effect Read value indicates the current enable status

Interrupt set enable registers ISER0-ISER7 (0xE000E100 – 0xE000E11C):

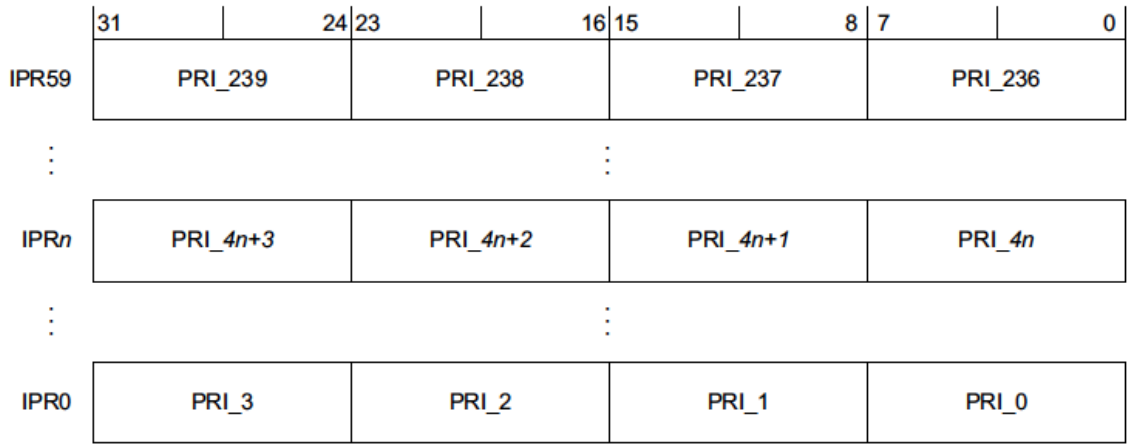


Interrupt clear enable registers ICER0-ICER7 (0xE000E180 – 0xE000E19C):



Interrupt priority level registers IPRn (0xE000E400 – 0xE000E4EF):

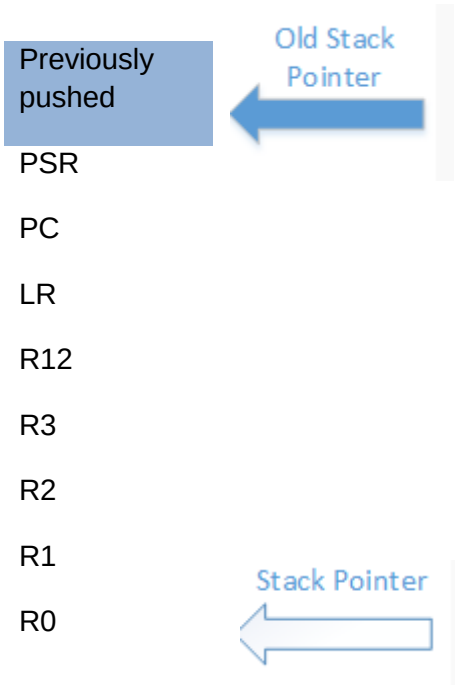
8-bit priority field for each interrupt and each register holds four priority fields.



Once an interrupt request has been asserted by the peripheral and recognized by the processor, the processor needs to service the peripheral which causes the following conditions:

1: Suspension of the main program

- Current instruction is completed
- Suspend execution and push 8 registers (R0-R3, R12, LR, PC, PSR) on to the stack as in the following figure.



- LR set to the value showing which Sp is used to restore register from when exiting an interrupt
  - Two SPs: Main SP (MSP) and Process SP (PSP)
  - Determined by operating mode, and CONTROL[0]
  - Thread mode ⟶ SP = PSP
  - Handler mode ⟶ SP = MSP if CONTROL[0] = 0; Otherwise SP = PSP
    - If LR = 0xFFFFFFF9, then SP = MSP
    - If LR = 0xFFFFFFFD, then SP = PSP
- IPSR (Interrupt PSR[b0:b7]) set to interrupt number
- PC set to ISR address

2: The ISR is executed

- Process the interrupt request by the peripheral
- Clears the flag that requested the interrupt
- Exit ISR by executing BX LR (other than bx lr, pop-ldr can also be used as shown below)

Return Instruction	Description
BX <*reg*>	If the EXC_RETURN value is still in LR, we can use the *BX LR* instruction to perform the interrupt return.
POP {PC}, or POP {...., PC}	Very often the value of LR is pushed to the stack after entering the exception handler. We can use the POP instruction, either a single POP or multiple POPs, to put the EXC_RETURN value to the program counter. This will cause the processor to perform the interrupt return.
LDR, or LDM	It is possible to produce an interrupt return using the LDR instruction with PC as the destination register.

3: Resume normal operation

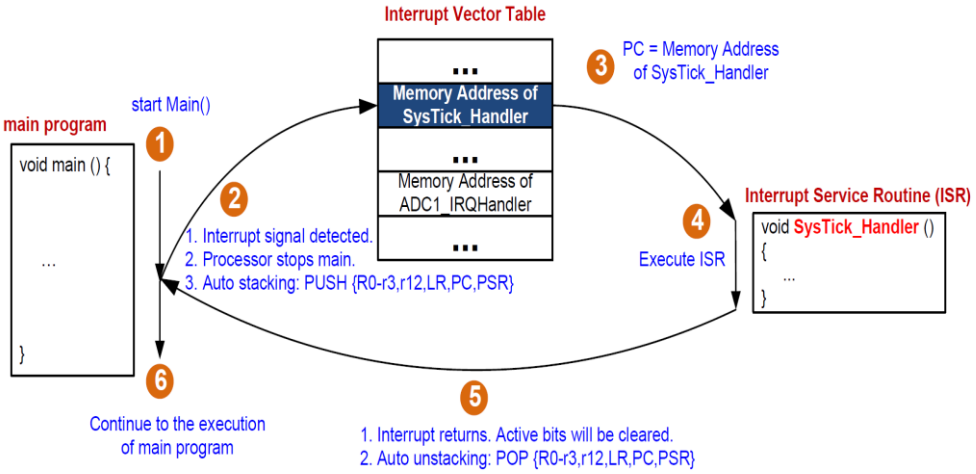
- Pulls 8 registers (R0-R3, R12, LR, PC, PCR) from the stack
- Return to the next instruction in the previously suspended task.

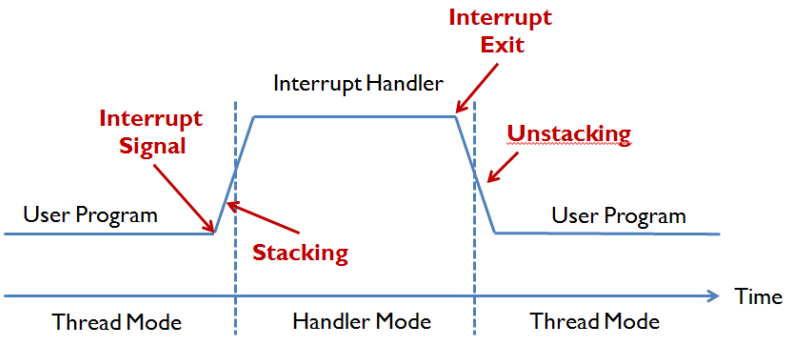
Note that the ISR can reside anywhere in the code memory (ROM), so how do we manage the memory space so that the processor can point to the right address for the correct ISR? From a programming perspective, we can write the ISR as regular subroutine but with specific names. These predefined ISR names are provided in the startup_TM4C123.s file provided with the Keil µvision software. A part of the startup_TM4C123.s showing some of the predefined ISR names is shown below.

```
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors
                EXPORT  __Vectors_End
                EXPORT  __Vectors_Size

__Vectors       DCD     __initial_sp              ; Top of Stack
                DCD     Reset_Handler             ; Reset Handler
                DCD     NMI_Handler               ; NMI Handler
                DCD     HardFault_Handler         ; Hard Fault Handler
                DCD     MemManage_Handler         ; MPU Fault Handler
                DCD     BusFault_Handler          ; Bus Fault Handler
                DCD     UsageFault_Handler        ; Usage Fault Handler
```

For example, if we want to create an ISR for RESET, then the name for the ISR should be Reset_Handler. The assembler will place the memory address of this ISR in the appropriate address in the vector table (0x00000004 in this case), i.e. if the following ISR for RESET starts at memory location 0x00000274, [0x00000004]=0x00000274 so that [pc]=0x00000274 after the interrupt.

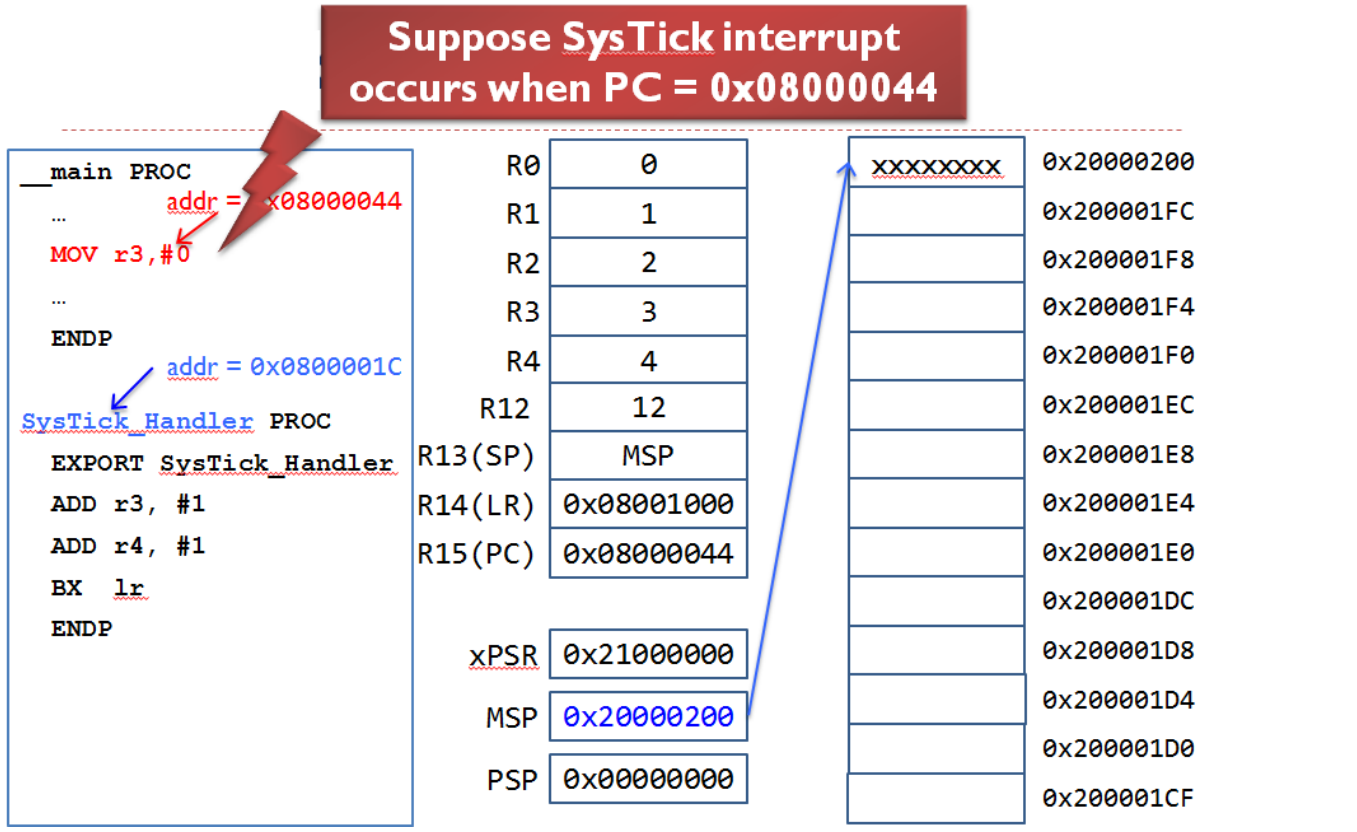
Interrupt loop



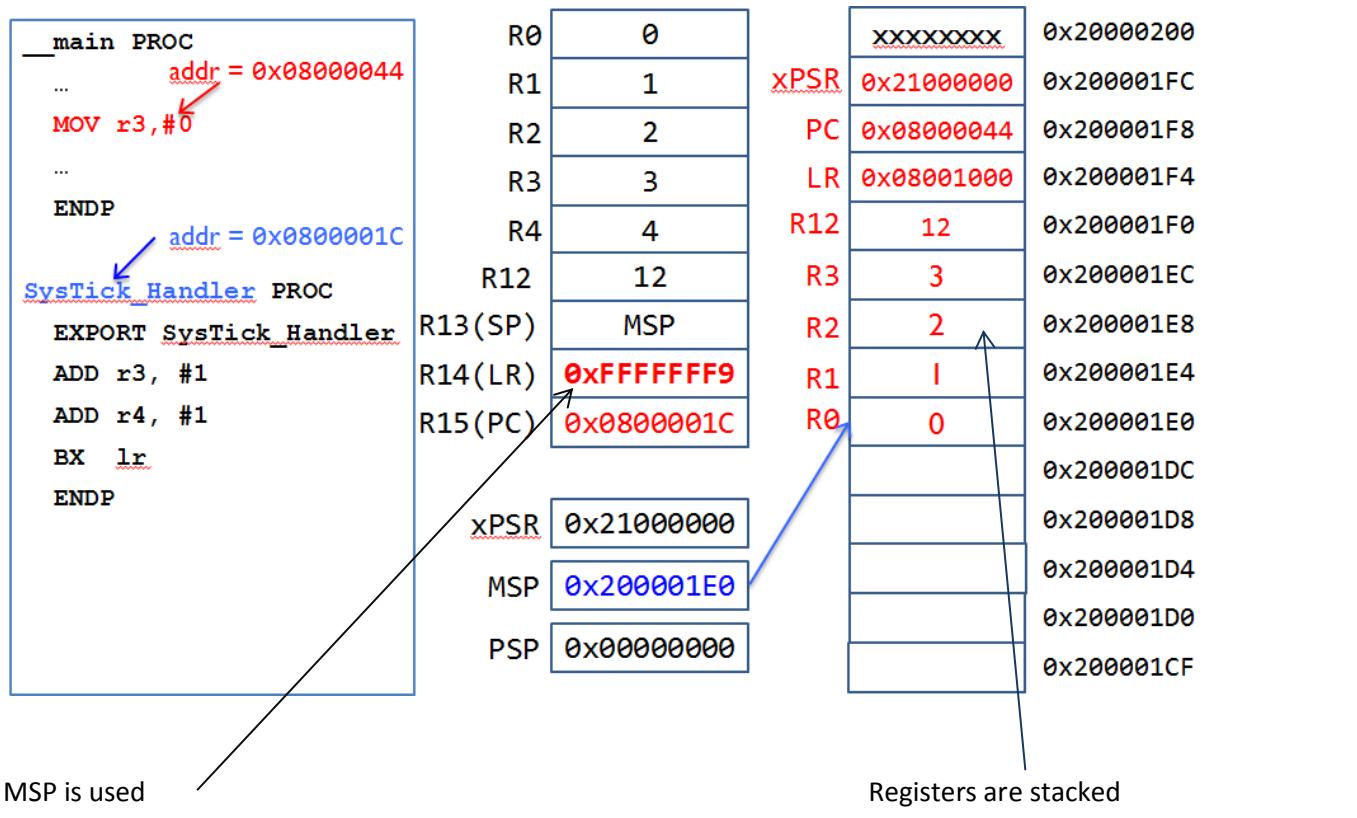


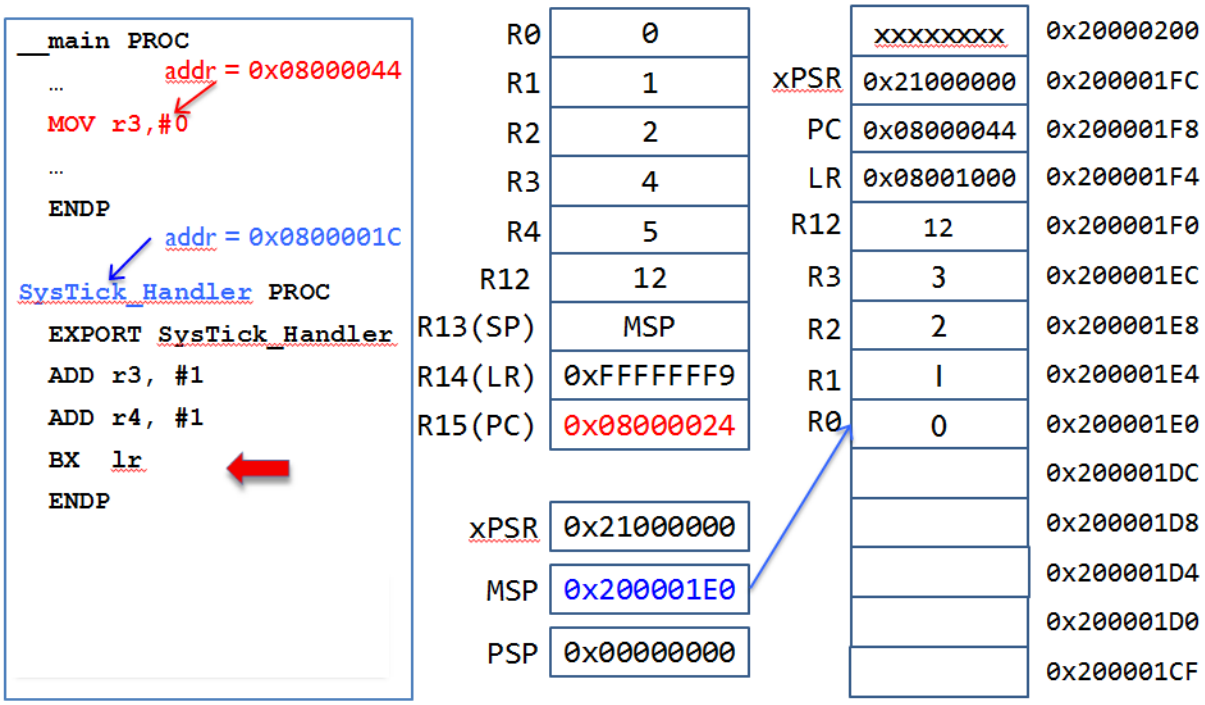
Thread mode

Ex:

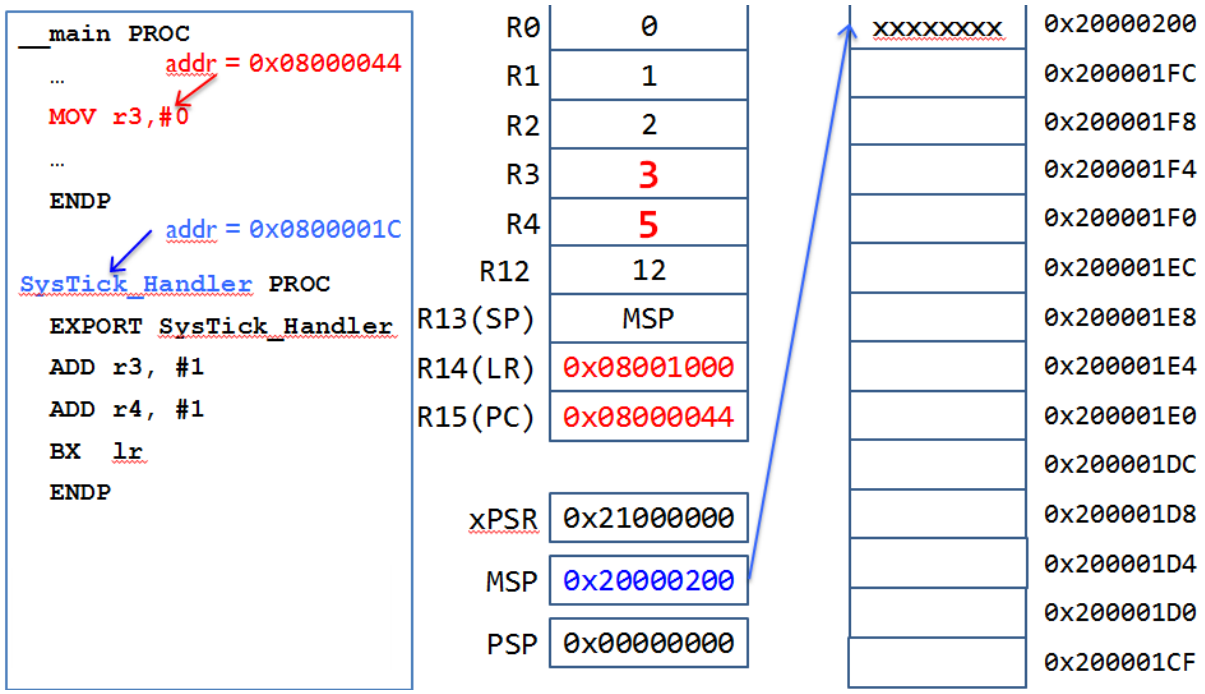


Handler mode





Exception is served.



Registers are unstacked and the main program continues.

Chapter 6

General Purpose Input/Output (GPIO)

1. I/O interfacing

One of the primary features of a microcontroller is to interact with and control external devices. These devices may be something as simple as an LED or a push button or might be complicated as a motor. In order to interface with such devices, almost all microcontrollers have dedicated pins and registers (interfaces) that can be controlled by software. This basic digital interface is called the General Purpose Input/Output port or GPIO port. In other words, a GPIO port is a collection of external pins on a microcontroller that can be configured to act as either an input or an output, digital or analog. A simple bi-directional pin and a typical I/O pin of TM4C123G is shown in Figures 6.1 and 6.2.

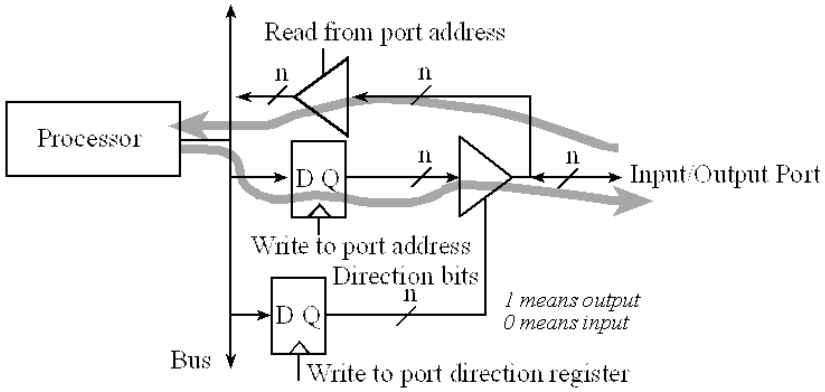


Figure 6.1. Bi-directional I/O pin

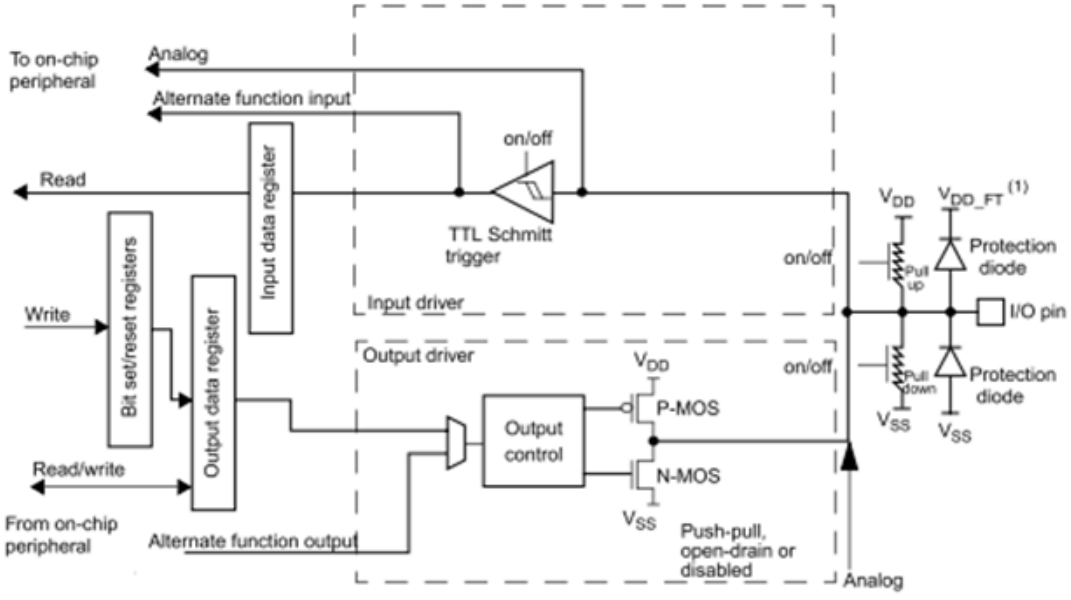


Figure 1-1.2. A typical I/O pin of TM4C123G

Each GPIO port has multiple registers (data and control) that can be modified to change the behavior of the GPIO port. The most basic ones are the direction and data registers. The direction register is used to control if a specific GPIO pin is an input or an output. After the configuration of the GPIO port is completed, the application can issue a read command that reads the voltage levels on a physical pin. If the GPIO pin is configured as an output, the application can write data to the GPIO port and the corresponding GPIO pins will be driven to the specified voltage level. Reading and writing the GPIO pins is done through what is commonly called the data register. Internally, the processor communicates with ports as normal memory addresses, uses the same instructions and normal address, data and control busses are used to control (memory mapped I/O).

The regular function of an I/O pin is to perform parallel I/O. Most pins, however, have an alternative function. TM4C123G microcontrollers have a wide range of alternative functions:

- UART — Universal asynchronous receiver/transmitter
- SSI (SPI) — Synchronous serial interface
- I2C — Inter-integrated circuit
- Timer — Periodic interrupts, input capture, and output compare
- PWM — Pulse width modulation
- ADC — Analog to digital converter, measure analog signals
- Analog Comparator — Compare two analog signals
- QEI — Quadrature encoder interface
- USB — Universal serial bus
- Ethernet — High-speed network
- CAN — Controller area network

The UART can be used for serial communication between computers. It is asynchronous and allows for simultaneous communication in both directions. The synchronous serial interface (SSI) is alternately called serial peripheral interface (SPI). It is used to interface medium-speed I/O devices. One could use SSI to interface a digital to analog converter (DAC) or a secure digital card (SDC). Inter interface communication (I2C) is a simple I/O bus that we will use to interface low speed peripheral devices. Input capture and output compare of timer can be used to create periodic interrupts and measure period, pulse width, phase, and frequency. Pulse width modulation (PWM) outputs can be used to apply variable power to motor interfaces. In a typical motor controller, input capture measures rotational speed, and PWM controls power. The analog to digital converter (ADC) is used to measure the amplitude of analog signals and is important in data acquisition systems. The analog comparator takes two analog inputs and produces a digital output depending on which analog input is greater. The QEI can be used to interface a brushless DC motor. USB is a high-speed serial communication channel. The Ethernet port can be used to bridge the microcontroller to the Internet or a local area network. The CAN creates a high-speed communication channel between microcontrollers and is commonly found in automotive and other distributed control applications. The I/O port structure for the TM4C123G is shown in Figure 6.3.

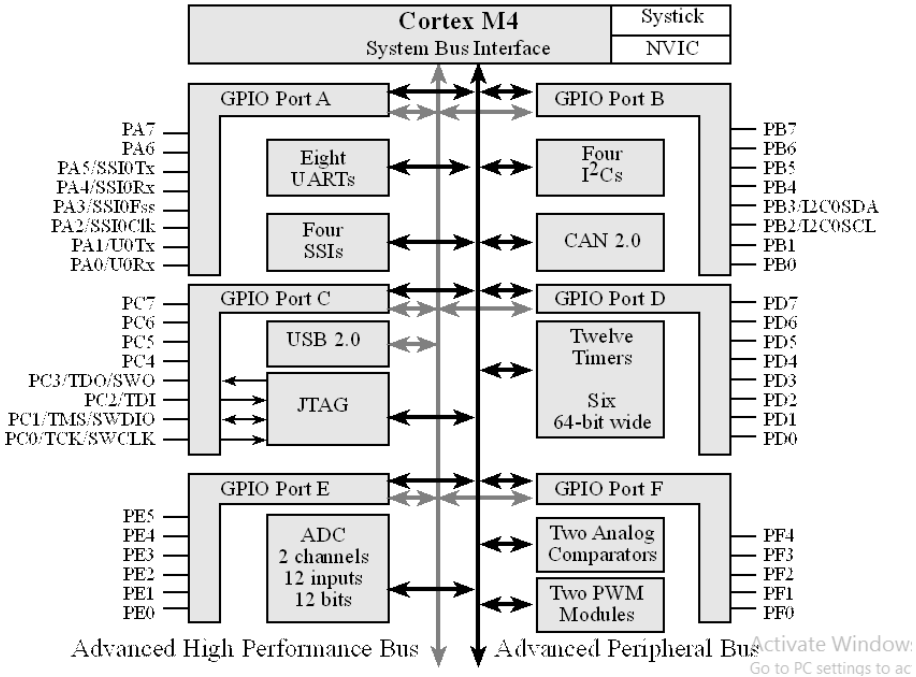


Figure 6.3: GPIO ports and other interfaces.

The GPIO module is composed of six physical GPIO blocks, each corresponding to an individual GPIO port (Port A, Port B, Port C, Port D, Port E, Port F). The GPIO module supports up to 43 programmable input/output pins, depending on the peripherals being used. Highly flexible pin multiplexing allows use as GPIO or one of several peripheral functions. Ports A,B,C and D have 8 pins, port E has 6 pins and port F has 5 pins however Pins PC3 – PC0 are reserved for the JTAG debugger and should not be used for regular I/O. Pins can be configured for digital I/O, analog input, timer I/O, or serial I/O. For example PA0 can be digital I/O or serial input. There are two buses used for I/O. The digital I/O ports are connected to both the advanced peripheral bus and the advanced high-performance bus. Because of the multiple buses, the microcontroller can perform I/O bus cycles simultaneous with instruction fetches from flash ROM. The TM4C123GH6PM has eight UART ports, four SSI ports, four I2C ports, two 12-bit ADCs, twelve timers, a CAN port, a USB interface and 16 PWM outputs. Each pin has one configuration bit in the GPIO analog mode select (GPIOAMSEL) register. We set this bit to connect the port pin to the ADC or analog comparator. For digital functions, each pin also has four bits in the GPIO control (GPIOPCTL) register, which we set to specify the alternative function for that pin (0 means regular I/O port). Not every pin can be connected to every alternative function. Table 1 lists the alternate names of the port pins.

Table 1 GPIO pins and alternate functions

IO	Pin	Analog Function	Digital Function (GPIOPCTL PMCx Bit Field Encoding)[a]										
			1	2	3	4	5	6	7	8	9	14	15
PA0	17	-	U0Rx	-	-	-	-	-	-	CAN1Rx	-	-	-
PA1	18	-	U0Tx	-	-	-	-	-	-	CAN1Tx	-	-	-
PA2	19	-	-	SSI0Clk	-	-	-	-	-	-	-	-	-
PA3	20	-	-	SSI0Fss	-	-	-	-	-	-	-	-	-
PA4	21	-	-	SSI0Rx	-	-	-	-	-	-	-	-	-
PA5	22	-	-	SSI0Tx	-	-	-	-	-	-	-	-	-
PA6	23	-	-	-	I2C1SCL	-	M1PWM2	-	-	-	-	-	-
PA7	24	-	-	-	I2C1SDA	-	M1PWM3	-	-	-	-	-	-
PB0	45	USB0ID	U1Rx	-	-	-	-	-	T2CCP0	-	-	-	-
PB1	46	USB0VBUS	U1Tx	-	-	-	-	-	T2CCP1	-	-	-	-
PB2	47	-	-	-	I2C0SCL	-	-	-	T3CCP0	-	-	-	-
PB3	48	-	-	-	I2C0SDA	-	-	-	T3CCP1	-	-	-	-
PB4	58	AIN10	-	SSI2Clk	-	M0PWM2	-	-	T1CCP0	CAN0Rx	-	-	-
PB5	57	AIN11	-	SSI2Fss	-	M0PWM3	-	-	T1CCP1	CAN0Tx	-	-	-
PB6	1	-	-	SSI2Rx	-	M0PWM0	-	-	T0CCP0	-	-	-	-
PB7	4	-	-	SSI2Tx	-	M0PWM1	-	-	T0CCP1	-	-	-	-
PC0	52	-	TCK SWCLK	-	-	-	-	-	T4CCP0	-	-	-	-
PC1	51	-	TMS SWDIO	-	-	-	-	-	T4CCP1	-	-	-	-
PC2	50	-	TDI	-	-	-	-	-	T5CCP0	-	-	-	-
PC3	49	-	TDO SWO	-	-	-	-	-	T5CCP1	-	-	-	-
PC4	16	C1-	U4Rx	U1Rx	-	M0PWM6	-	IDX1	WT0CCP0	U1RTS	-	-	-
PC5	15	C1+	U4Tx	U1Tx	-	M0PWM7	-	PhA1	WT0CCP1	U1CTS	-	-	-
PC6	14	C0+	U3Rx	-	-	-	-	PhB1	WT1CCP0	USB0EPEN	-	-	-
PC7	13	C0-	U3Tx	-	-	-	-	-	WT1CCP1	USB0PFLT	-	-	-
PD0	61	AIN7	SSI3Clk	SSI1Clk	I2C3SCL	M0PWM6	M1PWM0	-	WT2CCP0	-	-	-	-
PD1	62	AIN6	SSI3Fss	SSI1Fss	I2C3SDA	M0PWM7	M1PWM1	-	WT2CCP1	-	-	-	-
PD2	63	AIN5	SSI3Rx	SSI1Rx	-	M0FAULT0	-	-	WT3CCP0	USB0EPEN	-	-	-
PD3	64	AIN4	SSI3Tx	SSI1Tx	-	-	-	IDX0	WT3CCP1	USB0PFLT	-	-	-
PD4	43	USB0DM	U6Rx	-	-	-	-	-	WT4CCP0	-	-	-	-
PD5	44	USB0DP	U6Tx	-	-	-	-	-	WT4CCP1	-	-	-	-
PD6	53	-	U2Rx	-	-	M0FAULT0	-	PhA0	WT5CCP0	-	-	-	-
PD7	10	-	U2Tx	-	-	-	-	PhB0	WT5CCP1	NMI	-	-	-
PE0	9	AIN3	U7Rx	-	-	-	-	-	-	-	-	-	-
PE1	8	AIN2	U7Tx	-	-	-	-	-	-	-	-	-	-
PE2	7	AIN1	-	-	-	-	-	-	-	-	-	-	-
PE3	6	AIN0	-	-	-	-	-	-	-	-	-	-	-
PE4	59	AIN9	U5Rx	-	I2C2SCL	M0PWM4	M1PWM2	-	-	CAN0Rx	-	-	-
PE5	60	AIN8	U5Tx	-	I2C2SDA	M0PWM5	M1PWM3	-	-	CAN0Tx	-	-	-
PF0	28	-	U1RTS	SSI1Rx	CAN0Rx	-	M1PWM4	PhA0	T0CCP0	NMI	C0o	-	-
PF1	29	-	U1CTS	SSI1Tx	-	-	M1PWM5	PhB0	T0CCP1	-	C1o	TRD1	-
PF2	30	-	-	SSI1Clk	-	M0FAULT0	M1PWM6	-	T1CCP0	-	-	TRD0	-
PF3	31	-	-	SSI1Fss	CAN0Tx	-	M1PWM7	-	T1CCP1	-	-	TRCLK	-
PF4	5	-	-	-	-	-	M1FAULT0	IDX0	T2CCP0	USB0EPEN	-	-	-

A clock to each GPIO port is turned off by default to save power and must be turned on to use that port. The Run Clock Gate Control Register for GPIO (RCGCGPIO) (address is 0x400FE608) is used to turn on/off the clock to each GPIO port independently.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved										R5	R4	R3	R2	R1	R0
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
5	R5	RW	0	GPIO Port F Run Mode Clock Gating Control Value Description 0 GPIO Port F is disabled. 1 Enable and provide a clock to GPIO Port F in Run mode.
. . .				
0	R0	RW	0	GPIO Port A Run Mode Clock Gating Control Value Description 0 GPIO Port A is disabled. 1 Enable and provide a clock to GPIO Port A in Run mode.

2. Registers

Each GPIO port is a separate hardware instantiation of the same physical block (Figure 6.4). The TM4C123G microcontroller contains six ports and thus six of these physical GPIO blocks. Note that not all pins are implemented on every block.

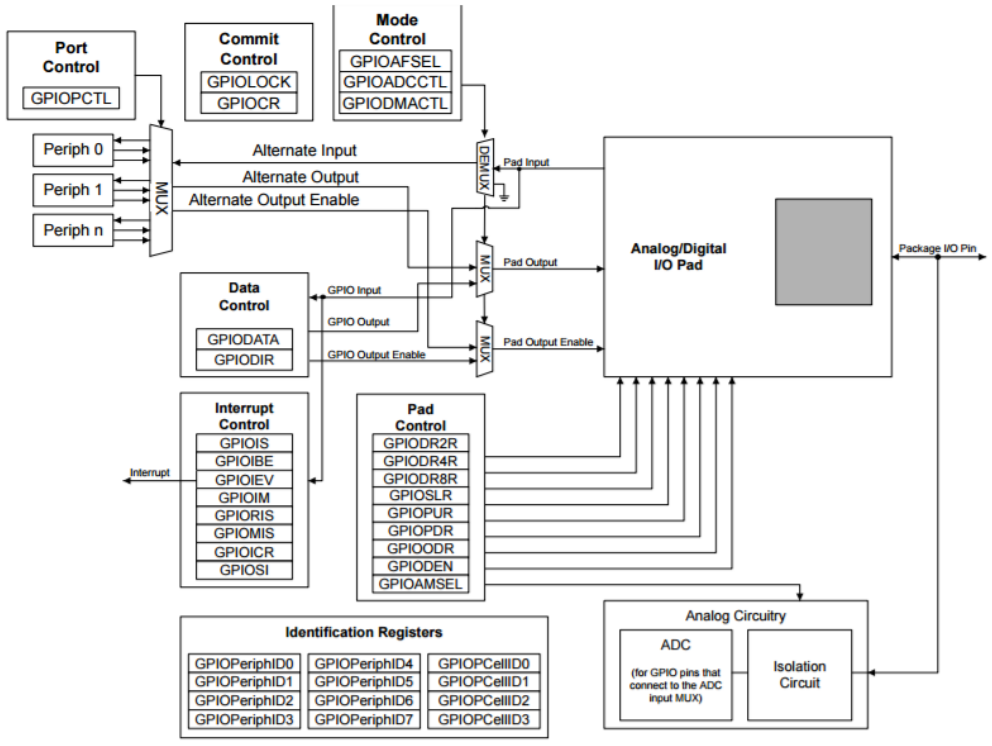


Figure 6.4: A port description

Each GPIO port has 36 registers. Each port has a base address and each register related to that port is assigned an offset. The same type of registers have the same offset value and can be reached relative to base address of their port. Base addresses are as follows

Port A: 0x40004000

Port B: 0x40005000

Port C: 0x40006000

Port D: 0x40007000

Port E: 0x40024000

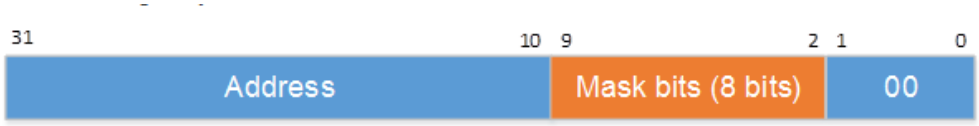
Port F: 0x40025000

For example, GPIOPUR register has the offset 0x510 so that GPIOPUR register on port D is at address 0x40007510 (0x40007000+0x510). Usually least significant eight bits of the registers are used and other bits are reserved.

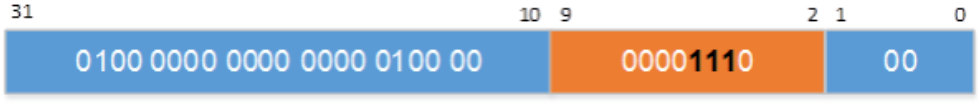
1. Data Register: GPIODATA, offset 0x000

The values written in the GPIODATA register are transferred onto the GPIO port pins if the respective pins have been configured as outputs through the GPIO Direction (GPIODIR) register. A read from GPIODATA returns the last bit value written if the respective pins are configured as outputs, or it returns the value on the corresponding input pin when these are configured as inputs.

To aid in the efficiency of software, the GPIO ports allow for the modification of individual bits in the GPIO Data (GPIODATA) register by using bits [9:2] of the address bus as a mask. In this manner, software drivers can modify individual GPIO pins in a single instruction without affecting the state of the other pins. This method is more efficient than the conventional method of performing a read-modify-write operation to set or clear an individual GPIO pin.



For example, writing to address 0x40004038 means that bits 1, 2 and 3 of port A must be changed, since the base address of port A is 0x40004000.

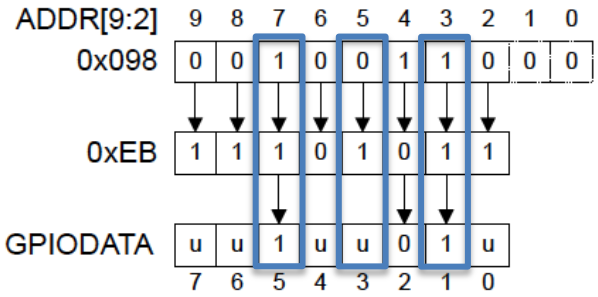


The following table help you calculate offset address for the bits of a port, to which you want to access. All these addresses are used to reach the same data register, but different bits of the data register.

If we want to access bit	Offset Constanct
7	0x200
6	0x100
5	0x080
4	0x040
3	0x020
2	0x010
1	0x008
0	0x004

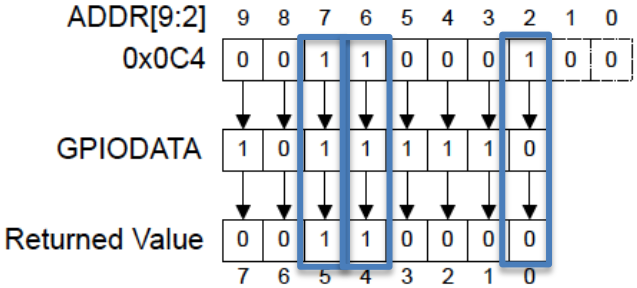
If we want to read and write all 8 bits of a port, it means that we need to sum all these 8 offset constants, which makes the offset address of 0x3FC (001111111100 in binary).

Ex: Writing a value of 0xEB to the address GPIODATA + 0x098 has the results shown in the following figure, where u indicates that data is unchanged by the write. This example demonstrates how GPIODATA bits 5, 2, and 1 are written.



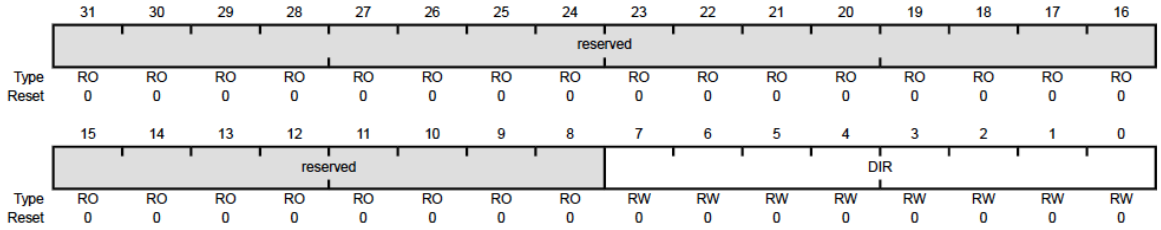
During a read, if the address bit associated with the data bit is set, the value is read. If the address bit associated with the data bit is cleared, the data bit is read as a zero, regardless of its actual value.

Ex: Reading address GPIODATA + 0x0C4 yields as shown in the following figure. This example shows how to read GPIODATA bits 5, 4, and 0.



2. Data direction register: GPIODIR, offset 0x400

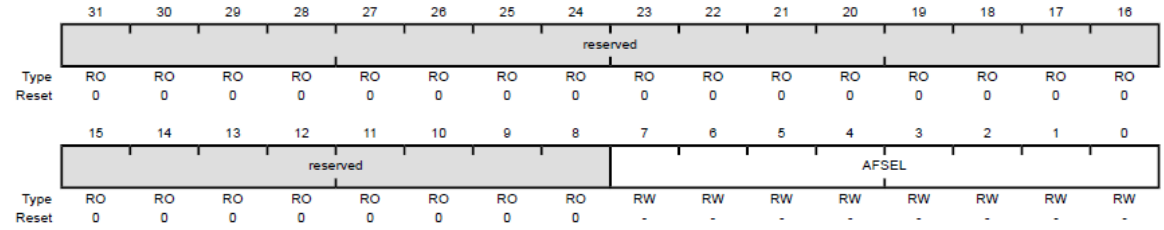
Setting a bit in the GPIODIR register configures the corresponding pin to be an output, while clearing a bit configures the corresponding pin to be an input. All bits are cleared by a reset, meaning all GPIO pins are inputs by default.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	DIR	RW	0x00	GPIO Data Direction Value Description 0 Corresponding pin is an input. 1 Corresponding pins is an output.

3. GPIO Alternate Function Select: GPIOAFSEL, offset 0x420

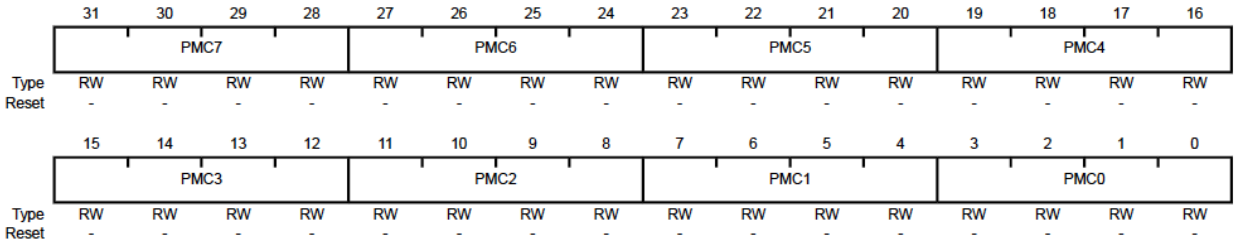
The GPIOAFSEL register is the mode control select register. If a bit is clear, the pin is used as a GPIO and is controlled by the GPIO registers. Setting a bit in this register configures the corresponding GPIO line to be controlled by an associated peripheral. Several possible peripheral functions are multiplexed on each GPIO. The GPIO Port Control (GPIOPCTL) register is used to select one of the possible functions.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	AFSEL	RW	-	GPIO Alternate Function Select Value Description 0 The associated pin functions as a GPIO and is controlled by the GPIO registers. 1 The associated pin functions as a peripheral signal and is controlled by the alternate hardware function.

4. GPIO Port Control: GPIOPCTL, offset 0x52C

The GPIOPCTL register is used in conjunction with the GPIOAFSEL register and selects the specific peripheral signal for each GPIO pin when using the alternate function mode. Most bits in the GPIOAFSEL register are cleared on reset, therefore most GPIO pins are configured as GPIOs by default. When a bit is set in the GPIOAFSEL register, the corresponding GPIO signal is controlled by an associated peripheral. The GPIOPCTL register selects one out of a set of peripheral functions for each GPIO, providing additional flexibility in signal definition. The peripheral function is selected based on the bit field encoding given in Table 1.



Bit/Field	Name	Type	Reset	Description
31:28	PMC7	RW	-	Port Mux Control 7 This field controls the configuration for GPIO pin 7.
27:24	PMC6	RW	-	Port Mux Control 6 This field controls the configuration for GPIO pin 6.
23:20	PMC5	RW	-	Port Mux Control 5 This field controls the configuration for GPIO pin 5.
19:16	PMC4	RW	-	Port Mux Control 4 This field controls the configuration for GPIO pin 4.
15:12	PMC3	RW	-	Port Mux Control 3 This field controls the configuration for GPIO pin 3.
11:8	PMC2	RW	-	Port Mux Control 2 This field controls the configuration for GPIO pin 2.
7:4	PMC1	RW	-	Port Mux Control 1 This field controls the configuration for GPIO pin 1.
3:0	PMC0	RW	-	Port Mux Control 0 This field controls the configuration for GPIO pin 0.

5. Interrupt control registers: The interrupt capabilities of each GPIO port are controlled by a set of seven registers. These registers are used to select the source of the interrupt, its polarity, and the edge properties. When one or more GPIO inputs cause an interrupt, a single interrupt output is sent to the interrupt controller for the entire GPIO port. For edge-triggered interrupts, software must clear the interrupt to enable any further interrupt to be recognized by the controller.

Three registers define the edge or sense that causes interrupts:

- GPIO Interrupt Sense (GPIOIS) register

- GPIO Interrupt Both Edges (GPIOIBE)
- GPIO Interrupt Event (GPIOIEV) register

Interrupts are enabled/disabled via the

- GPIO Interrupt Mask (GPIOIM) register

When an interrupt condition occurs, the state of the interrupt signal can be viewed in two locations:

- GPIO Raw Interrupt Status (GPIORIS) and
- GPIO Masked Interrupt Status (GPIOMIS) registers

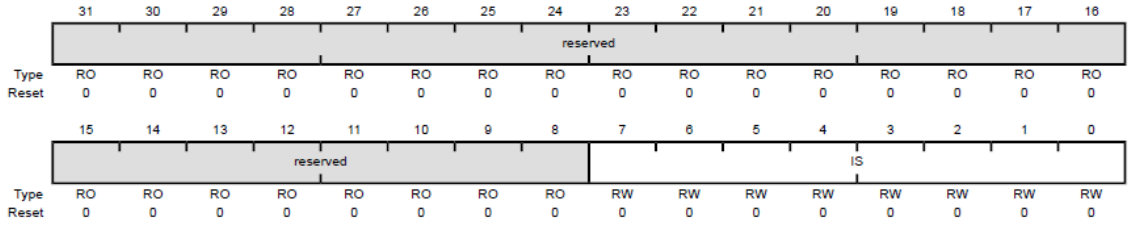
For a GPIO edge-detect interrupt, the RIS bit in the GPIORIS register is cleared by writing a '1' to the corresponding bit in the

- GPIO Interrupt Clear (GPIOICR) register

The details of each of each of these registers are given below.

GPIO Interrupt Sense (GPIOIS), offset 0x404

Setting a bit in the GPIOIS register configures the corresponding pin to detect levels, while clearing a bit configures the corresponding pin to detect edges. All bits are cleared by a reset.

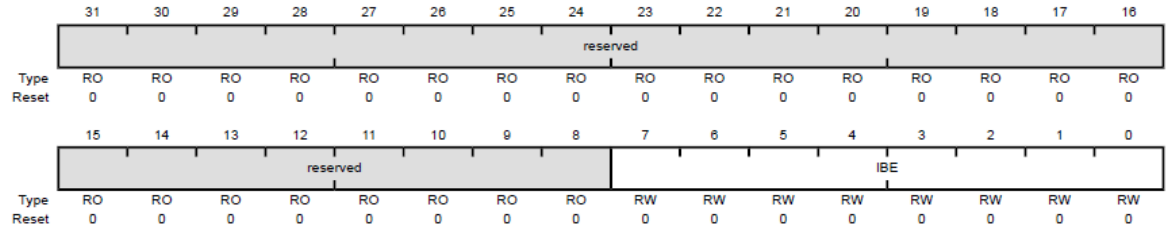


Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IS	RW	0x00	GPIO Interrupt Sense

Value	Description
0	The edge on the corresponding pin is detected (edge-sensitive).
1	The level on the corresponding pin is detected (level-sensitive).

GPIO Interrupt Both Edges (GPIOIBE), offset 0x408

The GPIOIBE register allows both edges to cause interrupts. When the corresponding bit in the GPIO Interrupt Sense (GPIOIS) register is set to detect edges, setting a bit in the GPIOIBE register configures the corresponding pin to detect both rising and falling edges, regardless of the corresponding bit in the GPIO Interrupt Event (GPIOIEV) register. Clearing a bit configures the pin to be controlled by the GPIOIEV register. All bits are cleared by a reset.

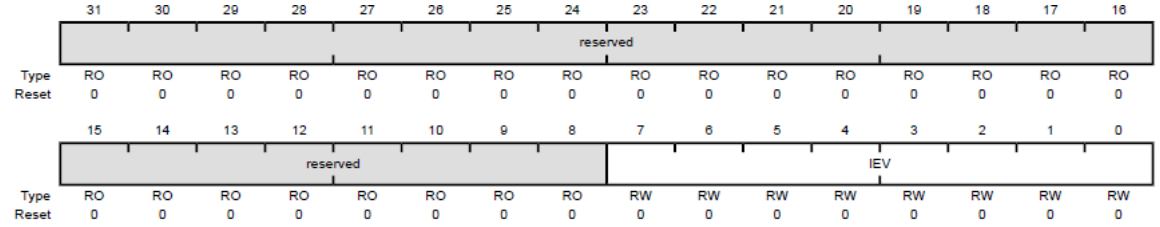


Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IBE	RW	0x00	GPIO Interrupt Both Edges

Value	Description
0	Interrupt generation is controlled by the **GPIO Interrupt Event (GPIOIEV)** register (see page 666).
1	Both edges on the corresponding pin trigger an interrupt.

GPIO Interrupt Event (GPIOIEV), offset 0x40C

Setting a bit in the GPIOIEV register configures the corresponding pin to detect rising edges or high levels, depending on the corresponding bit value in the GPIO Interrupt Sense (GPIOIS) register. Clearing a bit configures the pin to detect falling edges or low levels, depending on the corresponding bit value in the GPIOIS register. All bits are cleared by a reset.

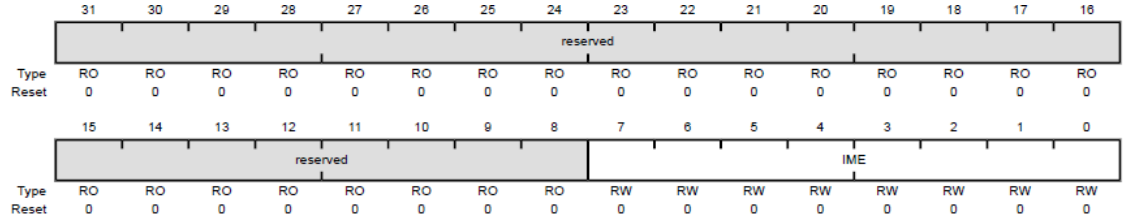


Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IEV	RW	0x00	GPIO Interrupt Event

Value	Description
0	A falling edge or a Low level on the corresponding pin triggers an interrupt.
1	A rising edge or a High level on the corresponding pin triggers an interrupt.

GPIO Interrupt Mask (GPIOIM), offset 0x410

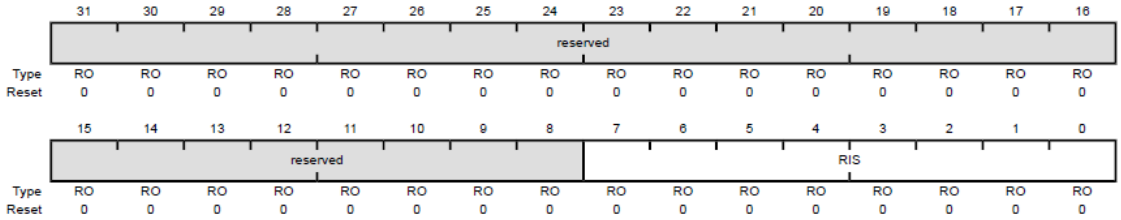
Setting a bit in the GPIOIM register allows interrupts that are generated by the corresponding pin to be sent to the interrupt controller on the combined interrupt signal. Clearing a bit prevents an interrupt on the corresponding pin from being sent to the interrupt controller. All bits are cleared by a reset.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IME	RW	0x00	GPIO Interrupt Mask Enable Value Description 0 The interrupt from the corresponding pin is masked. 1 The interrupt from the corresponding pin is sent to the interrupt controller.

GPIO Raw Interrupt Status (GPIORIS), offset 0x414 (read only)

A bit in this register is set when an interrupt condition occurs on the corresponding GPIO pin. If the corresponding bit in the GPIO Interrupt Mask (GPIOIM) register is set, the interrupt is sent to the interrupt controller. Bits read as zero indicate that corresponding input pins have not initiated an interrupt. For a GPIO level-detect interrupt, the interrupt signal generating the interrupt must be held until serviced. Once the input signal deasserts from the interrupt generating logical sense, the corresponding RIS bit in the GPIORIS register clears. For a GPIO edge-detect interrupt, the RIS bit in the GPIORIS register is cleared by writing a '1' to the corresponding bit in the GPIO Interrupt Clear (GPIOICR) register. The corresponding GPIOMIS bit reflects the masked value of the RIS bit.

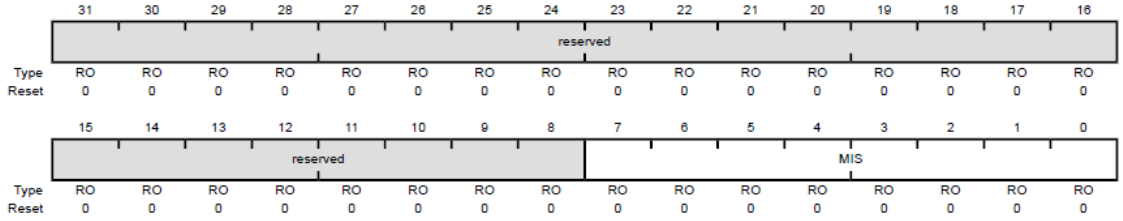


Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	RIS	RO	0x00	GPIO Interrupt Raw Status Value Description 0 An interrupt condition has not occurred on the corresponding pin. 1 An interrupt condition has occurred on the corresponding pin. For edge-detect interrupts, this bit is cleared by writing a 1 to the corresponding bit in the **GPIOICR** register. For a GPIO level-detect interrupt, the bit is cleared when the level is deasserted.

GPIO Masked Interrupt Status (GPIOMIS), offset 0x418 (read only)

If a bit is set in this register, the corresponding interrupt has triggered an interrupt to the interrupt controller. If a bit is clear, either no interrupt has been generated, or the interrupt is

masked. Note that if the Port B GPIOADCCTL register is cleared, PB4 can still be used as an external trigger for the ADC. This is a legacy mode, which allows code written for previous devices to operate on this microcontroller. GPIOMIS is the state of the interrupt after masking.

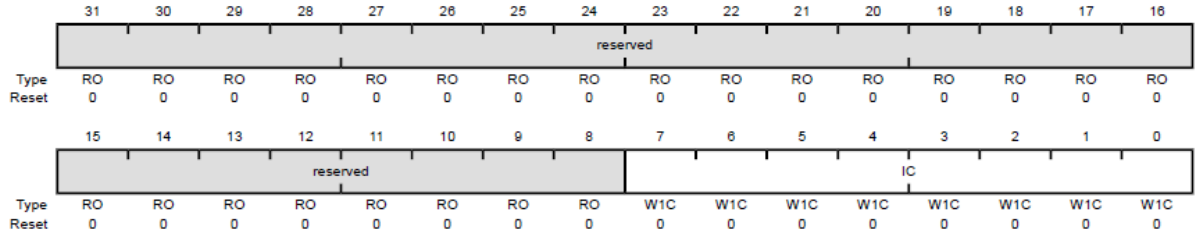


Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	MIS	RO	0x00	GPIO Masked Interrupt Status

Value	Description
0	An interrupt condition on the corresponding pin is masked or has not occurred.
1	An interrupt condition on the corresponding pin has triggered an interrupt to the interrupt controller.

GPIO Interrupt Clear (GPIOICR), offset 0x41C

For edge-detect interrupts, writing a 1 to the IC bit in the GPIOICR register clears the corresponding bit in the GPIORIS and GPIOMIS registers. If the interrupt is a level-detect, the IC bit in this register has no effect. In addition, writing a 0 to any of the bits in the GPIOICR register has no effect.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IC	W1C	0x00	GPIO Interrupt Clear

Value	Description
0	The corresponding interrupt is unaffected.
1	The corresponding interrupt is cleared.

6. GPIO Pad Control Registers:

The pad control registers allow software to configure the GPIO pads based on the application requirements. The pad control registers include the **GPIODR2R, GPIODR4R, GPIODR8R, GPIOODR, GPIOPUR, GPIOPDR, GPIOSLR**, and **GPIODEN** registers. These registers control drive strength, open-drain configuration, pull-up and pull-down resistors, slew-rate

control and digital input enable for each GPIO. If 5 V is applied to a GPIO configured as an open-drain output, the output voltage will depend on the strength of your pull-up resistor. The GPIO pad is not electrically configured to output 5 V.

GPIO 2-mA Drive Select (GPIODR2R), offset 0x500

The GPIODR2R register is the 2-mA drive control register. Each GPIO signal in the port can be individually configured without affecting the other pads. When setting the DRV2 bit for a GPIO signal, the corresponding DRV4 bit in the GPIODR4R register and DRV8 bit in the GPIODR8R register are automatically cleared by hardware. By default, all GPIO pins have 2-mA drive.

GPIO 4-mA Drive Select (GPIODR4R), offset 0x504

The GPIODR4R register is the 4-mA drive control register. Each GPIO signal in the port can be individually configured without affecting the other pads. When setting the DRV4 bit for a GPIO signal, the corresponding DRV2 bit in the GPIODR2R register and DRV8 bit in the GPIODR8R register are automatically cleared by hardware.

GPIO 8-mA Drive Select (GPIODR8R), offset 0x508

The GPIODR8R register is the 8-mA drive control register. Each GPIO signal in the port can be individually configured without affecting the other pads. When setting the DRV8 bit for a GPIO signal, the corresponding DRV2 bit in the GPIODR2R register and DRV4 bit in the GPIODR4R register are automatically cleared by hardware. The 8-mA setting is also used for high-current operation.

GPIO Open Drain Select (GPIOODR), offset 0x50C

The GPIOODR register is the open drain control register. Setting a bit in this register enables the open-drain configuration of the corresponding GPIO pad. When open-drain mode is enabled, the corresponding bit should also be set in the GPIO Digital Enable (GPIODEN) register. Corresponding bits in the drive strength and slew rate control registers (GPIODR2R, GPIODR4R, GPIODR8R, and GPIOSLR) can be set to achieve the desired fall times. The GPIO acts as an input if the corresponding bit in the GPIODIR register is cleared. If open drain is selected while the GPIO is configured as an input, the GPIO will remain an input and the open-drain selection has no effect until the GPIO is changed to an output.

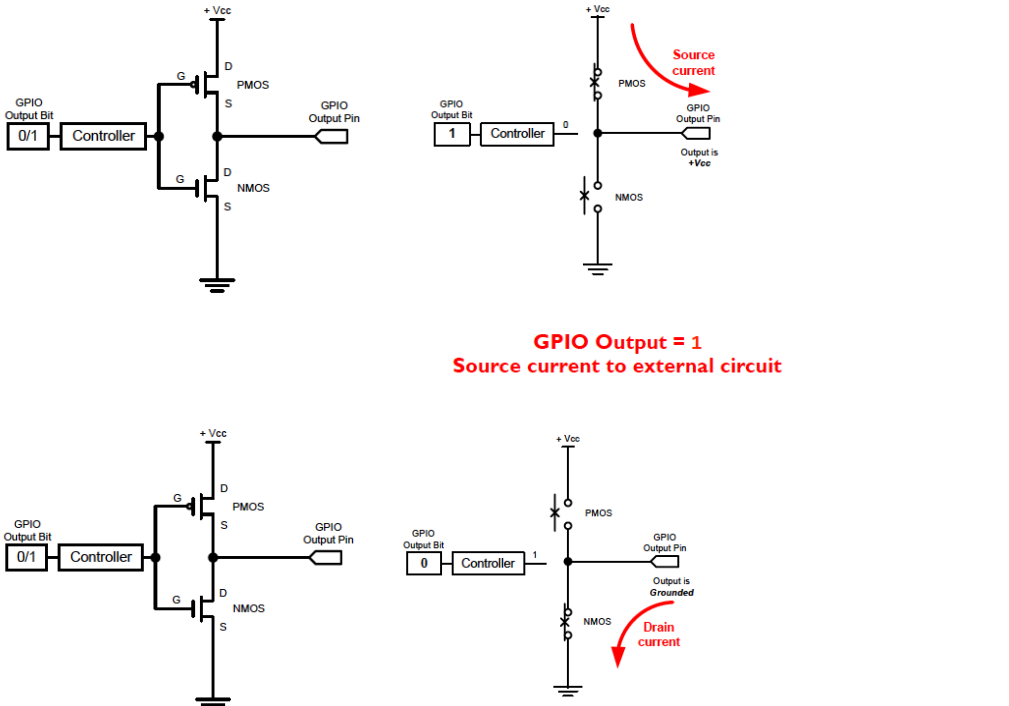


Figure 6.5: Push pull

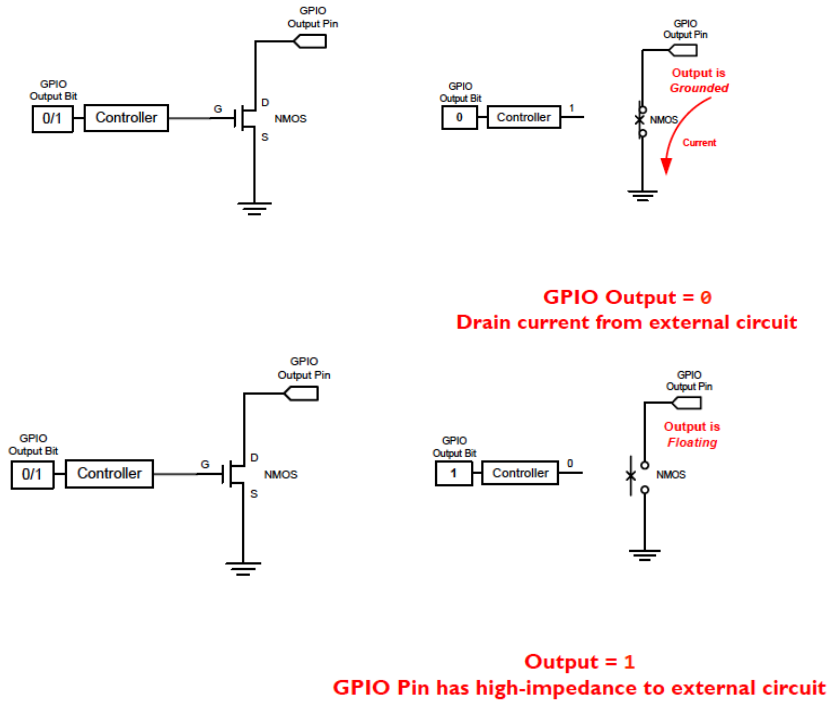


Figure 6.6: Open drain

GPIO Pull-Up Select (GPIOPUR), offset 0x510

The GPIOPUR register is the pull-up control register. When a bit is set, a weak pull-up resistor (around 20 KΩ) on the corresponding GPIO signal is enabled. Setting a bit in GPIOPUR automatically clears the corresponding bit in the GPIO Pull-Down Select (GPIOPDR) register. Write access to this register is protected with the GPIOCR register. Bits in GPIOCR that are cleared prevent writes to the equivalent bit in this register.

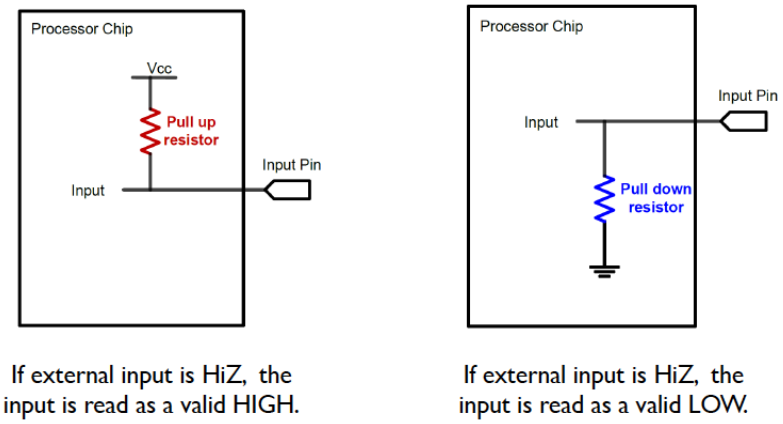
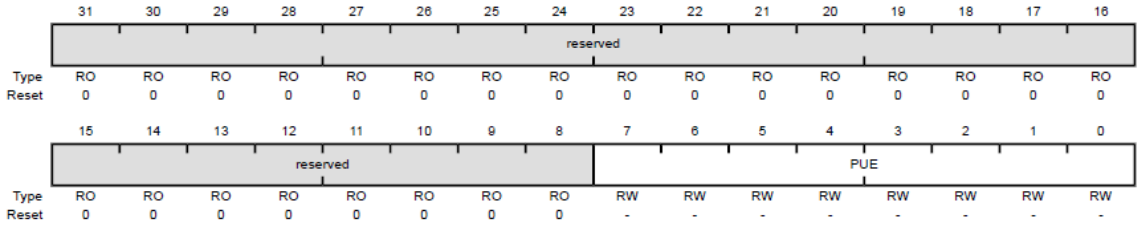


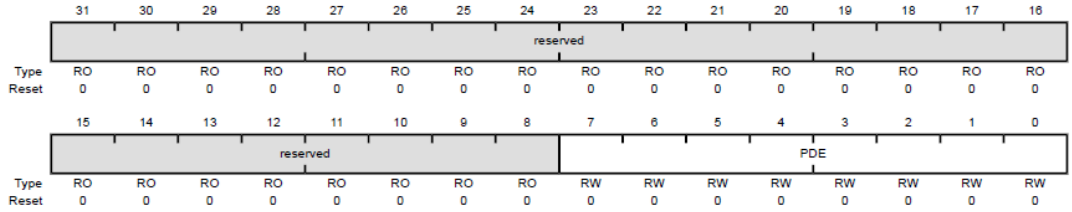
Figure 6.7: Pull-up and pull-down resistors. A digital input can have three states: High, low and high-impedance.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	PUE	RW	-	Pad Weak Pull-Up Enable Value Description 0 The corresponding pin's weak pull-up resistor is disabled. 1 The corresponding pin's weak pull-up resistor is enabled.

GPIO Pull-Down Select (GPIOPDR), offset 0x514

The GPIOPDR register is the pull-down control register. When a bit is set, a weak pull-down resistor on the corresponding GPIO signal is enabled. Setting a bit in GPIOPDR automatically clears the corresponding bit in the GPIO Pull-Up Select (GPIOPUR) register.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	PDE	RW	0x00	Pad Weak Pull-Down Enable Value Description 0 The corresponding pin's weak pull-down resistor is disabled. 1 The corresponding pin's weak pull-down resistor is enabled.

GPIO Slew Rate Control Select (GPIOSLR), offset 0x518

The GPIOSLR register is the slew rate control register. Slew rate control is only available when using the 8-mA drive strength option. The selection of drive strength is done through the GPIO 8-mA Drive Select (GPIODR8R) register.

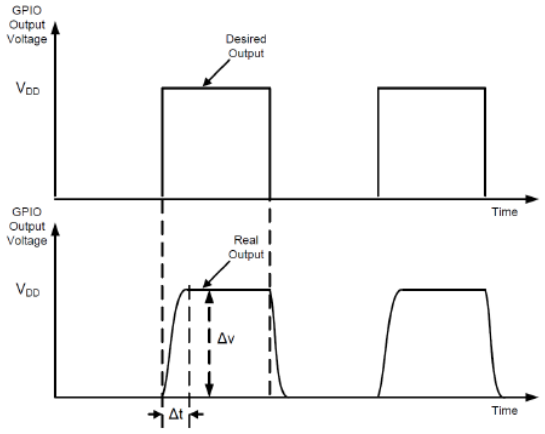
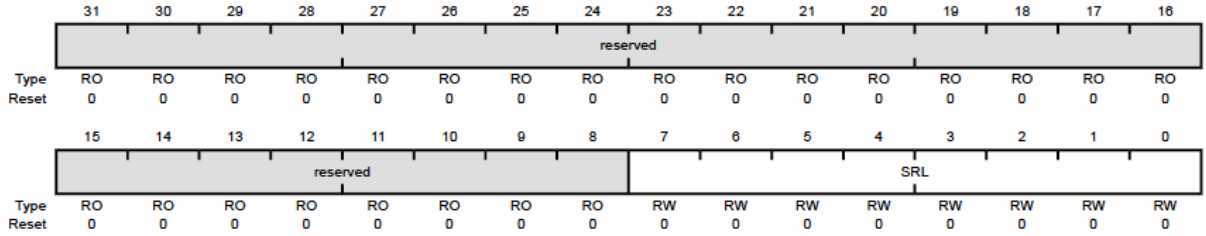


Figure 6.8: Slew rate=max(ΔV/Δt) is the maximum rate of change of the output voltage. A high slew rate allows the output to be toggled at a fast speed.



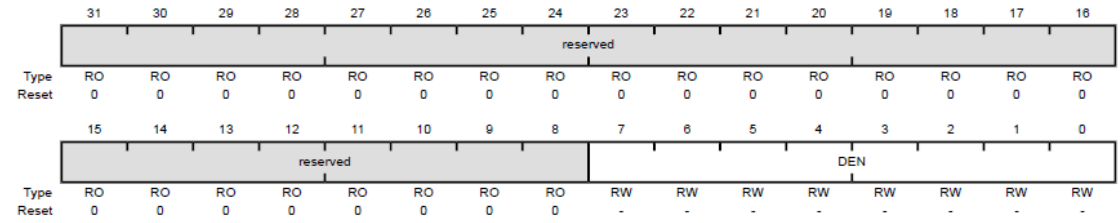
Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	SRL	RW	0x00	Slew Rate Limit Enable (8-mA drive only)

Value	Description
0	Slew rate control is disabled for the corresponding pin.
1	Slew rate control is enabled for the corresponding pin.

GPIO Digital Enable (GPIODEN), offset 0x51C

Note: Pins configured as digital inputs are Schmitt-triggered.

The GPIODEN register is the digital enable register. By default, all GPIO signals (except some) are configured out of reset to be undriven (tristate). Their digital function is disabled; they do not drive a logic value on the pin and they do not allow the pin voltage into the GPIO receiver. To use the pin as a digital input or output (either GPIO or alternate function), the corresponding GPIODEN bit must be set.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	DEN	RW	-	Digital Enable

Value	Description
0	The digital functions for the corresponding pin are disabled.
1	The digital functions for the corresponding pin are enabled.

7. GPIO Lock: GPIOLOCK, offset 0x520

TI locks some GPIO pins as shown in the table below because they're required for programming interfaces like JTAG. SW2 on the LaunchPad is connected to PF0 , which is one of these locked up pins. We can't use it as an input without unlocking it first.

GPIO Pins	Default Reset State	GPIOAFSEL	GPIODEN	GPIOPDR	GPIOPUR	GPIOPCTL	GPIOCR
PA[1:0]	UART0	0	0	0	0	0x1	1
PA[5:2]	SSI0	0	0	0	0	0x2	1
PB[3:2]	$I^{21}C0$	0	0	0	0	0x3	1
PC[3:0]	JTAG/SWD	1	1	0	1	0x1	0
PD[7]	GPIO[a]	0	0	0	0	0x0	0
PF[0]	GPIO[a]	0	0	0	0	0x0	0

In order to unlock, GPIOLOCK needs to be set to 0x4C4F.434B. This process enables write access to GPIOCR, register. Writing any other value to the GPIOLOCK register re-enables the locked state. Reading the GPIOLOCK register returns the lock status rather than the 32-bit value that was previously written. Therefore, when write accesses are disabled, or locked, reading the GPIOLOCK register returns 0x0000.0001. When write accesses are enabled, or unlocked, reading the GPIOLOCK register returns 0x0000.0000.

8. GPIO Commit: GPIOCR, offset 0x524

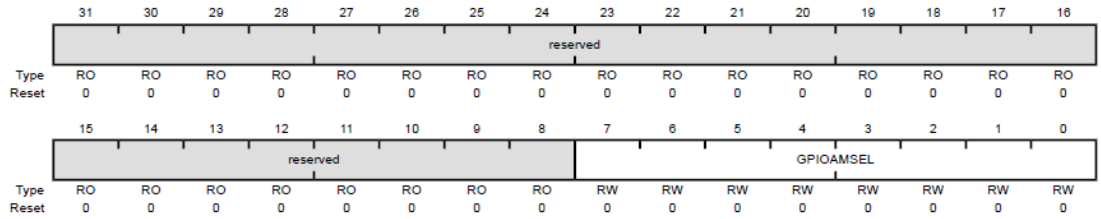
The value of the GPIOCR register determines which bits of the GPIOAFSEL, GPIOPUR, GPIOPDR, and GPIODEN registers are committed when a write to these registers is performed. If a bit in the GPIOCR register is cleared, the data being written to the corresponding bit in the GPIOAFSEL, GPIOPUR, GPIOPDR, or GPIODEN registers cannot be committed and retains its previous value. If a bit in the GPIOCR register is set, the data being written to the corresponding bit of the GPIOAFSEL, GPIOPUR, GPIOPDR, or GPIODEN registers is committed to the register and reflects the new value.

The contents of the GPIOCR register can only be modified if the status in the GPIOLOCK register is unlocked. Writes to the GPIOCR register are ignored if the status in the GPIOLOCK register is locked.

9. GPIO Analog Mode Select: GPIOAMSEL, offset 0x528

Important: This register is only valid for ports and pins that can be used as ADC AINx inputs.

If any pin is to be used as an ADC input, the appropriate bit in GPIOAMSEL must be set to disable the analog isolation circuit. The GPIOAMSEL register controls isolation circuits to the analog side of a unified I/O pad. Because the GPIOs may be driven by a 5-V source and affect analog operation, analog circuitry requires isolation from the pins when they are not used in their analog function. Each bit of this register controls the isolation circuitry for the corresponding GPIO signal.

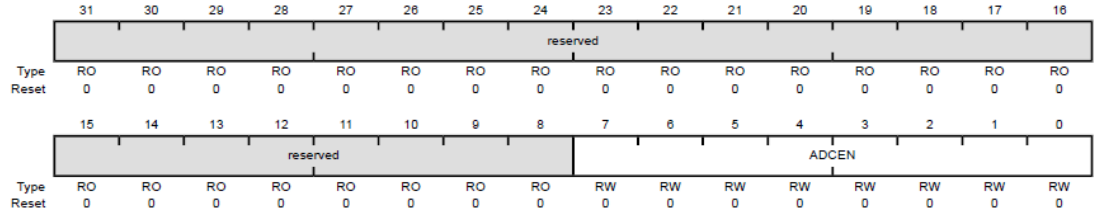


Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	GPIOAMSEL	RW	0x00	GPIO Analog Mode Select

Value	Description
0	The analog function of the pin is disabled, the isolation is enabled, and the pin is capable of digital functions as specified by the other GPIO configuration registers.
1	The analog function of the pin is enabled, the isolation is disabled, and the pin is capable of analog functions.

10. GPIO ADC Control: GPIOADCCTL, offset 0x530

This register is used to configure a GPIO pin as a source for the ADC trigger. Note that if the Port B GPIOADCCTL register is cleared, PB4 can still be used as an external trigger for the ADC. This is a legacy mode which allows code written for previous devices to operate on this microcontroller.



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	ADCEN	RW	0x00	ADC Trigger Enable

Value	Description
0	The corresponding pin is not used to trigger the ADC.
1	The corresponding pin is used to trigger the ADC.

11. GPIO DMA Control: GPIODMACTL, offset 0x534

This register is used to configure a GPIO pin as a source for the µDMA trigger.

12. GPIO Peripheral Identification Registers (read only)

The GPIOPeriphID4, GPIOPeriphID5, GPIOPeriphID6, and GPIOPeriphID7 registers can conceptually be treated as one 32-bit register; each register contains eight bits of the 32-bit register, used by software to identify the peripheral.

12. GPIO Prime Cell Identification Registers (read only)

The GPIOPCellID0, GPIOPCellID1, GPIOPCellID2, and GPIOPCellID3 registers are four 8-bit wide registers, that can conceptually be treated as one 32-bit register. The register is used as a standard cross-peripheral identification system.

In summary the GPIO registers are

Offset	Name	Type	Reset	Description	See page
0x000	GPIODATA	RW	0x0000.0000	GPIO Data	662
0x400	GPIODIR	RW	0x0000.0000	GPIO Direction	663
0x404	GPIOIS	RW	0x0000.0000	GPIO Interrupt Sense	664
0x408	GPIOIBE	RW	0x0000.0000	GPIO Interrupt Both Edges	665
0x40C	GPIOIEV	RW	0x0000.0000	GPIO Interrupt Event	666
0x410	GPIOIM	RW	0x0000.0000	GPIO Interrupt Mask	667
0x414	GPIORIS	RO	0x0000.0000	GPIO Raw Interrupt Status	668
0x418	GPIOMIS	RO	0x0000.0000	GPIO Masked Interrupt Status	669
0x41C	GPIOICR	W1C	0x0000.0000	GPIO Interrupt Clear	670
0x420	GPIOAFSEL	RW	-	GPIO Alternate Function Select	671
0x500	GPIODR2R	RW	0x0000.00FF	GPIO 2-mA Drive Select	673
0x504	GPIODR4R	RW	0x0000.0000	GPIO 4-mA Drive Select	674
0x508	GPIODR8R	RW	0x0000.0000	GPIO 8-mA Drive Select	675
0x50C	GPIOODR	RW	0x0000.0000	GPIO Open Drain Select	676
0x510	GPIOPUR	RW	-	GPIO Pull-Up Select	677
0x514	GPIOPDR	RW	0x0000.0000	GPIO Pull-Down Select	679
0x518	GPIOSLR	RW	0x0000.0000	GPIO Slew Rate Control Select	681
0x51C	GPIODEN	RW	-	GPIO Digital Enable	682
0x520	GPIOLOCK	RW	0x0000.0001	GPIO Lock	684
0x524	GPIOCR	-	-	GPIO Commit	685
0x528	GPIOAMSEL	RW	0x0000.0000	GPIO Analog Mode Select	687
0x52C	GPIOPCTL	RW	-	GPIO Port Control	688
0x530	GPIOADCCTL	RW	0x0000.0000	GPIO ADC Control	690
0x534	GPIODMACTL	RW	0x0000.0000	GPIO DMA Control	691
0xFD0	GPIOPeriphID4	RO	0x0000.0000	GPIO Peripheral Identification 4	692
0xFD4	GPIOPeriphID5	RO	0x0000.0000	GPIO Peripheral Identification 5	693
0xFD8	GPIOPeriphID6	RO	0x0000.0000	GPIO Peripheral Identification 6	694
0xFDC	GPIOPeriphID7	RO	0x0000.0000	GPIO Peripheral Identification 7	695
0xFE0	GPIOPeriphID0	RO	0x0000.0061	GPIO Peripheral Identification 0	696
0xFE4	GPIOPeriphID1	RO	0x0000.0000	GPIO Peripheral Identification 1	697
0xFE8	GPIOPeriphID2	RO	0x0000.0018	GPIO Peripheral Identification 2	698
0xFEC	GPIOPeriphID3	RO	0x0000.0001	GPIO Peripheral Identification 3	699
0xFF0	GPIOPCellID0	RO	0x0000.000D	GPIO PrimeCell Identification 0	700
0xFF4	GPIOPCellID1	RO	0x0000.00F0	GPIO PrimeCell Identification 1	701
0xFF8	GPIOPCellID2	RO	0x0000.0005	GPIO PrimeCell Identification 2	702
0xFFC	GPIOPCellID3	RO	0x0000.00B1	GPIO PrimeCell Identification 3	703

3. Detailed GPIO Initialization Sequence

1. Enable the Port's Clock in **RCGCGPIO**. If you don't enable the peripheral's clock, your application will end up in the fault handler.

```
Ex:   LDR R1,=SYSCTL_RCGCGPIO          ;Clock register
      LDR R0, [R1]
      ORR R0, R0, #0x2F                ;turn on clock for all 5 ports
      STR R0, [R1]                     ;least sign bit corresponds to PortA
```

2. The clock to a peripheral does not become active immediately. As a result you will need to poll the GPIO Peripheral Ready Register (**PRGPIO**) or just wait.

```
Ex:   NOP                              ; To wait to stabilize clocks
      NOP
      NOP
```

3. Unlock the ports.

```
Ex:   LDR R1, =PORTF_LOCK_R
      LDR R0, =0x4C4F434B
      STR R0, [R1]
      LDR R1, =PORTF_CR
      MOV R0, #0xE0              ; committed  Port F(4:7)
      STR R0, [R1]
```

4. Set the direction of the GPIO port pins by programming the GPIODIR register. A write of a 1 indicates output and a write of a 0 indicates input.

```
Ex:   LDR R1, =GPIO_PORTB_DIR      ;direction register address
      LDR R0, [R1]                 ;direction register data
      ORR R0, R0, #0x0F            ;set 3,2,1,0 of PORTB to output
      STR R0, [R1]
```

5. Clear or set the bits in the Alternate Function Select register (**AFSEL**). (0 means regular I/O port). If alternate function is selected, modify **PCTL** register, which is set to specify the alternative function for that pin (0 means regular I/O port). Each GPIO pin needs four bits in its corresponding PCTL register. Not every pin can be configured to every alternative function.

```
Ex:   LDR R1, =GPIO_PORTB_PCTL
      LDR R0,[R1]                  ;  configure the Port as GPIO
      BIC R0, #0xFF
      STR R0, [R1]
      LDR R1, =GPIO_PORTB_AFSEL
      STR R0, [R1]                 ; 0 means disable alternate function
```

6. Configure the pins as digital pins or analog pins by writing the **DEN** and **AMSEL** registers. If this pin is connected to the ADC or analog comparator, its corresponding bit in AMSEL must be set as 1. Otherwise it should be zero.

```
Ex:   LDR R1, =GPIO_PORTB_DEN
      MOV R0, #0xFF                 ; enable Port as digital port
      STR R0, [R1]
      LDR R1, =GPIO_PORTB_AMSL
      MOV R0, #0                    ; disable analog functionality
      STR R0, [R1]
```

7. Set the drive strength for each of the pins through the GPIODR2R, GPIODR4R, and GPIODR8R registers. Program each pad in the port to have either pull-up, pull-down, or open drain functionality through the GPIOPUR, GPIOPDR, GPIOODR register. Slew rate may also be programmed, if needed, through the GPIOSLR register.

```
Ex:   LDR R1, =GPIO_PORTB_PUR_R
      MOV R0, #0x11                 ;pull ups on pins 0 and 4 of PORT B
      STR R0, [R1]
```

8. Program the GPIOIS, GPIOIBE, GPIOEV, and GPIOIM registers to configure the type, event, and mask of the interrupts for each port.

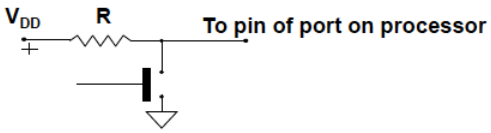
```
Ex:   LDR R1, =GPIO_PORTB_IS
      LDR R2, =GPIO_PORTB_IBE
      LDR R3, =GPIO_PORTB_IEV
      LDR R4, =GPIO_PORTB_IM
      LDR R5, =GPIO_PORTB_ICR

      MOV R0, #0x00
      STR R0, [R1]                  ;  PB is edge-sensitive
      STR R0, [R2]                  ;  PB is not both edges
      MOV R0, #0x10                 ;  PB4 is rising edge
      STR R0, [R3]
      STR R0, [R4]                  ;  enable interrupt for PB4
      STR R0, [R5]                  ; clear interrupt flag for PB4
```

9. Read/Write data from the Data register (**DATA**).

```
Ex:   PBdata  EQU  0x400053FC       ; port B data reg (all 8 bit)

      LDR R1, =PBdata
      MOV R0, #0x06                 ;assume all bits are defined as output
      STR R0, [R1]
```

4. Input and Output Devices

4.1. Switch



When switch is in the open/close position, a logic 1/0 is provided to the port pin it is connected to. The pull-up resistor defines pin voltage when switch is open and limits the current flow when switch is pushed.

Real switches typically make/break contact several times when depressed and released and this usually takes 20-100 ms. This is called as debouncing and usually handled by reading the actual switch position with some delay.

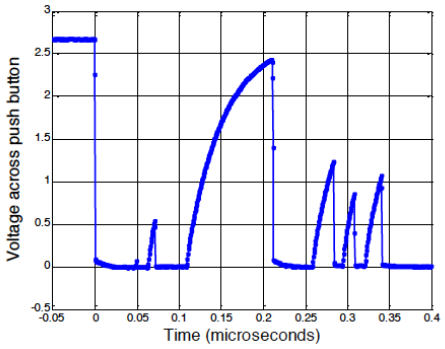
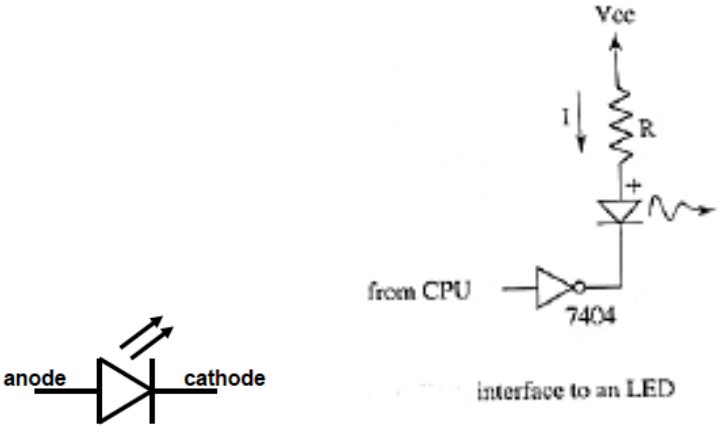


Figure 6.9: Switch bouncing.

4.2. LED



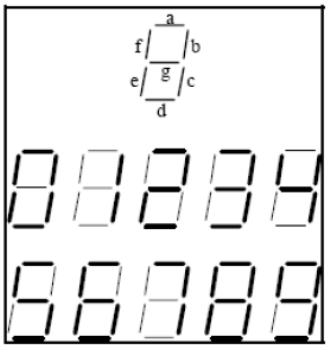
Light emitting diodes (LED) emit light in forward bias mode, i.e., when anode is at higher voltage than cathode. There are different color LEDs. You must choose the right voltage and

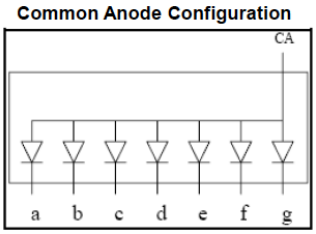
current level for the brightness you desire. Also a buffer might be needed between the port pin and the LED.

Example: Blind data transfer: Write a function that initializes GPIO Port C for PC4 button input and PC5 LED output.

```
PC4                     EQU 0x40006040
PC5                     EQU 0x40006080
GPIO_PORTC_DIR_R        EQU 0x40006400
GPIO_PORTC_AFSEL_R      EQU 0x40006420
GPIO_PORTC_DEN_R        EQU 0x4000651C
GPIO_PORTC_PUR_R        EQU 0x40006510
SYSCTL_RCGC2_R          EQU 0x400FE108

; PortC_Init
; Activate Port C and initialize it for built-in buttons and LED.
; Modifies: R0, R1

PortC_Init
   ; activate clock
        LDR R1, =SYSCTL_RCGC2_R         ; R1 = &SYSCTL_RCGC2_R
        LDR R0, [R1]
        ORR R0, R0, #0x2F               ; turn on clock for all 5 ports
        STR R0, [R1]                    ; least sign bit corresponds to PortA
        NOP
        NOP                              ; allow time to finish activating
   ; set direction register
        LDR R1, =GPIO_PORTC_DIR_R       ; R1 = &GPIO_PORTC_DIR_R
        LDR R0, [R1]
        ORR R0, R0, #0x20               ; make PC5 output; PC5 LED
        BIC R0, R0, #0x10               ; make PC4 input; PC4 switch
        STR R0, [R1]
   ; regular port function
        LDR R1, =GPIO_PORTC_AFSEL_R     ; R1 = &GPIO_PORTC_AFSEL_R
        LDR R0, [R1]
        BIC R0, R0, #0x30               ; disable alt funct, pins 4,5
        STR R0, [R1]
   ; pull-up resistor on switch pin
        LDR R1, =GPIO_PORTC_PUR_R
        MOV R0 #0x10                    ;pull up on pins 4 of PORT C
        STR R0, [R1]
   ; enable digital port
        LDR R1, =GPIO_PORTC_DEN_R       ; R1 = &GPIO_PORTC_DEN_R
        LDR R0, [R1]
        ORR R0, R0, #0x30               ; enable digital I/O on PC5-4
        STR R0, [R1]
        BX  LR                           ; return
```

4.3. Seven segment displays

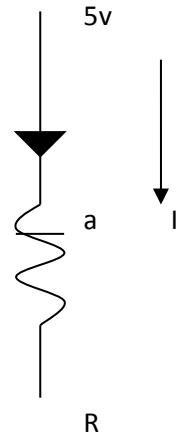




7 individual LEDs can display all ten digits and hexadecimal letters and several other symbols. A simple software routine can convert a hex digit to a display code.

1. Common anode connection

(current is sinking when bit is low)

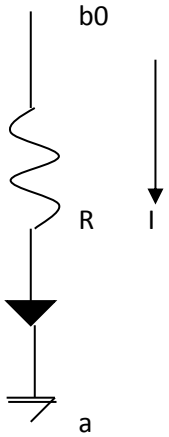


x	0	0	1	1	0	0	1
g	f	e	d	c	b	a	

0: segment illuminated

2. Common cathode connection

(current is sinking when bit is high)



x	1	0	0	1	1	1	1
g	f	e	d	c	b	a	

1: segment illuminated

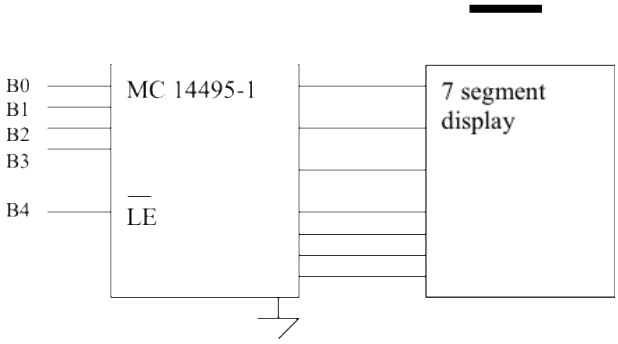


Figure 6.10: hexadecimal to 7 segment display code conversion.

Example: Data transfer by polling: Write a function that initializes GPIO Port C for PC0-3 as input switches and PC4-7 connected to a seven segment display. When a switch is pressed its number will be displayed on seven segment display. Assume hex-to-seven segment decoder is available.

```
PC_INP                  EQU 0x4000603C
PC_OUT                  EQU 0x400063C0
GPIO_PORTC_DIR_R        EQU 0x40006400
GPIO_PORTC_AFSEL_R      EQU 0x40006420
GPIO_PORTC_DEN_R        EQU 0x4000651C
GPIO_PORTC_PDR          EQU 0x40006514
SYSCTL_RCGC2_R          EQU 0x400FE108

Start
        BL  PortC_Init              ; initialize Port C
loop
        LDR R1, =PC_INP             ; R1 = 0x4000603C
loop_c LDR R0, [R1]                 ; R0 = [R1] (read pc0-pc3)
        AND R2,R0,#0x0F             ; check if any switch is pressed
        CBZ R2,loop_c
        MOV   R3,#0x04              ; counter for shift amount
        MOV   R2,#0x00              ;index of switch
Lshift   LSRS R0,R0,#1              ;loop until the pressed sw is found
        ADDCC R2,R2,#1
        BCS    display
        SUBS  R3,R3,#1
        BNE    Lshift
display LDR R1, =PCOUT              ; R1 = 0x400063C0
        STR R2, [R1]                ; display number of switch
        B   loop

; PortC_Init
; Activate Port C and initialize it for built-in buttons and LED.
```

```
; Modifies: R0, R1

PortC_Init
   ; activate clock
        LDR R1, =SYSCTL_RCGC2_R          ; R1 = &SYSCTL_RCGC2_R
        LDR R0, [R1]
        ORR R0, R0, #0x2F                ;turn on clock for all 5 ports
        STR R0, [R1]                     ; least sign bit corresponds to PortA
        NOP
        NOP                               ; allow time to finish activating
   ; set direction register
        LDR R1, =GPIO_PORTC_DIR_R        ; R1 = &GPIO_PORTC_DIR_R
         LDR R0, [R1]
        ORR R0, R0, #0xF0                ; make PC7-4 output
         BIC R0, R0, #0x0F               ; make PC3-0 input
        STR R0, [R1]
   ; regular port function
        LDR R1, =GPIO_PORTC_AFSEL_R      ; R1 = &GPIO_PORTC_AFSEL_R
        LDR R0, [R1]
        BIC R0, R0, #0xFF                ; disable alt funct,
        STR R0, [R1]
   ; pull-up resistors on switch pins
        LDR R1, =GPIO_PORTC_DR
        MOV R0 #0x0F                     ;pull down on pins 0-3 of PORT C
        STR R0, [R1]
   ; enable digital port
        LDR R1, =GPIO_PORTC_DEN_R         ; R1 = &GPIO_PORTC_DEN_R
        LDR R0, [R1]
        ORR R0, R0, #0xFF                 ; enable digital I/O on PC
        STR R0, [R1]
        BX  LR                            ; return
```

Ex: Same example with interrupt driven I/0.

Important comments

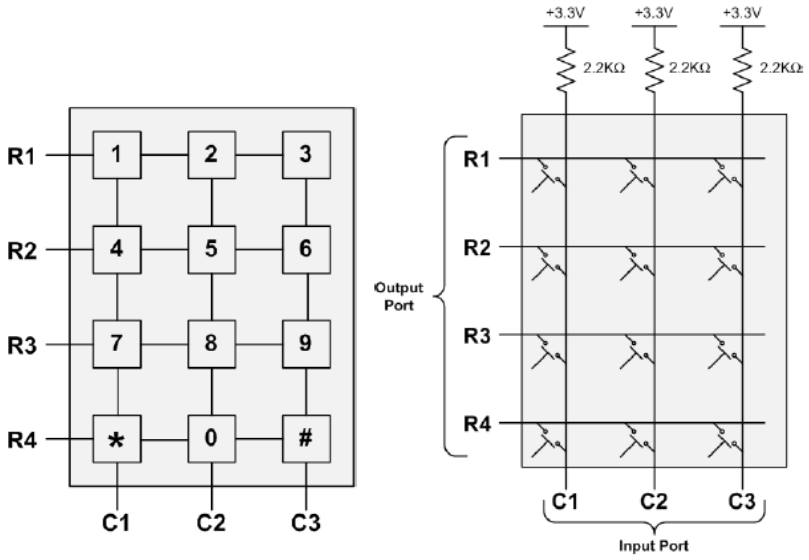
In interrupt vector table, there is only one address for GPIO PortC. So you have to check which pin is the interrupt source within the ISR and also clear the related flag. If you don't clear it, it will create an infinite loop of always getting right back into the handler after leaving it.

```
PC_INP                  EQU 0x4000603C
PC_OUT                  EQU 0x400063C0
GPIO_PORTC_DIR_R        EQU 0x40006400
GPIO_PORTC_AFSEL_R      EQU 0x40006420
GPIO_PORTC_DEN_R        EQU 0x4000651C
GPIO_PORTC_PDR          EQU 0x40006514
SYSCTL_RCGC2_R          EQU 0x400FE108
```

```
GPIO_PORTC_IS           EQU 0x40006404
GPIO_PORTC_IBE          EQU 0x40006408
GPIO_PORTC_IEV          EQU 0x4000640C
GPIO_PORTC_IM           EQU 0x40006410
GPIO_PORTC_ICR          EQU 0x4000641C
GPIO_PORTC_RIS          EQU 0x40006414

Start
        BL  PortC_Init              ; initialize Port G
Loop    WAI
        B Loop

GPIOC_Handler
        LDR R1, = GPIO_PORTC_RIS
        LDR R0,[R1]
        MOV R3,#1
        ANDS R2,R0,R3
        BNE DISP1
        ANDS R2,R0,R3, LSL #1
        BNE DISP2
loop_c ANDS R2,R0,R3, LSL #2
        BNE DISP3
        BEQ DISP4
DISP1 MOV R2,#1
        B display
DISP2  MOV R2,#2
        B display
DISP3  MOV R2,#3
        B display
DISP4  MOV R2,#4
        B display
display LDR R1, =PC5                ; R1 = 0x400063C0
        STR R2, [R1]                ; display number of switch
         LDR R1, =GPIO_PORTC_ICR
        MOV R0 #0xFF
        STR R0, [R1]
        BX      LR

; PortC_Init
; Activate Port C and initialize it for built-in buttons and LED.
; Modifies: R0, R1

PortC_Init
   ; activate clock
        LDR R1, =SYSCTL_RCGC2_R             ; R1 = &SYSCTL_RCGC2_R
        LDR R0, [R1]
        ORR R0, R0, #0x2F                   ;turn on clock for all 5 ports
        STR R0, [R1]                        ; least sign bit corresponds to PortA
```

```
        NOP
        NOP                                 ; allow time to finish activating
    ; set direction register
        LDR R1, =GPIO_PORTC_DIR_R           ; R1 = &GPIO_PORTC_DIR_R
         LDR R0, [R1]
        ORR R0, R0, #0xF0                   ; make PC7-4 output
        BIC R0, R0, #0x0F                   ; make PC3-0 input
        STR R0, [R1]
    ; regular port function
        LDR R1, =GPIO_PORTC_AFSEL_R         ; R1 = &GPIO_PORTC_AFSEL_R
        LDR R0, [R1]
        BIC R0, R0, #0xFF                   ; disable alt funct,
        STR R0, [R1]
    ; pull-up resistors on switch pins
        LDR R1, =GPIO_PORTC_DR
        MOV R0 #0x0F                        ;pull down on pins 0-3 of PORT C
        STR R0, [R1]
    ; enable digital port
        LDR R1, =GPIO_PORTC_DEN_R           ; R1 = &GPIO_PORTC_DEN_R
        LDR R0, [R1]
        ORR R0, R0, #0xFF                   ; enable digital I/O on PC
        STR R0, [R1]
;configure interrupt for Portc pins 0-3
        LDR R1, =GPIO_PORTC_IS
        LDR R2, =GPIO_PORTC_IBE
        LDR R3, =GPIO_PORTC_IEV
        LDR R4, =GPIO_PORTC_IM
        LDR R5, =GPIO_PORTC_ICR

        MOV R0 #0x00                        ;  PC is edge-sensitive
        STR R0, [R1]
        STR R0, [R2]                        ;  PC is not both edge
        MOV R0 #0x0F                        ;  PC0-3  rising edge
        STR R0, [R3]
        STR R0, [R4]                        ;  enable interrupt for PC0-3
        STR R0, [R5]                        ; clear interrupt flag for PC0-3

        BX  LR                               ; return
```

4.4. Keypad



Each key has a momentary contact switch that is connected to an intersection of row and column wires. When a key is released, an open circuit exists between all wires/terminals. When a key is pressed the contact closure connects the row and column wire. Thus, a short circuit exists between row and column wires when a key is pressed. To determine which key is pressed, a microcontroller must scan the rows and columns to identify the row and column intersection of the short circuit. A keyboard decoder can drive a signal to a column and sense the row lines to determine which key is pressed, if any.

For example, for a 4 row by 3 column key pad (12 button keypad) if rows are connected to output pins of the microprocessor and columns are connected to the input pins of the processor and if outputs corresponding to rows R1,R2,R3,R4 are set to 0000 and if inputs corresponding to columns C1,C2,C3 are 111, this means that no key is pressed. If 101 is read than some key in C2 is pressed down (Figure 6.11). To understand which key is pressed, we need to can the rows. To scan R1, output should be 0111. If the input is 111, this means that no key in R1 is pressed. To scan R2, output should be 1011. If the input is 111, this means that no key in R2 is pressed. To scan R3, output should be 1101. If the input is 111, this means that no key in R3 is pressed. To scan R4, output should be 1110. If the input is 101, this means that key at C2 is pressed. As a result, key “0” is pressed which belongs to R4 and C2. The algorithm fort his scanning process is given in Figure 6.12.

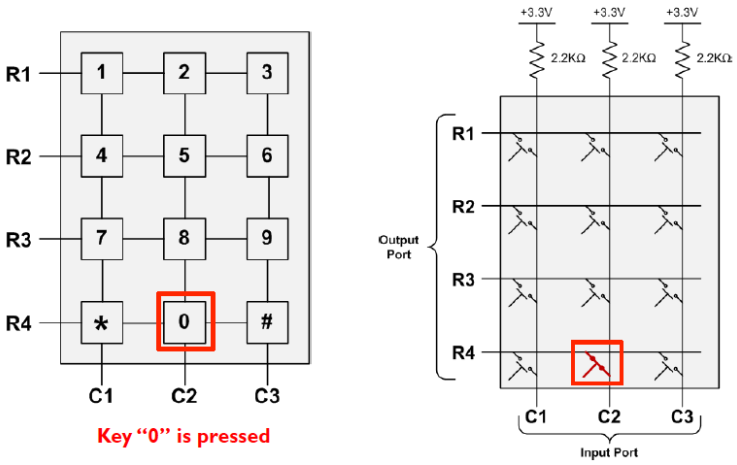


Figure 6.11: 4 by 3 keypad where key "0" is pressed.

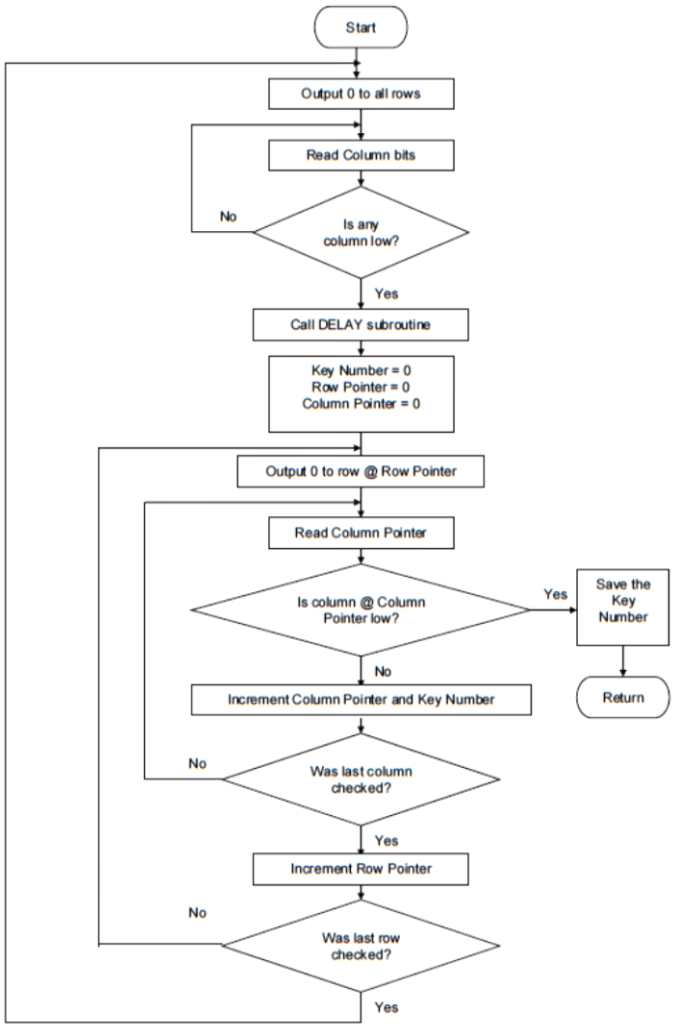


Figure 6.12: Algorithm to detect the pressed key.

Chapter 7

TIMER

Timers and counters are very useful parts of the microcontrollers providing capability to generate periodic events, measure the duration of external signals, etc.The TM4C123 has a total of 12 general purpose timers and a system timer (SysTick) which is supported by all ARM Cortex-M architectures and PWM timer.

1. The SysTick timer

It is a small timer that is integrated as part of the NVIC of the Cortex-M4 processor.This timer is generally used to support an operating system or other system management software. The reason for having the timer inside the processor is to help software portability. Since all the Cortex-M processors have the same SysTick timer, an OS written for one Cortex-M3/M4 microcontroller can be reused on other Cortex-M3/M4 microcontrollers.

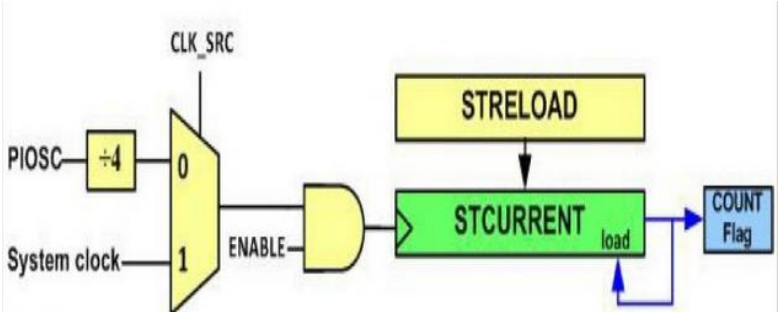
This timer can also be used to create time delays and generate periodic interrupts. Since our microcontroller does not have an operating system, we will mainly use this timer for delays and periodic interrupts. Using a timer for delays (or periodic polling an event) is preferable to using a loop because the processor is free to process other tasks while the timer works in the background.

The SysTick timer contains four registers, as shown below. The counter inside the SysTick is a 24-bit decrement counter meaning that it can count 16.777.215 cycle. It can decrement using different clock sources which can be selected by software. However, for now we will simply use the Precision Internal OSCillator (PIOSC) divided by 4. The PIOSC runs at 16 MHz on the TM4C123GH6PM, so PIOSC/4 is 4 MHz. Hence the clock period is 250 ns.

Address	CMSIS-Core Symbol	Register
0xE000E010	SysTick->CTRL	SysTick Control and Status Register
0xE000E014	SysTick->LOAD	SysTick Reload Value Register
0xE000E018	SysTick->VAL	SysTick Current Value Register
0xE000E01C	SysTick->CALIB	SysTick Calibration Register

When the counter is enabled by setting bit 0 of the Control and Status register, the current value register decrements at every processor clock cycle or every rising edge of the reference clock. If it reaches zero, it will then load the value from the reload value register and continue.

An additional register called SysTick Calibration Register is available to allow the on-chip hardware to provide calibration information for the software.



Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	STCTRL
$E000E014	0	24-bit RELOAD value						STRELOAD
$E000E018	0	24-bit CURRENT value of SysTick counter						STCURRENT

SysTick Control and Status Register(STCTRL)

Enable: 0 timer disabled, 1 timer enabled

INTEN: 0 interrupt disabled, 1 interrupt is generated to the NVIC when SysTick counts to 0.

CLK_SRC: 0 PIOSC divided by 4, 1 system clock

COUNT: 0 SysTick timer has not counted to 0 yet, 1 SysTick timer has counted to 0. This bit is cleared by a read of the register or if the STCURRENT register is written with any value.

SysTick Reload Value Register (STRELOAD)

This register specifies the start value to load into the SysTick Current Value (STCURRENT) register when the counter reaches 0.

Start value can be between 0x1 and 0x00FFFFFF (only 24 bit value can be written)

SysTick Current Value Register (STCURRENT)

This register is write-clear. Writing to it with any value clears the register. Clearing this register also clears the COUNT bit of the STCTRL register.

1.1. Steps to use SysTick

Polling

- Disable SysTick during setup, STCTRL=0.
- Write the initial counter value to STRELOAD.
- Enable SysTick and choose configurations.
    - STCTRL=0x01 (enable, no interrupt, use PIOSC as clock)
- Wait flag (Check COUNT flag)

Interrupt

- Disable SysTick during setup, STCTRL=0.
- Write the initial counter value to STRELOAD.
- Enable SysTick and choose configurations.
    - STCTRL=0x03 (enable, interrupt, use PIOSC as clock)
    - Set interrupt priority level (lower number has higher priority)
- Serve interrupt

EX: Write a program to turn on/off a LED at every 3 sec.

```
PC5                     EQU 0x40006080
GPIO_PORTC_DIR_R        EQU 0x40006400
GPIO_PORTC_AFSEL_R      EQU 0x40006420
GPIO_PORTC_DEN_R        EQU 0x4000651C
GPIO_PORTC_PUR_R        EQU 0x40006510
SYSCTL_RCGC2_R          EQU 0x400FE108
SYSCTRL                 EQU 0xE000E010


MAIN

        BL INIT_TIMER
        BL INIT_GPIO
LOOP  B LOOP


INIT_TIMER

        LDR     R0, =SYSCTRL        ; SYSTICK control and status register
        MOV     R1, #0
        STR     R1, [R0]            ; stop counter to prevent interrupt triggered accidentally
        LDR     R1, =11999999       ; trigger every 12000000 cycles
        STR     R1, [R0,#4]         ; write reload value to reload value register
        STR     R1, [R0,#8]         ; write any value to current value
        MOV     R1, #0x3            ; enable interrupt, enable SYSTICK counter
        STR     R1, [R0]            ; start counter
        BX LR


INIT_GPIO

   ; activate clock
        LDR R1, =SYSCTL_RCGC2_R             ; R1 = &SYSCTL_RCGC2_R
        LDR R0, [R1]
        ORR R0, R0, #0x2F                   ; turn on clock for all 5 ports
        STR R0, [R1]                        ; least sign bit corresponds to PortA
        NOP
        NOP                                  ; allow time to finish activating
   ; set direction register
        LDR R1, =GPIO_PORTC_DIR_R           ; R1 = &GPIO_PORTC_DIR_R
        LDR R0, [R1]
```

```
        ORR R0, R0, #0x20                   ; make PC5 output;
        STR R0, [R1]
    ; regular port function
        LDR R1, =GPIO_PORTC_AFSEL_R         ; R1 = &GPIO_PORTC_AFSEL_R
        LDR R0, [R1]
        BIC R0, R0, #0x20                    ; disable alt funct, 5
        STR R0, [R1]
    ; enable digital port
        LDR R1, =GPIO_PORTC_DEN_R           ; R1 = &GPIO_PORTC_DEN_R
        LDR R0, [R1]
        ORR R0, R0, #0x20                   ; enable digital I/O on PC5
        STR R0, [R1]
        BX  LR                              ; return


SysTick_Handler PROC
    EXPORT SysTick_Handler
        LDR R0, =PC5                ; GPIOC
        MOV R1, =0x20
        LDR R2, [R0]
        EOR R2, R2, R1
        STR R2, [R0]
        BX  LR
    ENDP
```

2. General Purpose Timers

As stated before, a timer is essentially a state machine that increments or decrements a register once per clock cycle. All timers have a CURRENT register that contains the current value of the time. Timers also have a RELOAD register. The RELOAD register is used to set the period it takes for the timer expire.

In addition to the CURRENT and RELOAD register, timers have several other characteristics to take into account. Below is a short list of characteristics that a timer might have.

Direction (Up/Down)

A timer can be a count up or count down timer. A count down timer counts down from the value in the RELOAD register until it reaches 0. A count up timer will initialize CURRENT to zero and count up until the CURRENT register reaches the value in the RELOAD register.

Periodic/One-Shot

A Timer that is free running, or periodic, will begin the count sequence over again once the programmed period of time expires. In the case of a count down time, once the CURRENT register equals zero, it will reload the CURRENT register with the value contained in the RELOAD register. This process will repeat until the timer is explicitly turned off by software.

A one-shot timer will count up/down for a single period of time. Once the timer expires, the timer will stop counting and wait for software to re-arm it.

As an example, lets say that the clock peripheral was running at 50 MHz. This means that the clock decrements once every 20 nS. If the RELOAD register was set to a value of 100.000, we can determine the period of the timer to be 20nS • 100,000 = 0.002S or two milliseconds.

Prescaler

Prescalars act as a method to slow down how quickly a clock will increment/decrement the CURRENT register. Instead of modifying the CURRENT register every clock cycle, a prescalar allows you to set the number of clock cycles the timer waits before modifying the count register.

If we assume of 50 MHz clock, a load value of 100.000 and add a prescalar of 4, the total amount of time it will take for one period is (20nS • 100.000 • 4) = 8mS.

2.1. The General-Purpose Timer Module (GPTM) of TM4C123

It contains six 16/32-bit blocks and six 32/64-bit Wide blocks with the following functional options. These are free-running counters (independent of processor).

- 16/32-bit operating modes:
  - 16- or 32-bit programmable one-shot timer
  - 16- or 32-bit programmable periodic timer
  - 16-bit general-purpose timer with an 8-bit prescaler
  - 32-bit Real-Time Clock (RTC) when using an external 32.768-KHz clock as the input
  - 16-bit input-edge count- or time-capture modes with an 8-bit prescaler
  - 16-bit PWM mode with an 8-bit prescaler and software-programmable output inversion of the PWM signal
- 32/64-bit operating modes:
  - 32- or 64-bit programmable one-shot timer
  - 32- or 64-bit programmable periodic timer
  - 32-bit general-purpose timer with a 16-bit prescaler
  - 64-bit Real-Time Clock (RTC) when using an external 32.768-KHz clock as the input
  - 32-bit input-edge count- or time-capture modes with a16-bit prescaler
  - 32-bit PWM mode with a 16-bit prescaler and software-programmable output inversion of the PWM signal
- Count up or down
- Twelve 16/32-bit Capture Compare PWM pins (CCP)
- Twelve 32/64-bit Capture Compare PWM pins (CCP)
- Daisy chaining of timer modules to allow a single timer to initiate multiple timing events
- Timer synchronization allows selected timers to start counting on the same clock cycle
- ADC event trigger
- Ability to determine the elapsed time between the assertion of the timer interrupt and entry into the interrupt service routine
- Efficient transfers using Micro Direct Memory Access Controller (µDMA)
  - Dedicated channel for each timer
  - Burst request generated on timer interrupt

The GP Timer signals are alternate functions for some GPIO signals and default to be GPIO signals at reset. Below are the names of the pins and in Table 2-1 pin numbers and

descriptions for timer 0 are given as an example. All pins and their descriptions can be found in Tiva TM4C123G data sheet.

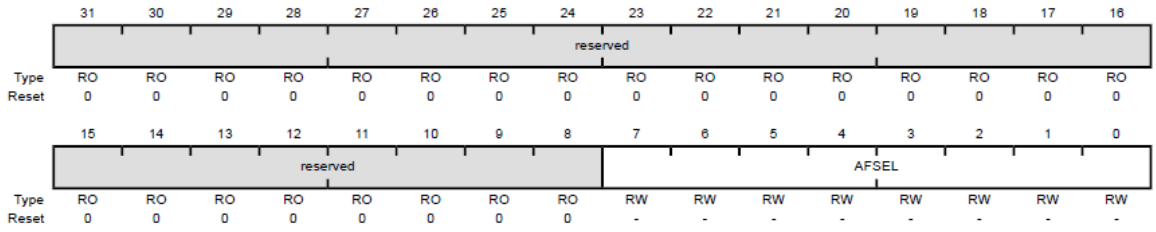
For n=0:5,		Capture Compare PWM (CCP) pin
16/32 bit Timer n	Timer A	TnCCP0
16/32 bit Timer n	Timer B	TnCCP1
32/64 bit Timer n	Timer A	WTnCCP0
32/64 bit Timer n	Timer B	WTnCCP1

Table 2-1 Pin numbers and descriptions for 16/32 bit timer 0.

Pin Name	Pin Number	Pin Mux / Pin Assignment	Pin Type	Buffer Type[a]	Description
T0CCP0	1 28	PB6 (7) PF0 (7)	I/O	TTL	16/32-Bit Timer 0 Capture/Compare/PWM 0.
T0CCP1	4 29	PB7 (7) PF1 (7)	I/O	TTL	16/32-Bit Timer 0 Capture/Compare/PWM 1.

The 2 pin assignments are like alternate pin assignment options. For example if PF0 is being used as a GPIO then you can use PB6 for T0CCP0 function and vice-versa. The AFSEL bit in the GPIO Alternate Function Select (GPIOAFSEL) register should be set to choose the GP Timer function for the specified GPIO port pin. The number in parentheses (7) is the encoding that must be programmed into the PMCn field in the GPIO Port Control (GPIOPCTL) register to assign the GP Timer signal to the specified GPIO port pin. Registers are given below as a remainder.

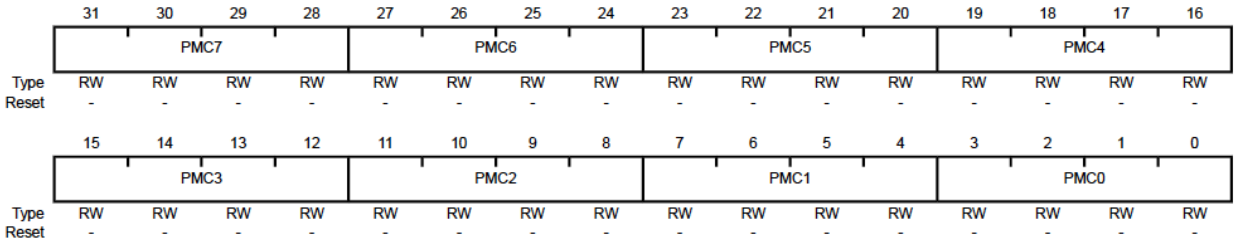
GPIO Alternate Function Select: GPIOAFSEL, offset 0x420



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	AFSEL	RW	-	GPIO Alternate Function Select

Value	Description
0	The associated pin functions as a GPIO and is controlled by the GPIO registers.
1	The associated pin functions as a peripheral signal and is controlled by the alternate hardware function.

GPIO Port Control: GPIOPCTL, offset 0x52C



Bit/Field	Name	Type	Reset	Description
31:28	PMC7	RW	-	Port Mux Control 7 This field controls the configuration for GPIO pin 7.
27:24	PMC6	RW	-	Port Mux Control 6 This field controls the configuration for GPIO pin 6.
23:20	PMC5	RW	-	Port Mux Control 5 This field controls the configuration for GPIO pin 5.
19:16	PMC4	RW	-	Port Mux Control 4 This field controls the configuration for GPIO pin 4.
15:12	PMC3	RW	-	Port Mux Control 3 This field controls the configuration for GPIO pin 3.
11:8	PMC2	RW	-	Port Mux Control 2 This field controls the configuration for GPIO pin 2.
7:4	PMC1	RW	-	Port Mux Control 1 This field controls the configuration for GPIO pin 1.
3:0	PMC0	RW	-	Port Mux Control 0 This field controls the configuration for GPIO pin 0.

Each block has two free-running up/down counters (referred to as Timer A and Timer B), two prescaler registers, two match registers, two prescaler match registers, two shadow registers, and two load/initialization registers and their associated control functions. The exact functionality of each is controlled by software and configured through the register interface. Timer A and Timer B can be used individually or Timer A and Timer B can be concatenated. When in one of the concatenated modes, Timer A and Timer B can only operate in one mode. Only the Timer A control and status bits must be used. However, when configured in an individual mode, Timer A and Timer B can be independently configured in any combination of the individual modes. Note that the prescaler can only be used when the timers are used individually.

Note that when counting down in one-shot or periodic modes, the prescaler acts as a true prescaler and contains the least-significant bits of the count. When counting up in one-shot or periodic modes, the prescaler acts as a timer extension and holds the most-significant bits of the count. In input edge count, input edge time and PWM mode, the prescaler always acts as a timer extension, regardless of the count direction.

The available modes and prescaler behavior for each block are summarized below.

Table 2-2 Timer system capabilities and prescale behaviour.

Mode	Timer Use	Count Direction	Counter Size		Prescaler Size[a]		Prescaler Behavior (Count Direction)
			16/32-bit GPTM	32/64-bit Wide GPTM	16/32-bit GPTM	32/64-bit Wide GPTM	
One-shot	Individual	Up or Down	16-bit	32-bit	8-bit	16-bit	Timer Extension (Up), Prescaler (Down)
	Concatenated	Up or Down	32-bit	64-bit	-	-	N/A
Periodic	Individual	Up or Down	16-bit	32-bit	8-bit	16-bit	Timer Extension (Up), Prescaler (Down)
	Concatenated	Up or Down	32-bit	64-bit	-	-	N/A
RTC	Concatenated	Up	32-bit	64-bit	-	-	N/A
Edge Count	Individual	Up or Down	16-bit	32-bit	8-bit	16-bit	Timer Extension (Both)
Edge Time	Individual	Up or Down	16-bit	32-bit	8-bit	16-bit	Timer Extension (Both)
PWM	Individual	Down	16-bit	32-bit	8-bit	16-bit	Timer Extension

Each timer channel can be configured in any one of the following modes:

One shot: Channel counter counts up (down) to (from) a specific value one time, giving a fixed time interval.

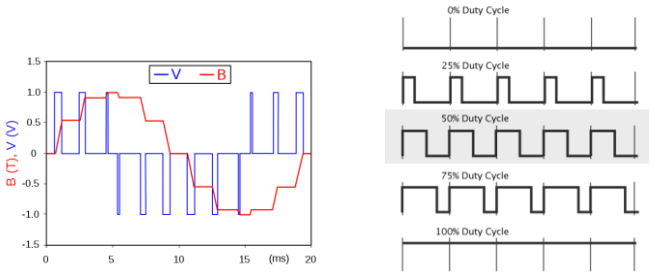
Periodic: Same as one shot, but counter counts these intervals repetitively.

Edge time: Starts counting when timer is enabled and detects an edge on a pulse signal, saving the count value at that time.

Edge count: Counts number of edges on pulse signal and saves that value.

Real Time clock (RTC): RTC is a digital clock that provides calendar time and date. Run independently from the processor core.

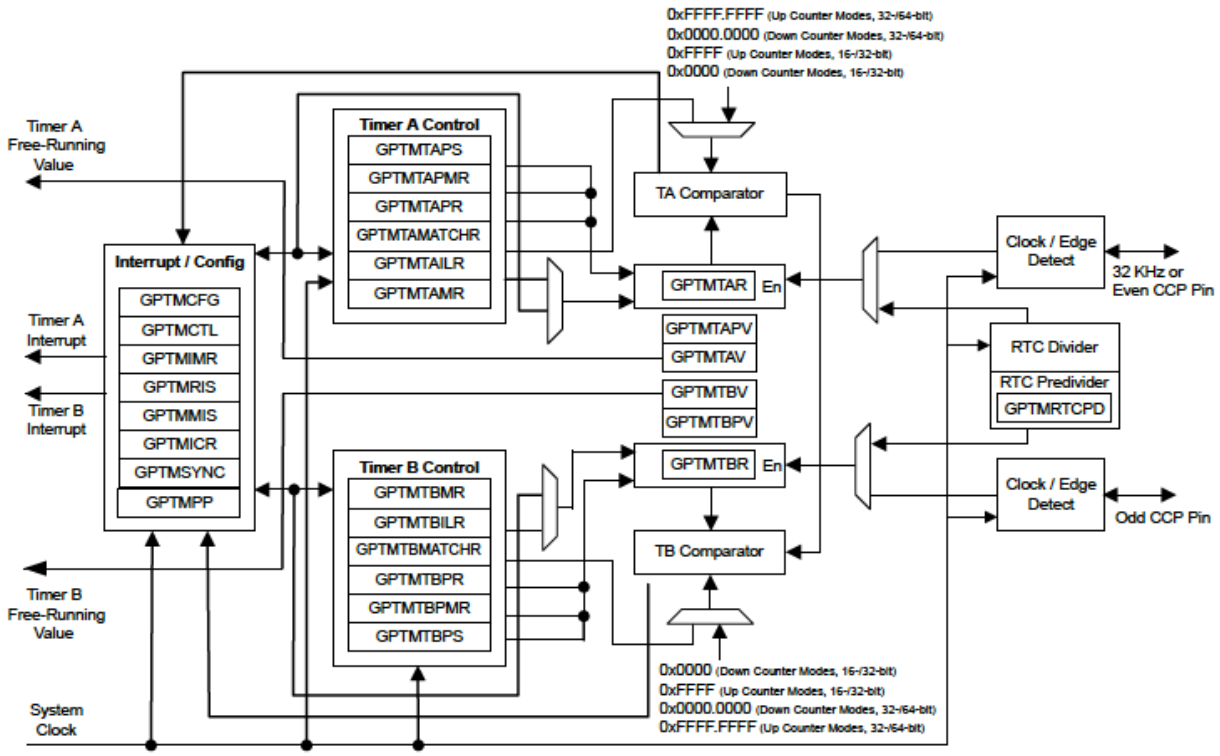
Pulse width modulation: Simple PWM mode. PWM is a method for encoding information into pulsing signal (Figure 7.1.a). Commonly used in power circuits such as motors. Varies power to a system by switching (pulsing) supply loads on and off at a fast rate. Switching must be high so that a smooth output power could be given. The higher the duty cycle (Figure .1.b) of these pulses, the higher the power supplied to the load.



a. **b.**

Figure 7.1: a. Voltage (V) applied as series of pulses modulated to a sine like wave. b. Duty cycle.

2.2. Block Diagram of a Channel



See Section 2.5 for details about registers.

Reset Condition

After reset has been applied to the module, the module is in an inactive state, all control registers are cleared to their default states. Counters Timer A and Timer B are initialized to all 1s, along with their corresponding registers:

- Load Registers:
  - Timer A Interval Load (TAILR) register
  - Timer B Interval Load (TBILR) register
- Shadow Registers:
  - Timer A Value (TAV) register
  - Timer B Value (TBV) register

The following prescale counters are initialized to all 0s:

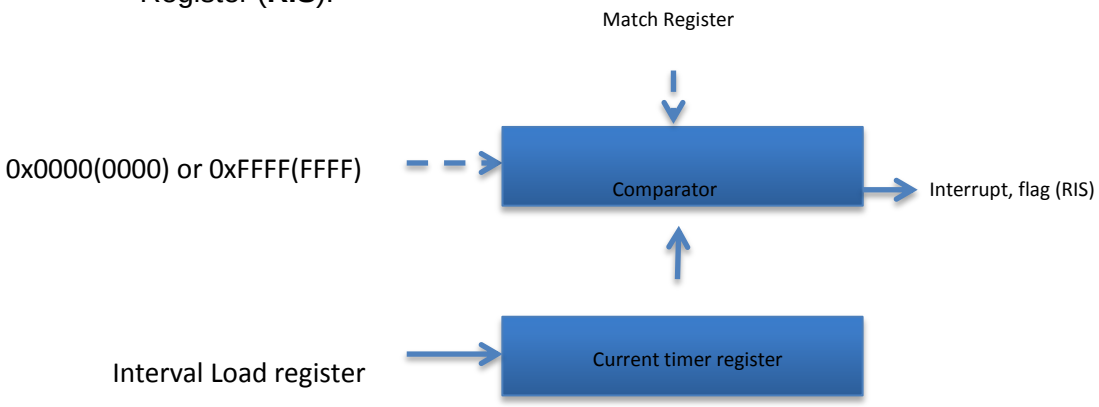
- Timer A Prescale (TAPR) register
- Timer B Prescale (TBPR) register
- Timer A Prescale Snapshot (TAPS) register
- Timer B Prescale Snapshot (TBPS) register
- Timer A Prescale Value (TAPV) register
- Timer B Prescale Value (TBPV) register

Basic Timer Initialization

(Ex: Timer0 , one-shot, count down)

1. Turn on the clock gating register for the specified timer (***RCGCTIMER***)
2. Wait for the peripheral to turn on (***PRTIMER***)
3. Disable both the A and B timers (**CTL**)
4. Set the timer to be in 32-bit mode (**CFG**)
5. Set the Timer A Mode Register to be in one-shot mode and count down (**TAMR**)
6. Set the number of clock cycles in the Timer A Interval Load Register (**TAILR**)
7. Clear the Timer Interrupt Clear Register for the specified timer (**ICR**). This will clear the status bit indicating the timer is done. Not doing can result in the timer indicating the timer has reached zero before the CURRENT register is equal to 0.
8. Enable interrupt if needed and configure **MATCH** register.
9. Enable Timer A using the Timer Control Register (**CTL**)

You can determine when the timer has reached zero by polling the Raw Interrupt Status Register (**RIS**).



2.3. Timer modes

When using Timer A and Timer B in concatenated mode, only the Timer A control and status bits must be used; there is no need to use Timer B control and status bits. The timer is placed into individual/split mode by writing a value of 0x4 to the Configuration (CFG) register.

2.3.1. One shot / Periodic Mode

Timer Time Out

To start counting

- Timer Control Register is used to start counting (by setting TnEN bits of CTL register for Timer A and B respectively).

To configure

- Timer Mode Register (TnMR) is used to configure the counting. By setting this register's bits we can configure if counting will start immediately or after a trigger, if counting will be up or down, if counting will be one shot or periodic or if a snap shot is needed.

To load count start/stop values

- The starting value of an up counter or the upper bound of a down counter can be written to the Timer Interval Load Register (TnILR) together with Timer Prescale Register (TnPR). When counting down, if time out (0x0) is reached, starting value is reloaded from these registers. When counting up, if time out is reached (upper bound), timer reloads 0x0.

Time out

- When time out occurs, if timer is configured for one shot, counting is stopped and control register is configured to stop counting. If it is configured as periodic, timer continues to count.

Counter values

- The value of free running counter is shown in Timer Value Register (TnV) and prescale in Timer Prescale Value Register (TnPV).

Interrupt and time out event

- Raw Interrupt Status Register (RIS) is set at time out event (bit TATO). Writing to Interrupt Clear Register (ICR) clears this flag. If interrupt is enabled in Interrupt Mask Register (IMR), then time out also sets Masked Interrupt Status Register (MIS).

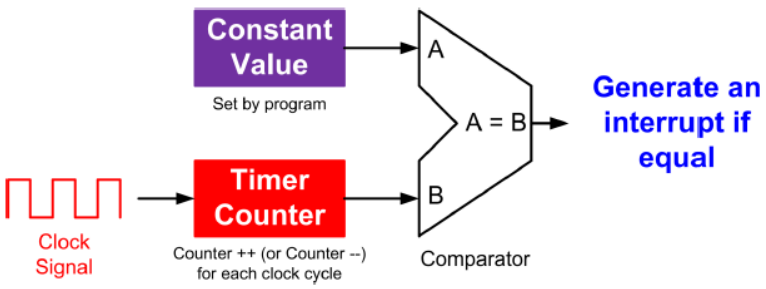
Snap shot mode

- If snap shot mode is set (in TnMR) (it cannot be set in one shot), the value of the counter at the time out is loaded to Timer Register (TnR) and prescale to Timer Prescale Snapshot Register (TnPS). The difference between these two registers (TnR, TnPS) and value registers (TnV, TnPV) can show the duration of interrupt start and handling.

If software updates the Timer n Interval Load register (TnILR) or the Prescale register (TnPR) register while the counter is counting down, the counter loads the new value on the next clock cycle and continues counting from the new value if the TnILD bit in the Timer Mode register (TnMR) is clear. If the TnILD bit is set, the counter loads the new value after the next timeout. If software updates the TnILR or the TnPR register while the counter is counting up, the timeout event is changed on the next cycle to the new value.

If software updates the Timer n Value (TnV) register while the counter is counting up or down, the counter loads the new value on the next clock cycle and continues counting from the new value. If software updates the Timer Match register (TnMATCHR) or the Timer PRescale Match register (TnPMR), the new values are reflected on the next clock cycle if the TnMRSU bit in the Timer Mode register (TnMR) is clear. If the TnMRSU bit is set, the new value will not take effect until the next timeout.

Timer Match



If TnMIE bit of Timer Mode register (TnMR) is set, an interrupt can be generated when the Timer Value and Prescale Value registers (TnV, TnPV) equal the value loaded into the Timer Match and Prescale Match registers (TnMATCHR, TnPMR). This interrupt has the same status, masking and clearing functions as the time-out interrupt and uses the related bits of the interrupt configuration and status registers.

Note that the interrupt status bits are not updated by the hardware unless the TnMIE bit in the Timer Mode register (TnMR) is set, which is different than the behavior for the time-out interrupt.

Prescaler configurations

The prescaler can only be used when a 16/32-bit timer is configured in 16-bit mode and when a 32/64-bit timer is configured in 32-bit mode.

Prescalars act as a method to slow down how quickly a clock will increment/decrement the current register (TAR/TBR). Instead of modifying the current register every clock cycle, a prescaler allows you to set the number of clock cycles the timer waits before modifying the count register.

In TM4C123, prescaling is done using an extension register to the counter. When in one-shot or periodic down count modes, this register acts as a true prescaler for the timer counter. When acting as a true prescaler, the prescaler counts down to 0 before the value in the current TAR/ TBR and value TAV/TBV registers are incremented. When counting up in one-shot or periodic modes, the prescaler acts as a timer extension and holds the most-significant bits of the count, bits 23:16 in the 16-bit modes of the 16/32-bit and bits 47:32 in the 32-bit modes of the 32/64-bit Wide.

In input edge count, input edge time and PWM mode, the prescaler always acts as a timer extension, regardless of the count direction.

The following table shows a variety of configurations for a 16-bit free-running timer while using the prescaler. All values assume an 80-MHz clock with Tc=12.5 ns (clock period).

Prescale (8-bit value)	# of Timer Clocks (Tc)[a]	Max Time	Units
00000000	1	0.8192	ms
00000001	2	1.6384	ms
00000010	3	2.4576	ms
----------	--	--	--
11111101	254	208.0768	ms
11111110	255	208.896	ms
11111111	256	209.7152	ms

a. Tc is the clock period.

Ex: Counting up

0000	0000
…	
0000	0000
0000	0001
…	
0000	0010

Prescale reg

0000	0000	0000	0000
…			
1111	1111	1111	1111
0000	0000	0000	0000
…			
1111	1111	1111	1111

Timer Reg

Ex: Counting down

1111	1111	1111	1111
1111	1111	1111	1111
1111	1111	1111	1110
…			
0000	0000	0000	0000

Timer Reg

0000	0001
0000	0000
0000	0001
…	
0000	0000

Prescale reg

2.3.2.Real Time Clock (RTC) timer Mode

The input clock on a CCP0 input is required to be 32.768 KHz in RTC mode. The clock signal is then divided down to a 1-Hz rate and is passed along to the input of the counter.

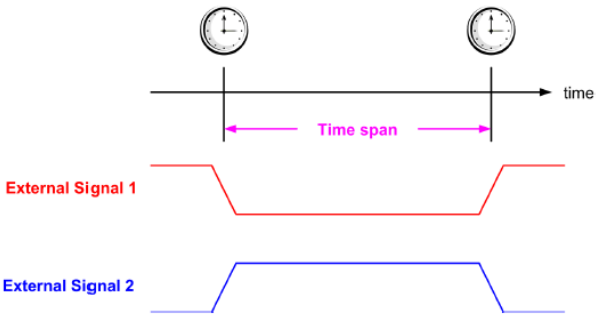
In Real-Time Clock (RTC) mode, the concatenated versions of the Timer A and Timer B registers are configured as an up-counter. When RTC mode is selected for the first time after reset, the counter is loaded with a value of 0x1. When the timer value reaches the terminal count, the timer rolls over and continues counting up from 0x0. If the Interval Load register (TnILR) is loaded with a new value, the counter begins counting at that value and rolls over at the fixed value of 0xFFFFFFFF.

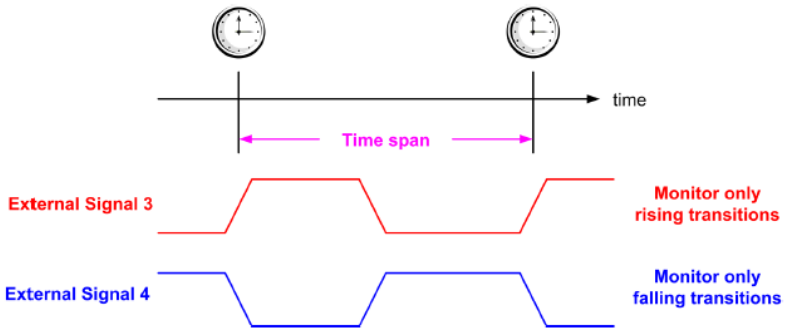
When the current count value matches the preloaded value in the Timer Match register (TnMATCHR), the timer asserts the RTCRIS bit in Raw Interrupt Status register (RIS) and continues counting until either a hardware reset, or it is disabled by software (clearing the TAEN bit in the Control register). If the RTC interrupt is enabled in the Interrupt Mask register (IMR), the timer also sets the RTCMIS bit in the Masked Interrupt Status register (MIS) and generates a controller interrupt. The status flags are cleared by writing the RTCCINT bit in the Interrupt Clear register (ICR).

In this mode, the Timer register (TnR) and Timer Value register (TnV) registers always have the same value.

The value of the RTC predivider can be read in the RTC Predivide (RTCPD) register.

2.3.3.Input Edge/Time Count Mode

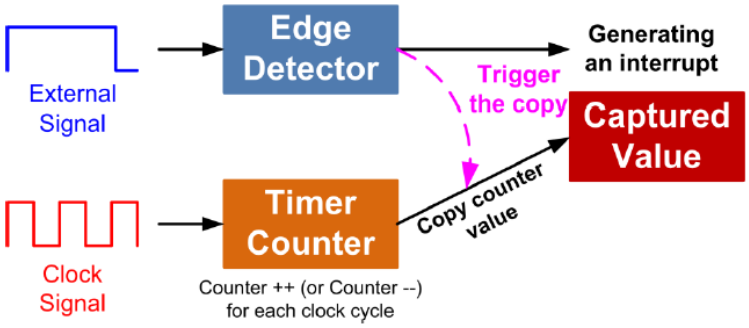




The rising and falling edges at a signal connected to a GPIO pin associated with a particular channel can be detected.

Signal's period or high/low durations can be computed at a resolution no better than channel clock period. For rising-edge detection, the input signal must be High for at least two system clock periods following the rising edge. Similarly, for falling-edge detection, the input signal must be Low for at least two system clock periods following the falling edge. Based on this criteria, the maximum input frequency for edge detection is 1/4 of the system frequency.

Edge Count Mode



When software writes the TnEN bit in the Control (CTL) register, the timer is enabled for event capture. Each input event on the CCP pin decrements or increments the counter by 1 until the event count matches Timer Match register (TnMATCHR) and Timer Prescale Match register (TnPMR).

To place the timer in Edge-Count mode, the TnCMR bit of the Mode register (TnMR) must be cleared. This bit is set for Edge-time count.

In Edge-Count mode, the timer is configured as a 24-bit or 48-bit up- or down-counter including the optional prescaler with the upper count value stored in the Timer n Prescale (TnPR) register and the lower bits in the TnR register.

In Edge-Count mode, the timer is capable of capturing three types of events: rising edge, falling edge, or both. The type of edge that the timer counts is determined by the TnEVENT fields of the control register (CTL).

During initialization in down-count mode, the match register TnMATCHR and prescale match register TnPMR are configured so that the difference between the value in the interval load register TnILR (which keeps the upper bound value) and prescale register TnPR and the match register TnMATCHR and prescale match register TnPMR equals the number of edge events that must be counted.

After the match value is reached in down-count mode, the counter is then reloaded using the value in Interval Load register (TnILR) and Prescale register (TnPR), and stopped because the timer automatically clears the TnEN bit in the CTL register. Once the event count has been reached, all further events are ignored until TnEN is re-enabled by software.

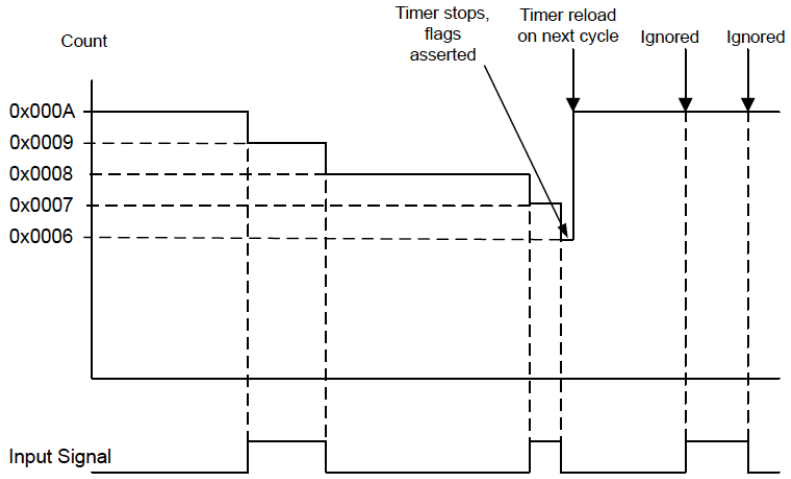


Figure 7.2: The timer (in down count mode) start value is set to TnILR = 0x000A and the match value is set to TnMATCHR = 0x0006 so that four edge events are counted. The counter is configured to detect both edges of the input signal.

In up-count mode, the timer counts from 0x0 to the value in the match register TnMATCHR and prescale match register TnPMR. After the match value is reached in up-count mode, the timer is reloaded with 0x0 and continues counting. In up count mode, the current count of input events is held in both the timer register (TnR) and timer value register (TnV).

When the counts match, the timer asserts the CnMRIS bit in the Raw Interrupt Status (RIS) register, and holds it until it is cleared by writing the Interrupt Clear (ICR) register. If the capture mode match interrupt is enabled in the Interrupt Mask (IMR) register, the timer also sets the CnMMIS bit in the Masked Interrupt Status (MIS) register. In this mode, the timer register (TnR) and timer prescale register (TnPS) hold the count of the input events while the timer value register (TnV) and timer prescale value register (TnPV) hold the free-running timer value and the free-running prescaler value.

Edge Time Mode

The timer is configured as a 24-bit or 48-bit up- or down-counter including the optional prescaler. By writing the TnEN bit in the CTL register, the timer is started. The timer is initialized to the value loaded in the timer load register (TnILR) and timer prescale register (TnPR) when counting down and 0x0 when counting up. When the timer reaches the timeout value, it is reloaded with 0x0 in up-count mode and the value from the timer interrupt load register (TnILR) and timer prescale register (TnPR) in down-count mode.

The timer is placed into Edge-Time mode by setting the TnCMR bit in the timer mode (TnMR) register, and the type of event that the timer captures is determined by the TnEVENT fields of the timer control (CTL) register. The timer can be configured for capturing three types of events: rising edge, falling edge, or both. When the selected input event is detected, the current timer counter value is captured in the timer register (TnR) and timer prescale (TnPS) register and is available to be read by the microcontroller.

When the selected input event is detected, the timer then asserts the CnERIS bit in the Raw Interrupt Status (RIS) register, and holds it until it is cleared by writing the Interrupt Clear (ICR) register. If the capture mode event interrupt is enabled in the Interrupt Mask (IMR) register, the also sets the CnEMIS bit in the Masked Interrupt Status (MIS) register. In this mode, the timer register (TnR) and prescale register (TnPS) hold the time at which the selected input event occurred while the timer value (TnV) and timer prescale value (TnPV) registers hold the free-running timer value and the free-running prescaler value. These registers can be read to determine the time that elapsed between the interrupt assertion and the entry into the ISR.

After an event has been captured, the timer does not stop counting. It continues to count until the TnEN bit in the CTL register is cleared.

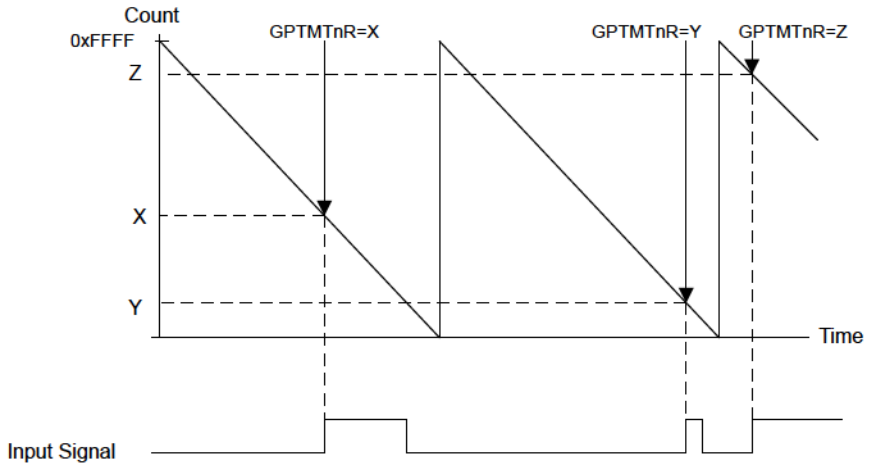


Figure 7.3: The start value of the timer is the default value of 0xFFFF, and the timer is configured to capture rising edge events.

2.3.4.PWM Mode

PWM mode is enabled with the timer mode (TnMR) register by setting the TnAMS bit to 0x1, the TnCMR bit to 0x0, and the TnMR field to 0x2. The timer is configured as a 24-bit or 48-bit down-counter with a start value (and thus period) defined by the timer interval load (TnILR) and timer prescale (TnPR) registers. When software writes the TnEN bit in the CTL register, the counter begins counting down until it reaches the 0x0 state. On the next counter cycle in periodic mode, the counter reloads its start value from the timer interval load (TnILR) and timer prescale (TnPR) registers and continues counting until disabled by software clearing the TnEN bit in the CTL register.

If the TnWOT bit is set in the timer mode (TnMR) register, once the TnEN bit is set, the timer waits for a trigger to begin counting.

The timer is capable of generating interrupts based on three types of events: rising edge, falling edge, or both. The event is configured by the TnEVENT field of the control (CTL) register, and the interrupt is enabled by setting the TnPWMIE bit in the mode (TnMR) register. When the event occurs, the CnERIS bit is set in the Raw Interrupt Status (RIS) register, and holds it until it is cleared by writing the Interrupt Clear (ICR) register. If the capture mode event interrupt is enabled in the Interrupt Mask (IMR) register, the timer also sets the CnEMIS bit in the Masked Interrupt Status (MIS) register. Note that the interrupt status bits are not updated unless the TnPWMIE bit is set.

In this mode, the timer register (TnR) and timer value (TnV) registers always have the same value, as do the timer prescale (TnPS) and the timer prescale value (TnPV) registers.

The output PWM signal asserts when the counter is at the value of the timer interval load (TnILR) and timer prescale (TnPR) registers (its start state), and is deasserted when the counter value equals the value in the timer match (TnMATCHR) and timer prescale match (TnPMR) registers. Software has the capability of inverting the output PWM signal by setting the TnPWML bit in the control (CTL) register.

Example: Assuming a 16-MHz clock and TnPWML =0 (duty cycle is 66%, it would be 33% for the TnPWML=1 configuration). The start value TnILR =0xC350 and the match value TnMATCHR =0x411A.

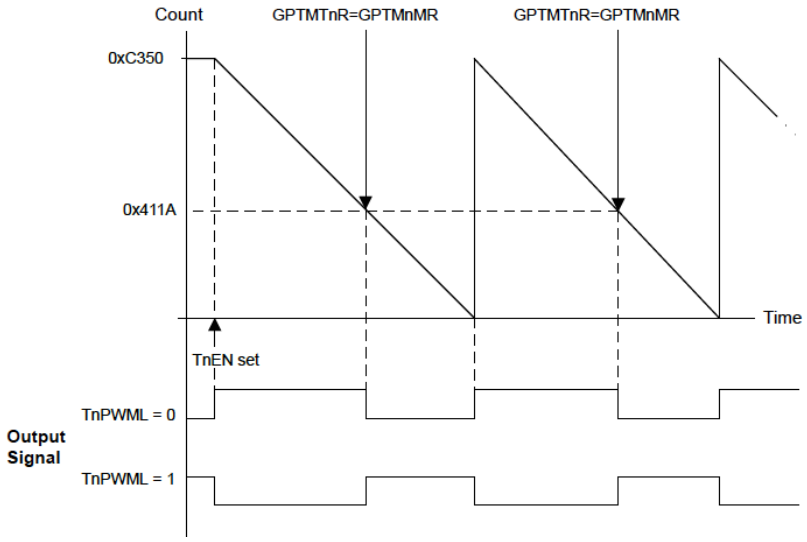


Figure17.4: An output PWM with 66% duty cycle.

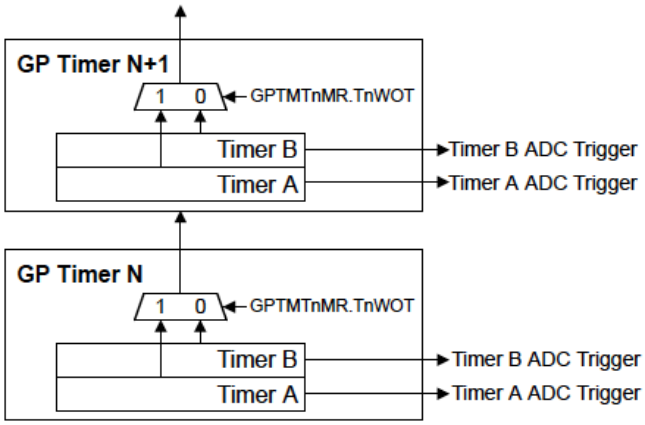
0xC350 = C*(16)^3+3*(16)^2+5*(16)+0*1=50.000
0x411A = 16.666
Clock period = 1/16MHz = 62,5 nsec
Signal period = 50.000*62,5 nsec = 3.125 msec
Signal frequency = 1/3.125 msec = 320 Hz
Duty cycle = (50.000 - 16.666)/50.000 = 66,6%

2.3.5. Wait-for-Trigger Mode

The Wait-for-Trigger mode allows daisy chaining of the timer modules such that once configured, a single timer can initiate multiple timing events using the Timer triggers. This function is valid for one-shot, periodic and PWM modes.

Wait-for-Trigger mode is enabled by setting the TnWOT bit in the timer mode (TnMR) register. When the TnWOT bit is set, Timer N+1 does not begin counting until the timer in the previous position in the daisy chain (Timer N) reaches its time-out event.

If Timer A is configured as a 32-bit (16/32-bit mode) or 64-bit (32/64-bit wide mode) timer (controlled by the CFG field in the configuration CFG register), it triggers Timer A in the next module. If Timer A is configured as a 16-bit (16/32-bit mode) or 32-bit (32/64-bit wide mode) timer, it triggers Timer B in the same module, and Timer B triggers Timer A in the next module.



2.4. Synchronizing GP Timer Blocks

The Synchronizer Control (SYNC) register in module 0 can be used to synchronize selected timers to begin counting at the same time. Setting a bit in the SYNC register causes the associated timer to perform the actions of a timeout event. An interrupt is not generated when the timers are synchronized. If a timer is being used in concatenated mode, only the bit for Timer A must be set in the SYNC register.

2.5. Registers

Timer system registers are given below. The offset values of these registers are used to reach the registers relative to that timer module's base address. Base addresses of six 16/32 bit Timer and six 32/64 bit Timer are listed below:

■ 16/32-bit Timer 0: 0x4003.0000 ■ 16/32-bit Timer 1: 0x4003.1000 ■ 16/32-bit Timer 2: 0x4003.2000 ■ 16/32-bit Timer 3: 0x4003.3000 ■ 16/32-bit Timer 4: 0x4003.4000 ■ 16/32-bit Timer 5: 0x4003.5000	■ 32/64-bit Wide Timer 0: 0x4003.6000 ■ 32/64-bit Wide Timer 1: 0x4003.7000 ■ 32/64-bit Wide Timer 2: 0x4004.C000 ■ 32/64-bit Wide Timer 3: 0x4004.D000 ■ 32/64-bit Wide Timer 4: 0x4004.E000 ■ 32/64-bit Wide Timer 5: 0x4004.F000

Offset	Name	Type	Reset	Description
0x000	GPTMCFG	RW	0x0000.0000	GPTM Configuration
0x004	GPTMTAMR	RW	0x0000.0000	GPTM Timer A Mode
0x008	GPTMTBMR	RW	0x0000.0000	GPTM Timer B Mode
0x00C	GPTMCTL	RW	0x0000.0000	GPTM Control
0x010	GPTMSYNC	RW	0x0000.0000	GPTM Synchronize
0x018	GPTMIMR	RW	0x0000.0000	GPTM Interrupt Mask
0x01C	GPTMRIS	RO	0x0000.0000	GPTM Raw Interrupt Status
0x020	GPTMMIS	RO	0x0000.0000	GPTM Masked Interrupt Status
0x024	GPTMICR	W1C	0x0000.0000	GPTM Interrupt Clear
0x028	GPTMTAILR	RW	0xFFFF.FFFF	GPTM Timer A Interval Load
0x02C	GPTMTBILR	RW	-	GPTM Timer B Interval Load
0x030	GPTMTAMATCHR	RW	0xFFFF.FFFF	GPTM Timer A Match
0x034	GPTMTBMATCHR	RW	-	GPTM Timer B Match
0x038	GPTMTAPR	RW	0x0000.0000	GPTM Timer A Prescale
0x03C	GPTMTBPR	RW	0x0000.0000	GPTM Timer B Prescale
0x040	GPTMTAPMR	RW	0x0000.0000	GPTM TimerA Prescale Match
0x044	GPTMTBPMR	RW	0x0000.0000	GPTM TimerB Prescale Match
0x048	GPTMTAR	RO	0xFFFF.FFFF	GPTM Timer A
0x04C	GPTMTBR	RO	-	GPTM Timer B
0x050	GPTMTAV	RW	0xFFFF.FFFF	GPTM Timer A Value
0x054	GPTMTBV	RW	-	GPTM Timer B Value
0x058	GPTMRTCPD	RO	0x0000.7FFF	GPTM RTC Predivide
0x05C	GPTMTAPS	RO	0x0000.0000	GPTM Timer A Prescale Snapshot
0x060	GPTMTBPS	RO	0x0000.0000	GPTM Timer B Prescale Snapshot
0x064	GPTMTAPV	RO	0x0000.0000	GPTM Timer A Prescale Value
0x068	GPTMTBPV	RO	0x0000.0000	GPTM Timer B Prescale Value
0xFC0	GPTMPP	RO	0x0000.0000	GPTM Peripheral Properties

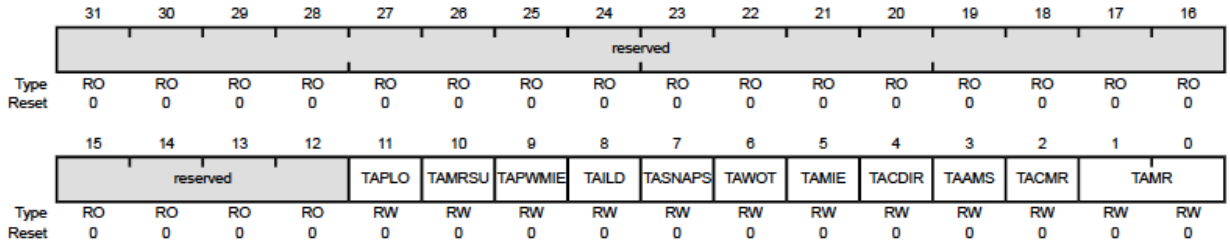
Timer Configuration (CFG), offset 0x000

This register configures the global operation of the module. The value written to this register determines whether the timer is in 32- or 64-bit mode (concatenated timers) or in 16- or 32-bit mode (individual, split timers). Bits other than 2:0 are reserved.

Value of Bit field 2:0	Description
0x0	For a 16/32-bit timer, this value selects the 32-bit timer configuration. For a 32/64-bit wide timer, this value selects the 64-bit timer configuration.
0x1	For a 16/32-bit timer, this value selects the 32-bit real-time clock (RTC) counter configuration. For a 32/64-bit wide timer, this value selects the 64-bit real-time clock (RTC) counter configuration.
0x4	For a 16/32-bit timer, this value selects the 16-bit timer configuration. For a 32/64-bit wide timer, this value selects the 32-bit timer configuration.

Timer A/B Mode (TAMR/ TBMR), offset 0x004/0x008

This register configures the timer based on the configuration selected in the CFG register. This register controls the modes for Timer A/B when it is used individually. When Timer A and Timer B are concatenated, TAMR controls the modes for both Timer A and Timer B, and the contents of TBMR are ignored.

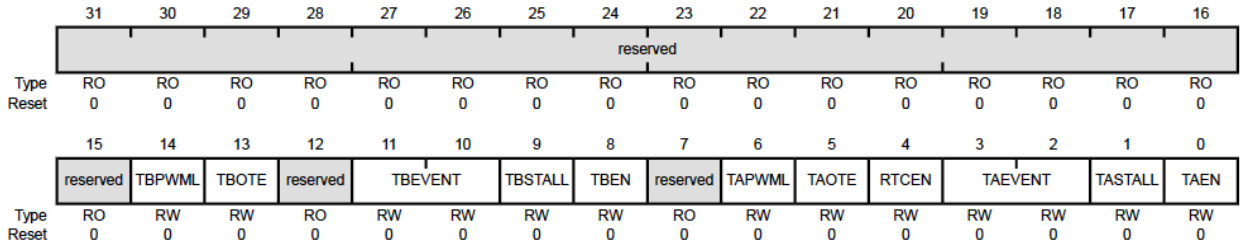


Bit field	Value	Description
1:0	0x1	One shot timer mode
	0x2	Periodic timer mode
	0x3	Capture mode
2	0	Edge count mode
	1	Edge time mode
3	0	Capture or compare mode
	1	PWM
4	0	Timer counts down
	1	Timer counts up
5	0	The match interrupt is disabled for match events.
	1	An interrupt is generated when the match value in the TAMATCHR register is reached in the one-shot and periodic modes.
6	0	Timer A begins counting as soon as it is enabled
	1	If Timer A is enabled (TAEN is set in the CTL register), Timer A does not begin counting until it receives a trigger from the timer in the previous position in the daisy chain
7	0	Snap shot mode is disabled
	1	If Timer A is configured in the periodic mode, the actual free-

		running, capture or snapshot value of Timer A is loaded at the time-out event/capture or snapshot event into the Timer A (TAR) register. If the timer prescaler is used, the prescaler snapshot is loaded into the timer Timer A (TAPR).
8	0	Update the TAR and TAV registers with the value in the TAILR register on the next cycle. Also update the TAPS and TAPV registers with the value in the TAPR register on the next cycle.
	1	Update the TAR and TAV registers with the value in the TAILR register on the next timeout. Also update the TAPS and TAPV registers with the value in the TAPR register on the next timeout.
9	0	Capture event interrupt is disabled.
	1	Capture event interrupt is enabled.
10	0	Update the TAMATCHR register and the TAPR register, if used, on the next cycle.
	1	Update the TAMATCHR register and the TAPR register, if used, on the next time out.
11	0	Legacy operation with CCP pin driven Low when the TAILR is reloaded after the timer reaches 0.
	1	CCP is driven High when the TAILR is reloaded after the timer reaches 0.

Timer Control (CTL), offset 0x00C

This register is used alongside the CFG and GMTMTAMR/GMTMTBMR registers to fine-tune the timer configuration, and to enable other features such as timer stall and the output trigger. The output trigger can be used to initiate transfers on the ADC module.

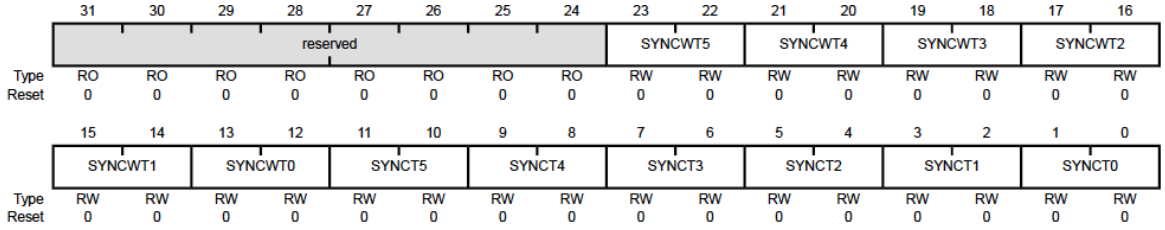


Bit field	Value	Description
14	0	Output is unaffected.
	1	Output is inverted.
13	0	The output Timer B ADC trigger is disabled.
	1	The output Timer B ADC trigger is enabled.
11:10	0x0	Timer B Event Mode, Positive edge
	0x1	Timer B Event Mode, Negative edge
	0x3	Timer B Event Mode, both edges
9	0	Timer B continues counting while the processor is halted by the debugger.
	1	Timer B freezes counting while the processor is halted by the debugger.
8	0	Timer B is disabled.
	1	Timer B is enabled and begins counting or the capture logic is enabled based on the CFG register.
6	0	Timer A PWM Output Level is unaffected.

	1	Timer A PWM Output Level is inverted.
5	0	The output Timer A ADC trigger is disabled.
	1	The output Timer A ADC trigger is enabled.
4	0	RTC counting freezes while the processor is halted by the debugger.
	1	RTC counting continues while the processor is halted by the debugger.
3:2	0x0	Timer A Event Mode, Positive edge
	0x1	Timer A Event Mode, Negative edge
	0x3	Timer A Event Mode, both edges
1	0	Timer A continues counting while the processor is halted by the debugger.
	1	Timer A freezes counting while the processor is halted by the debugger.
0	0	Timer A is disabled.
	1	Timer A is enabled and begins counting or the capture logic is enabled based on the CFG register.

Timer Synchronize (SYNC), offset 0x010

This register is only implemented on timer Module 0 only. This register allows software to synchronize a number of timers.

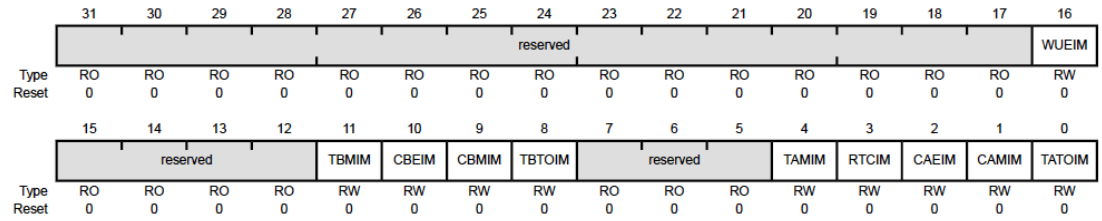


Two bits for each timer to synchronize. For a timer Timer n, bit values and descriptions are as follows:

0x0 Timer n is not affected.
0x1 A timeout event for Timer A of Timer n is triggered.
0x2 A timeout event for Timer B of Timer n is triggered.
0x3 A timeout event for both Timer A and Timer B of Timer n is triggered.

Timer Interrupt Mask (IMR), offset 0x018

This register allows software to enable/disable timer controller-level interrupts. Setting a bit enables the corresponding interrupt, while clearing a bit disables it.

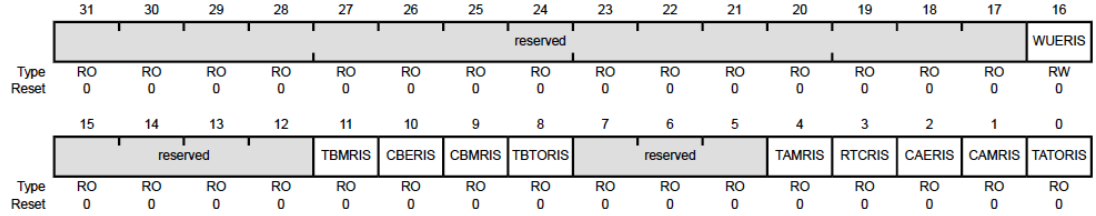


bit	function
16	32/64-Bit Wide timer Write Update (WUE) Error Interrupt Mask

11	Timer B Match (TBM) Interrupt Mask
10	Timer B Capture Mode Event (CBE) Interrupt Mask
9	Timer B Capture Mode Match (CBM) Interrupt Mask
8	Timer B Time-Out (TBTO) Interrupt Mask
4	Timer A Match (TAM) Interrupt Mask
3	RTC Interrupt Mask
2	Timer A Capture Mode Event (CAE) Interrupt Mask
1	Timer A Capture Mode Match (CAM) Interrupt Mask
0	Timer A Time-Out (TATO) Interrupt Mask

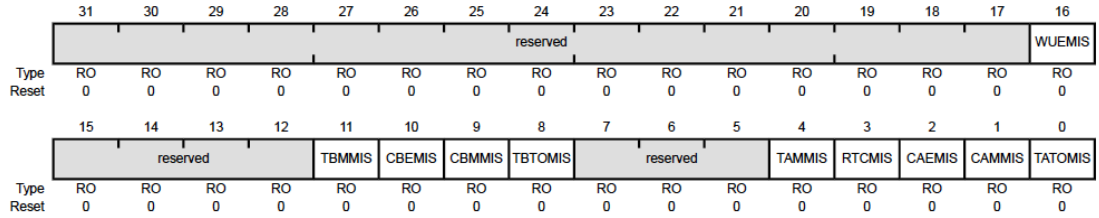
Timer Raw Interrupt Status (RIS), offset 0x01C

This register shows the state of the timer's internal interrupt signal. These bits are set whether or not the interrupt is masked in the IMR register. Each bit can be cleared by writing a 1 to its corresponding bit in ICR.



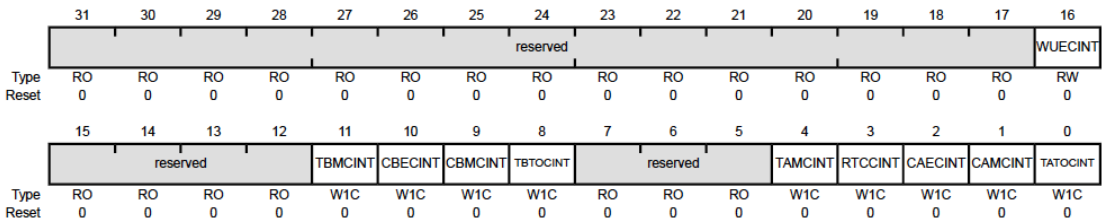
Timer Masked Interrupt Status (MIS), offset 0x020

This register shows the state of the timer's controller-level interrupt. If an interrupt is unmasked in IMR, and there is an event that causes the interrupt to be asserted, the corresponding bit is set in this register. All bits are cleared by writing a 1 to the corresponding bit in ICR.



Timer Interrupt Clear (ICR), offset 0x024

This register is used to clear the status bits in the RIS and MIS registers. Writing a 1 to a bit clears the corresponding bit in the RIS and MIS registers.



Timer A/B Interval Load (TAILR/ TBILR), offset 0x028/0x02C

When the timer is counting down, this register is used to load the starting count value into the timer. When the timer is counting up, this register sets the upper bound for the timeout event.

When a 16/32-bit is configured to one of the 32-bit modes, TAILR/ TBILR appears as a 32-bit register (the upper 16-bits correspond to the contents of the TBILR and the lower 16 bits to TAILR). In a 16-bit mode, the upper 16 bits of this register read as 0s.

When a 32/64-bit Wide is configured to one of the 64-bit modes, TAILR contains bits 31:0 of the 64-bit count and TBILR register contains bits 63:32.

Timer A/B Match (TAMATCHR/ TBMATCHR), offset 0x030/0x034

This register is loaded with a match value. Interrupts can be generated when the timer value is equal to the value in this register in one-shot or periodic mode.

In Edge-Count mode, this register along with TAILR/ TBILR, determines how many edge events are counted. The total number of edge events counted is equal to the value in TAILR/ TAILR minus this value. Note that in edge-count mode, when executing an up-count, the value of TnPR and TnILR must be greater than the value of TnPMR and TnMATCHR.

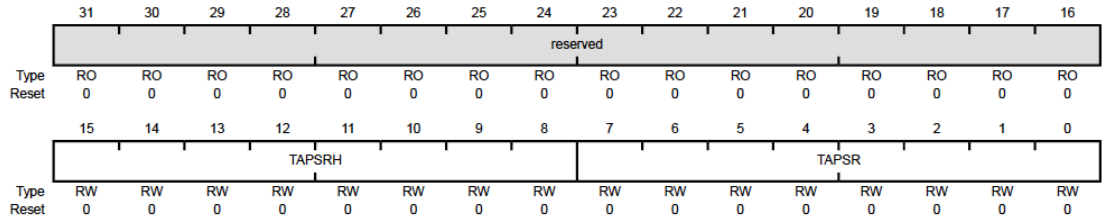
In PWM mode, this value along with TAILR/ TBILR, determines the duty cycle of the output PWM signal.

When a 16/32-bit is configured to one of the 32-bit modes, TAMATCHR/ TBMATCHR appears as a 32-bit register (the upper 16-bits correspond to the contents of TBMATCHR register, the lower 16-bits to TAMATCHR). When a 32/64-bit Wide is configured to one of the 64-bit modes, TAMATCHR contains bits 31:0 of the 64-bit match value and TBMATCHR register contains bits 63:32.

Timer A/B Prescale (TAPR/ TBPR), offset 0x038/0x03C

This register allows software to extend the range of the timers when they are used individually. When in one-shot or periodic down count modes, this register acts as a true prescaler for the timer counter. When acting as a true prescaler, the prescaler counts down to 0 before the value in the TAR/ TBR and TAV/ TBV registers are incremented. In all other individual/split modes, this register is a linear extension of the upper range of the timer counter, holding bits 23:16 in the 16-bit modes of the 16/32-bit and bits 47:32 in the 32-bit modes of the 32/64-bit Wide .

Note that the prescaler can only be used when the timers are used individually. Input edge count, input edge time and PWM mode, the prescaler always acts as a timer extension, regardless of the count direction.



Timer A/B Prescale Match (TAPMR/ TBPMR), offset 0x040/0x044

This register allows software to extend the range of the TAMATCHR/ TAMATCHR when the timers are used individually. This register holds bits 23:16 in the 16-bit modes of the 16/32-bit and bits 47:32 in the 32-bit modes of the 32/64-bit Wide.

Timer A/B (TAR/ TBR), offset 0x048/0x04C

This read-only register shows the current value of the Timer A/B counter in all cases except for Input Edge Count and Edge Time modes. In the Input Edge Count mode, this register contains the number of edges that have occurred. In the Input Edge Time mode, this register contains the time at which the last edge event took place.

When a 16/32-bit is configured to one of the 32-bit modes, TAR appears as a 32-bit register (the upper 16-bits correspond to the contents of the TBR).

In the16-bit Input Edge Count, Input Edge Time, and PWM modes, bits 15:0 contain the value of the counter and bits 23:16 contain the value of the prescaler, which is the upper 8 bits of the count.

Bits 31:24 always read as 0. To read the value of the prescaler in 16-bit One-Shot and Periodic modes, read bits [23:16] in the TAV/ TBV register. To read the value of the prescalar in periodic snapshot mode, read the Timer A/B Prescale Snapshot (TAPS/ TBPS) register.

When a 32/64-bit Wide is configured to one of the 64-bit modes, TAR contains bits 31:0 of the 64-bit timer value and TBR register contains bits 63:32. In a 32-bit mode, the value of the prescaler is stored in the Timer A/B Prescale Snapshot (TAPS/ TBPS) register.

Timer A/B Value (TAV/ TBV), offset 0x050/0x054

When read, this read/write register shows the current, free-running value of Timer A/B in all modes. Software can use this value to determine the time elapsed between an interrupt and the ISR entry when using the snapshot feature with the periodic operating mode. When written, the value written into this register is loaded into the TAR/ TBR register on the next clock cycle.

When a 16/32-bit is configured to one of the 32-bit modes, TAV appears as a 32-bit register (the upper 16-bits correspond to the contents of the TBV register). In a 16-bit mode, bits 15:0 contain the value of the counter and bits 23:16 contain the current, free-running value of the prescaler, which is the upper 8 bits of the count in Input Edge Count, Input Edge Time, PWM and one-shot or periodic up count modes. In one-shot or periodic down count modes, the prescaler stored in 23:16 is a true prescaler, meaning bits 23:16 count down before decrementing the value in bits 15:0. The prescaler in bits 31:24 always reads as 0.

When a 32/64-bit Wide is configured to one of the 64-bit modes, TAV contains bits 31:0 of the 64-bit timer value and the TBV register contains bits 63:32. In a 32-bit mode, the current, free-running value of the prescaler is stored in the TAPV/ TBPV.

Timer RTC Predivide (RTCPD), offset 0x058

This register provides the current RTC predivider value when the timer is operating in RTC mode. Software must perform an atomic access with consecutive reads of the TAR, TBR, and RTCPD registers.

Timer A/B Prescale Snapshot (TAPS/ TBPS), offset 0x05C/0x060

For the 32/64-bit Wide, this register shows the current value of the Timer A/B prescaler in the 32-bit modes. For 16-/32-bit wide, this register shows the current value of the Timer A/B prescaler for periodic snapshot mode.

Timer A/B Prescale Value (TAPV/ TBPV), offset 0x064/0x068

For the 32/64-bit Wide , this register shows the current free-running value of the Timer A prescaler in the 32-bit modes. Software can use this value in conjunction with the TAV/ TBV register to determine the time elapsed between an interrupt and the ISR entry. This register is ununsed in 16/32-bit mode.

Timer Peripheral Properties (PP), offset 0xFC0

The PP register provides information regarding the properties of the General-Purpose Timer module.

Bit field 3:0 defines the size where

0 Timer A and Timer B counters are 16 bits each with an 8-bit prescale counter.
1 Timer A and Timer B counters are 32 bits each with a 16-bit prescale counter.

2.6. Initialization and Configuration

2.6.1. One-Shot/Periodic Timer Mode

The timer is configured for One-Shot and Periodic modes by the following sequence:

1. Ensure the timer is disabled (the TnEN bit in the CTL register is cleared) before making any changes.
2. Write the Configuration Register (CFG) with a value of 0x0000.0000.
3. Configure the TnMR field in the Timer n Mode Register (TnMR):
    a. Write a value of 0x1 for One-Shot mode.
    b. Write a value of 0x2 for Periodic mode.
4. Optionally configure the TnSNAPS, TnWOT, TnMTE, and TnCDIR bits in the TnMR register to select whether to capture the value of the free-running timer at time-out, use an external trigger to start counting, configure an additional trigger or interrupt, and count up or down.
5. Load the start value into the Timer n Interval Load Register (TnILR).
6. If interrupts are required, set the appropriate bits in the Interrupt Mask Register (IMR).
7. Set the TnEN bit in the CTL register to enable the timer and start counting.
8. Poll the RIS register or wait for the interrupt to be generated (if enabled). In both cases, the status flags are cleared by writing a 1 to the appropriate bit of the Interrupt Clear Register (ICR).

If the TnMIE bit in the TnMR register is set, the RTCRIS bit in the RIS register is set, and the timer continues counting. In One-Shot mode, the timer stops counting after the time-out event. To re-enable the timer, repeat the sequence. A timer configured in Periodic mode reloads the timer and continues counting after the time-out event.

2.6.2.Real-Time Clock (RTC) Mode

To use the RTC mode, the timer must have a 32.768-KHz input signal on an even CCP input. To enable the RTC feature, follow these steps:

1. Ensure the timer is disabled (the TAEN bit is cleared) before making any changes.
2. If the timer has been operating in a different mode prior to this, clear any residual set bits in the Timer n Mode (TnMR) register before reconfiguring.
3. Write the Configuration Register (CFG) with a value of 0x0000.0001.
4. Write the match value to the Timer n Match Register (TnMATCHR).
5. Set/clear the RTCEN and TnSTALL bit in the Control Register (CTL) as needed.
6. If interrupts are required, set the RTCIM bit in the Interrupt Mask Register (IMR).
7. Set the TAEN bit in the CTL register to enable the timer and start counting.

When the timer count equals the value in the TnMATCHR register, the timer asserts the RTCRIS bit in the RIS register and continues counting until Timer A is disabled or a hardware reset. The interrupt is cleared by writing the RTCCINT bit in the ICR register. Note that if the TnILR register is loaded with a new value, the timer begins counting at this new value and continues until it reaches 0xFFFF.FFFF, at which point it rolls over.

2.6.3.Input Edge-Count Mode

A timer is configured to Input Edge-Count mode by the following sequence:

1. Ensure the timer is disabled (the TnEN bit is cleared) before making any changes.
2. Write the Configuration (CFG) register with a value of 0x0000.0004.
3. In the Timer Mode (TnMR) register, write the TnCMR field to 0x0 and the TnMR field to 0x3.
4. Configure the type of event(s) that the timer captures by writing the TnEVENT field of the Control (CTL) register.
5. Program registers according to count direction:
    a. In down-count mode, the TnMATCHR and TnPMR registers are configured so that the difference between the value in the TnILR and TnPR registers and the TnMATCHR and TnPMR registers equals the number of edge events that must be counted.
    b. In up-count mode, the timer counts from 0x0 to the value in the TnMATCHR and TnPMR registers. Note that when executing an up-count, the value of the TnPR and TnILR must be greater than the value of TnPMR and TnMATCHR .
6. If interrupts are required, set the CnMIM bit in the Interrupt Mask (IMR) register.
7. Set the TnEN bit in the CTL register to enable the timer and begin waiting for edge events.
8. Poll the CnMRIS bit in the RIS register or wait for the interrupt to be generated (if enabled). In both cases, the status flags are cleared by writing a 1 to the CnMCINT bit of the Interrupt Clear (ICR) register.

When counting down in Input Edge-Count Mode, the timer stops after the programmed number of edge events has been detected. To re-enable the timer, ensure that the TnEN bit is cleared and repeat steps 4 through 8.

2.6.4.Input Edge Time Mode

A timer is configured to Input Edge Time mode by the following sequence:

1. Ensure the timer is disabled (the TnEN bit is cleared) before making any changes.
2. Write the Configuration (CFG) register with a value of 0x0000.0004.
3. In the Timer Mode (TnMR) register, write the TnCMR field to 0x1 and the TnMR field to 0x3 and select a count direction by programming the TnCDIR bit.

4. Configure the type of event that the timer captures by writing the TnEVENT field of the Control (CTL) register.
5. If a prescaler is to be used, write the prescale value to the Timer n Prescale Register (TnPR).
6. Load the timer start value into the Timer n Interval Load (TnILR) register.
7. If interrupts are required, set the CnEIM bit in the Interrupt Mask (IMR) register.
8. Set the TnEN bit in the Control (CTL) register to enable the timer and start counting.
9. Poll the CnERIS bit in the RIS register or wait for the interrupt to be generated (if enabled).

In both cases, the status flags are cleared by writing a 1 to the CnECINT bit of the Interrupt Clear (ICR) register. The time at which the event happened can be obtained by reading the Timer n (TnR) register.

In Input Edge Timing mode, the timer continues running after an edge event has been detected, but the timer interval can be changed at any time by writing the TnILR register and clearing the TnILD bit in the TnMR register. The change takes effect at the next cycle after the write.

2.6.5.PWM Mode

A timer is configured to PWM mode using the following sequence:

1. Ensure the timer is disabled (the TnEN bit is cleared) before making any changes.
2. Write the Configuration (CFG) register with a value of 0x0000.0004.
3. In the Timer Mode (TnMR) register, set the TnAMS bit to 0x1, the TnCMR bit to 0x0, and the TnMR field to 0x2.
4. Configure the output state of the PWM signal (whether or not it is inverted) in the TnPWML field of the Control (CTL) register.
5. If a prescaler is to be used, write the prescale value to the Timer n Prescale Register (TnPR).
6. If PWM interrupts are used, configure the interrupt condition in the TnEVENT field in the CTL register and enable the interrupts by setting the TnPWMIE bit in the TnMR register. Note that edge detect interrupt behavior is reversed when the PWM output is inverted.
7. Load the timer start value into the Timer n Interval Load (TnILR) register.
8. Load the Timer n Match (TnMATCHR) register with the match value.
9. Set the TnEN bit in the Control (CTL) register to enable the timer and begin generation of the output PWM signal.

In PWM Time mode, the timer continues running after the PWM signal has been generated. The PWM period can be adjusted at any time by writing the TnILR register, and the change takes effect at the next cycle after the write.

2.7. Programming Examples

Set up Port B for signal input and Timer 0 as 16 bit and edge time mode:

```
; Timer channel registers for TIMER0:
TIMER0_CFG      EQU  0x40030000  ; Configuration Register
TIMER0_TAMR     EQU  0x40030004  ; Mode Register
TIMER0_CTL      EQU  0x4003000C  ; Control Register
TIMER0_RIS      EQU  0x4003001C  ; Raw interrupt Status
TIMER0_ICR      EQU  0x40030024  ; Interrupt Clear Register
TIMER0_TAILR    EQU  0x40030028  ; Interval Load Register
TIMER0_TAMATCHR EQU  0x40030030  ; Match Register
```

```
TIMER0_TAPR             EQU   0x40030038   ; Prescaling Divider
TIMER0_TAR              EQU   0x40030048   ; Counter Register

; Timer Gate Control
SYSCTL_RCGCTIMER        EQU   0x400FE604   ; Timer Clock Gating

;GPIO Registers for Port B
;Port B base 0x40005000
GPIO_PORTB_IM           EQU   0x40005010   ; Interrupt Mask
GPIO_PORTB_DIR          EQU   0x40005400   ; Port Direction
GPIO_PORTB_AFSEL        EQU   0x40005420   ; Alt Function enable
GPIO_PORTB_DEN          EQU   0x4000551C   ; Digital Enable
GPIO_PORTB_AMSEL        EQU   0x40005528   ; Analog enable
GPIO_PORTB_PCTL         EQU   0x4000552C   ; Alternate Functions

;GPIO Gate Control Register
SYSCTL_RCGCGPIO         EQU   0x400FE608

; Setup Port B for signal input
; set direction of PB6
        LDR R1, =GPIO_PORTB_DIR
        LDR R0, [R1]
        BIC R0, R0, #0x40           ; clear bit 6 for input
        STR R0, [R1]
; enable alternate function
        LDR R1, =GPIO_PORTB_AFSEL
        LDR R0, [R1]
        ORR R0, R0, #0x40           ; set bit6 for alternate fuction on PB6
        STR R0, [R1]
; set alternate function to T0CCP0 (7)
        LDR R1, =GPIO_PORTB_PCTL
        LDR R0, [R1]
        ORR R0, R0, #0x07000000  ; set bits 27:24 of PCTL to 7
        STR R0, [R1]                ; to enable T0CCP0 on PB6
; disable analog
        LDR R1, =GPIO_PORTB_AMSEL
        MOV R0, #0                  ; clear AMSEL to diable analog
        STR R0, [R1]
; Start Timer 0 clock
        LDR R1, =SYSCTL_RCGCTIMER
        LDR R2, [R1]                ; Start timer 0
        ORR R2, R2, #0x01           ; Timer module = bit position (0)
        STR R2, [R1]
        NOP
        NOP
        NOP                         ; allow clock to settle
; disable timer during setup
        LDR R1, =TIMER0_CTL
```

```
        LDR R2, [R1]
        BIC R2, R2, #0x01          ; clear bit 0 to disable Timer 0
        STR R2, [R1]
; set to 16bit Timer Mode
        LDR R1, =TIMER0_CFG
        MOV R2, #0x04              ; set bits 2:0 to 0x04 for 16bit timer
        STR R2, [R1]
; set for edge time and capture mode
        LDR R1, =TIMER0_TAMR
        MOV R2, #0x07              ; set bit2 to 0x01 for Edge Time Mode,
        STR R2, [R1]               ; set bits 1:0 to 0x03 for Capture Mode
;; set edge detection to both
        ;LDR R1, =TIMER0_CTL
        ;LDR R2, [R1]
        ;ORR R2, R2, #0x0C         ; set bits 3:2 to 0x03
        ;STR R2, [R1]
; set start value
        LDR R1, =TIMER0_TAILR      ; counter counts down,
        MOV R0, #0xFFFFFFFF        ; so start counter at max value
        STR R0, [R1]
; Enable timer
        LDR R1, =TIMER0_CTL ;
        LDR R2, [R1] ;
        ORR R2, R2, #0x01          ; set bit 0 to enable
        STR R2, [R1]
; Await edge capture event
        LDR R1, =TIMER0_RIS
loop    LDR R2, [R1]
        ANDS R2 #04                ; isolate CAERIS bit
        BEQ loop                   ; if no capture, then loop
; Need to clear CAERIS bit of TIMER0_RIS.
; Write this part
        LDR R1, =TIMER0_TAR        ; address of timer register
        LDR R0, [R1]               ; Get timer register value
; Now use this data, with other measured data to compute
; period, pulse width, duty cycle, frequency,...
;home exercise☺
```

System Tick Timer setup subroutine:

```
; SysTick_Init
; disable SysTick during setup
        LDR R1, =NVIC_ST_CTRL_R
        MOV R0, #0                 ; Clear Enable
        STR R0, [R1]
; set reload to maximum reload value
        LDR R1, =NVIC_ST_RELOAD_R
        LDR R0, =0x00FFFFFF        ; Specify RELOAD value
```

```
        STR R0, [R1]            ; reload at maximum
; writing any value to CURRENT clears it
        LDR R1, =NVIC_ST_CURRENT_R
        STR R0, [R1]            ; clear counter
; enable SysTick with core clock
        LDR R1, =NVIC_ST_CTRL_R
        MOV R0, #0x0005         ; Enable but no interrupts (later)
        STR R0, [R1]            ; ENABLE with System Clock
        BX LR                   ; Return from subroutine
```

Time delay subroutine (by polling):

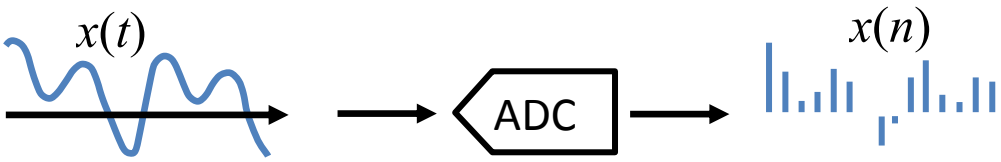
```
; Time delay using busy wait.
; Input: R0 delay parameter in units of the core clock
; At 16 MHz, tick is 62.5 nsec
; at 80 MHz, 12.5 nsec
; Output: none
; Modifies: R1
SysTick_Wait
        SUB R0, R0, #1                  ; delay-1
        LDR R1, =NVIC_ST_RELOAD_R
        STR R0, [R1]                    ; time to wait
        LDR R1, =NVIC_ST_CURRENT_R
        STR R0, [R1]                    ; any value written to CURRENT clears
        LDR R1, =NVIC_ST_CTRL_R
SysTick_Wait_loop
        LDR R0, [R1]                    ; read status
        ANDS R0, R0, #0x00010000        ; bit 16 is COUNT flag
        BEQ SysTick_Wait_loop           ; repeat until flag set
        BX LR                           ; flag set, so return from sub

; Delay using SysTick_Wait subroutine

;------------SysTick_Wait10ms------------
; Call this routine to wait for R0*10 ms
; Time delay using busy wait. This assumes 16 MHz clock
; Input: R0 number of times to wait 10 ms before returning
; Output: none
; Modifies: R0
DELAY10MS  EQU   160000                 ; clock cycles for 10 ms delay
SysTick_Wait10ms
                PUSH {R4, LR}           ; save R4 and LR
                MOVS R4, R0             ; R4 = R0 = remainingWaits
                BEQ SysTick_Wait10ms_done ; R4 == 0, done
; SysTick_Wait10ms_loop
                LDR R0, =DELAY10MS      ; R0 = DELAY10MS
                BL SysTick_Wait         ; wait 10 ms
                SUBS R4, R4, #1         ; remainingWaits--
```

```
                BHI SysTick_Wait10ms_loop       ; if(R4>0), wait another 10 ms
; SysTick_Wait10ms_done
                POP {R4, PC}                    ; pop R4 and Return
```

Chapter 8

Analog to Digital Converter

Analog refers to physical quantities that vary continuously instead of discretely. Physical phenomena typically involve analog signals. Examples include temperature, speed, position, pressure, voltage, altitude, etc. Microprocessors work with digital quantities (values taken from the discrete domain). For a digital system to interact with analog systems, conversion between analog and digital values is needed. Building blocks to perform the conversions are: (1) Digital to analog converters (DACs), (2) Analog to digital converters (ADCs). A digital to analog converter has a digital input that specifies an output whose value changes in steps. These step changes are in volts or amperes. The analog to digital converter has an input that can vary from a minimum to a maximum value of volts or amperes. The output is a digital number that represents the input value.



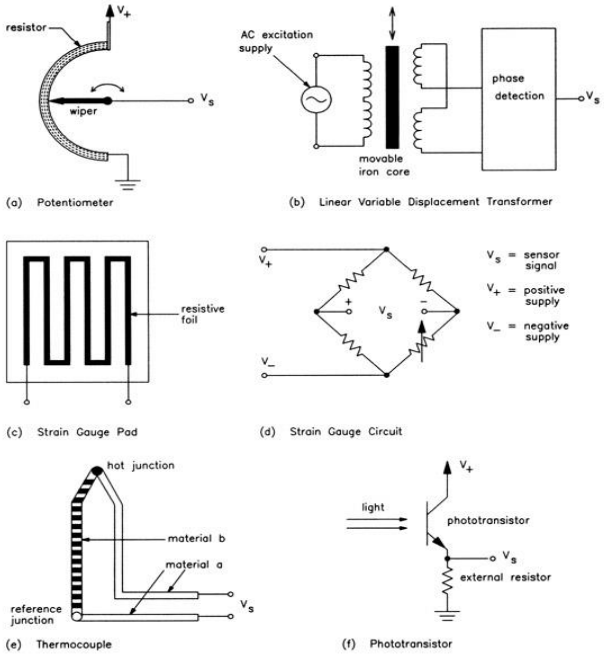
1. Transducers

A device that converts a process variable (ex. Car speed) into an electrical signal or vice versa.

Sensors (Input transducers): Potentiometer (position); strain gauge, piezoelectric device (force); thermistor, thermocouple (temperature); photoconductive cell, phototransistor (light); current transformer, SENSEFET (current); microphone (sound), etc.

Actuators (Output transducers): solenoids, relays, speakers, darlington transistors, triacs, etc.

Some common sensors:



2. Signal Conditioning

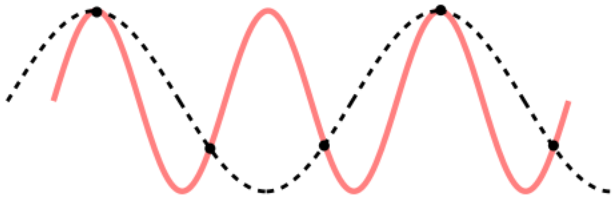
Signal conditioning is sometimes necessary because raw sensor outputs are not always suitable for analog-to-digital conversion.

- Amplification: Op amps are preferred because of their performance characteristics.
- Filtering: Low Pass Filtering is applied to remove noise.

3. Sampling

In order to be able to reconstruct the analog input signal, the sampling rate should be at least twice the maximum frequency component contained in the input signal.

Example of two sine waves have the same sampling values. This is called aliasing.



Antialiasing

- Pre-filtering: use analog hardware to filtering out high-frequency components and only sampling the low-frequency components. The high-frequency components are ignored.
- Post-filtering: Oversample continuous signal, then use software to filter out high-frequency components

4. Basic Definitions

In order to analyze an analog signal, the application must sample the analog signal and reconstruct it in the digital domain. A microprocessor samples an analog signal using an analog to digital converter (ADC). The ADC takes the magnitude of the voltage and converts it to a binary value representing the magnitude. One of the most important characteristics of an ADC is how many bits the ADC uses to represent the magnitude of a signal. The number of bits in a sample is often times referred to as the number of bits of precision. The more bits of precision, the more precisely we can measure an analog signal. To understand this better, let's give some definitions:

- **Offset**: minimum analog value.
- **Span**: maximum analog value – minimum analog value.
    - *Some common spans:*
        - range of 0 V to 5 V: span = 5 V,
        - range of –12 V to 12 V: span = 24 V,
        - range of 4 mA to 20 mA: span = 16 mA.
- **Weight:** the analog change corresponding to a change in a bit in the digital number.
- **Step size:** span / 2^n (Typically, the digital representation is an unsigned *n*- bit integer, where n is the number of bits in the corresponding digital code.)

- **Resolution**: Same as step size. Weight of the LSB.

Example: Let's assume that we have a signal that varies between 3.3V and 0.0V, i.e. Offset=0 V, Span= 3.3V. Let's also assume that we have an ADC which has 3 bits of precision, i.e. n=3. With 3 bits of precision, the ADC can return one of eight unique values. Based on having eight possible values, we can determine the resolution of the ADC to be 3.3V/8 = 0.4125.

If the ADC returned a value of 000 what voltage would this value represent? 000 could represent any value in [0,0.4125). A value of 001 represents any value in [0.4125,0.825).

If we were to increase the number of bits of precision to be 10 bits, the number of possible values returned by the ADC is now 1024 distinct values. The resolution of the ADC is now 3.3/1024 = 0.0032. An ADC value of 00.0000.0000 represents a magnitude of 0.0V to 0.0032V. It should be fairly obvious that increasing the number of bits of precision allows us to more accurately reconstruct a signal.

Example: Analog signal in range +5 to -5 volts, 8- bit ADC:

Offset = – 5 V
Span = 10 V
Step size = 10 / 256 = 39.1 mV
Notice that the MSB has a weight of 5 V (= span/ 2), and the LSB has a weight of .0391 V (the resolution).

5. Equations

Analog number = $(b_{n-1}\ 2^{-1} + b_{n-2}\ 2^{-2} + \ldots + b_0\ 2^{-n})$ x span + offset

= (digital number x step size) + offset

Digital number = (analog number – offset)/ (step size)

= 2^n x (analog number – offset)/span

These equations hold true as long as the numbers are within the range. If the input of an ADC is below the minimum or above the maximum of the range, the corresponding digital value will be minimum or maximum, respectively. In practice, extreme input swing beyond the specified range may damage the device.

Example: 6-bit D/A converter, analog output range: -2.5 to 5 volts. Calculate the analog output when the input is %010101 (decimal 21).

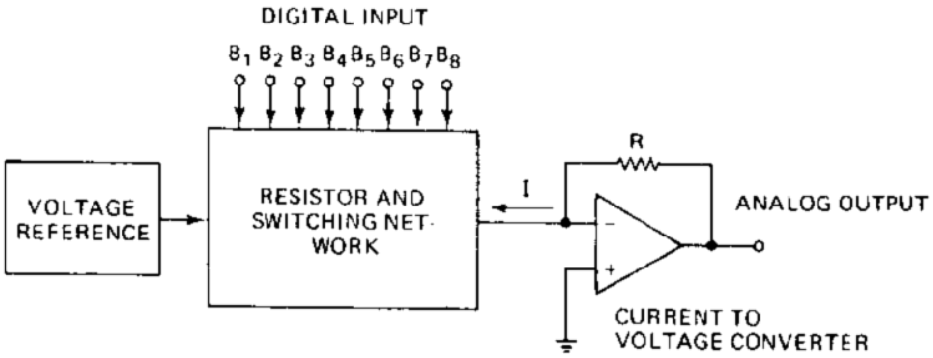
Offset = -2.5 V
Span = 7.5 V
Step size = 0.1172 V
Analog number = 21 x 0.1171 – 2.5 = -0.039 V

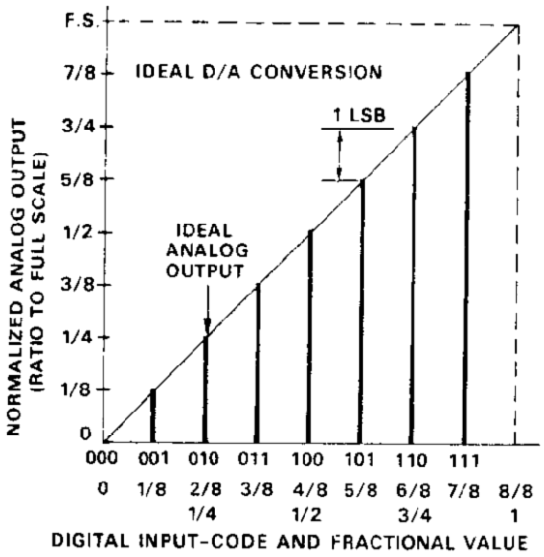
Example: As an example the following table lists some values for -5 V to +5 V analog range to 8-bit digital conversion:

Analog (V)	Digital (hex)
-5	00
-3.75	20
-2.5	40
-1.25	60
0	80
1.25	A0
2.5	C0
3.75	E0
5-0.0391=4.961	FF

6. Digital-to-Analog Converters

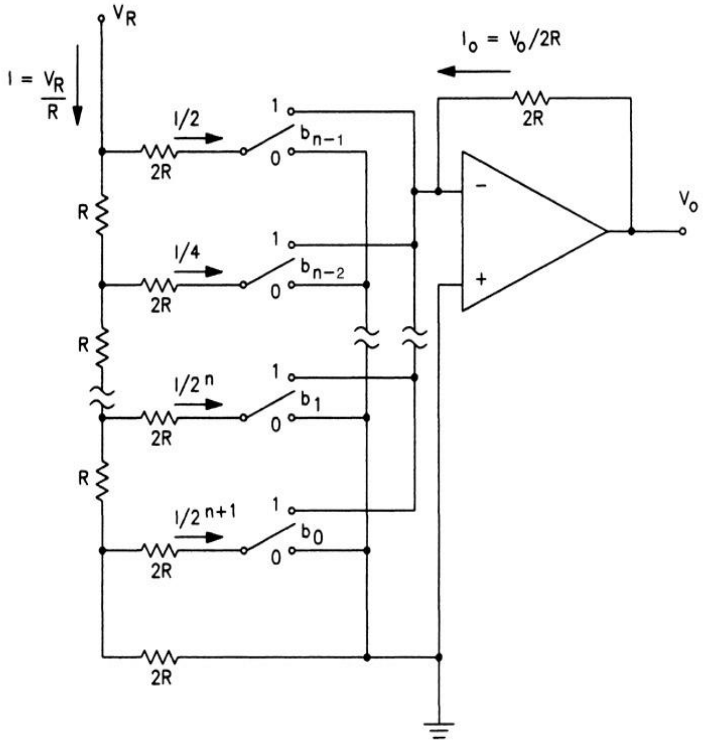
A DAC takes an *n*- bit digital input and outputs a corresponding analog voltage. DAC systems normally consist of three components: (a) A reference voltage, (b) The DAC itself, (c) An op amp for output buffering.





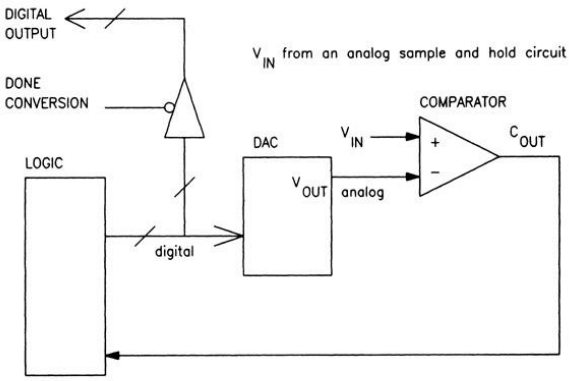
Many digital-to-analog converters use R-2R ladder network. Inverted ladder type of network is shown in the figure below. The switches are analog switches controlled by digital signals. The output voltage (V_o) is proportional to the binary input. Each branch of the ladder network contributes current whose value is proportional to the bit weight of that branch. The amplifier circuit sums the current components to produce a voltage proportional to the binary input.

$$V_R\left[b_{n-1}+b_{n-2}\frac{1}{2}+\ldots+b_0\frac{1}{2^n}\right]=V_O$$

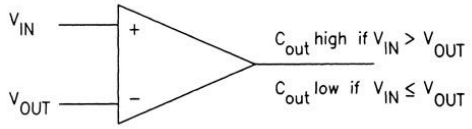


7. Analog-to-Digital Converters

The function of ADCs is to quantize the analog voltage and then output the corresponding digital code value. As with the DAC conversion, a full- scale analog voltage will be divided into 2^n quantization levels or steps for an *n*- bit digital coding scheme. Several techniques are used to do the conversion. Among them successive approximation which has a medium conversion speed is common in computer systems.

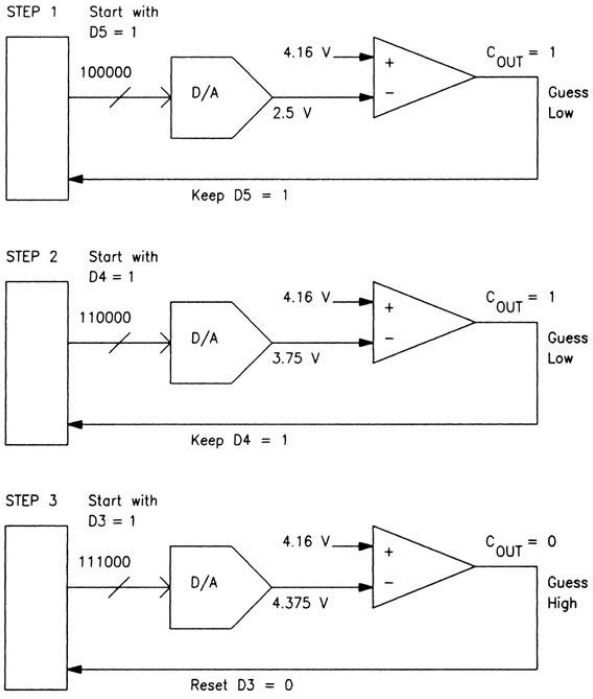


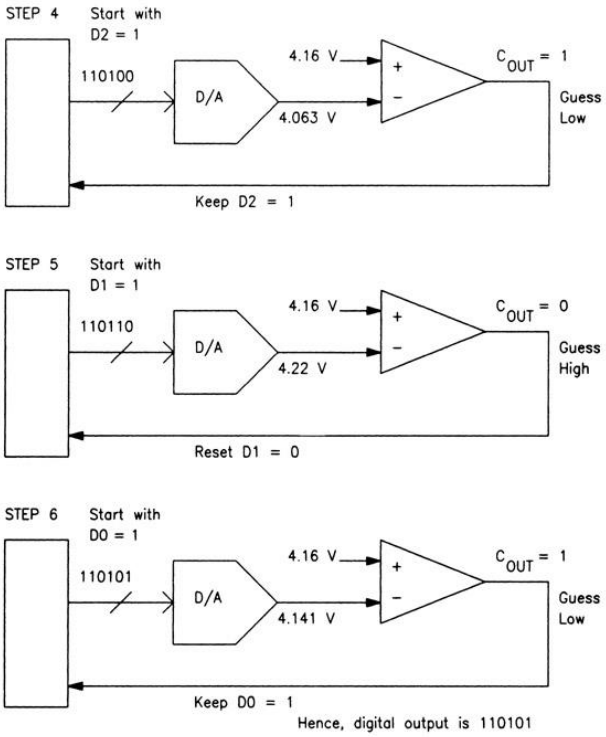
(a) Block Diagram of Successive–Approximation System



(b) Operation of Comparator

Successive Approximation example:

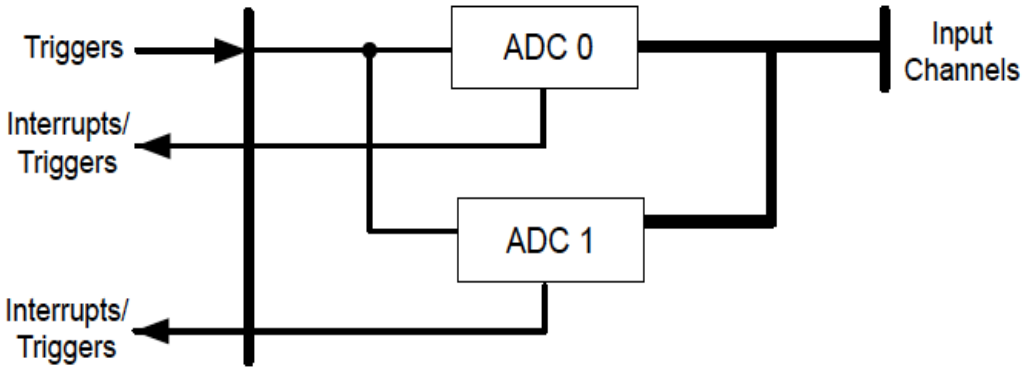




The ADC has a built in DAC. A sample and hold circuit stores an analog input. The ADC logic steps through a sequence of trial-and-error guessing to find the digital equivalent of the input. It begins the sequence by sending a digital signal that is at midrange to the DAC. The analog output V_{out} from the DAC is compared to the analog input V_{in} from the sample-and-hold circuit. Thus, the ADC determines whether the analog input is above, at or below half scale. It continues determining to which half of the next range selection the analog input belongs.

8. TM4C123GH6PM ADC module

The TM4C123GH6PM has two identical analog to digital converters (ADC0 and ADC1). Each converter has 12-bits of precision (is programmable) and can sample at a maximum rate of 1 million samples per second. The ADCs support signals with an amplitude between 0.0V to 3.3V. Resolution = 3.3/4095 which is less than 1mV. The ADCs share 12 analog inputs (AIN0 – AIN11) plus an internal temperature sensor.



The trigger source for ADC0 and ADC1 may be independent or the two ADC modules may operate from the same trigger source and operate on the same or different inputs. Each ADC can be triggered by a number of different events. The ADC can be triggered by software, a timer, an external GPIO pin, etc. A phase shifter can delay the start of sampling by a specified phase angle. When using both ADC modules, it is possible to configure the converters to start the conversions coincidentally or within a relative phase from each other.

Each ADC has four sample sequencers. Each sample sequencer provides flexible programming with fully configurable input source, trigger events, interrupt generation, and sequencer priority. There are 12 analog pins that the ADC can covert to a digital value. Since only one conversion can take place at a time, the sample sequencers allow us to set an arbitrary order in which the conversions take place.

In addition, the conversion value can optionally be diverted to a digital comparator module. Each ADC module provides eight digital comparators. Each digital comparator evaluates the ADC conversion value against its two user-defined values to determine the operational range of the signal.

Initializing the ADC is a two step process. We need to configure the GPIO pins as analog inputs (clearing the corresponding DEN bit in the GPIO Digital Enable (GPIODEN) register and setting the corresponding AMSEL bit in the GPIO Analog Mode Select (GPIOAMSEL) register). After that is done, we need to initialize ADC. Similar to other ports the ADC module clock must be enabled before the registers can be programmed. There must be a delay of 3 system clocks after the ADC module clock is enabled before any ADC module registers are accessed.

Table 8-1. ADC input channel pins and assignments.

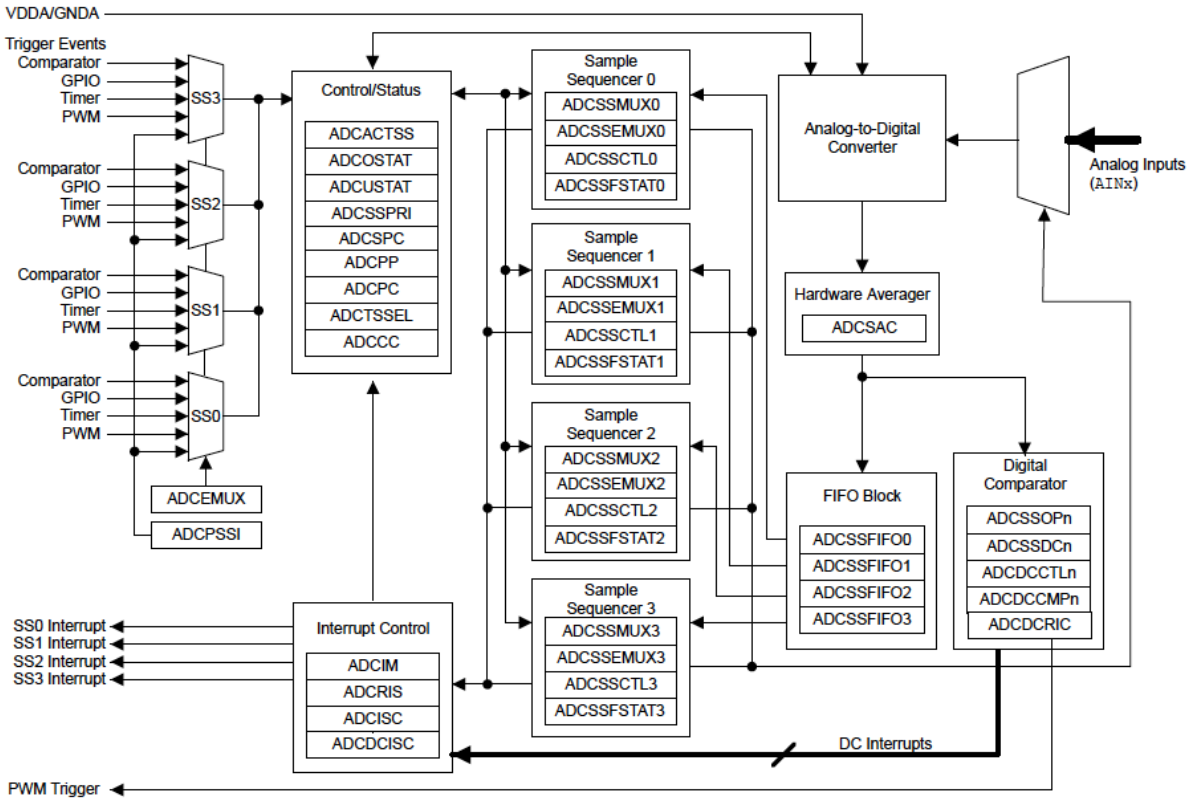
Pin Name	Pin Number	Pin Mux / Pin Assignment	Pin Type	Buffer Type[a]	Description
AIN0	6	PE3	I	Analog	Analog-to-digital converter input 0.
AIN1	7	PE2	I	Analog	Analog-to-digital converter input 1.
AIN2	8	PE1	I	Analog	Analog-to-digital converter input 2.
AIN3	9	PE0	I	Analog	Analog-to-digital converter input 3.
AIN4	64	PD3	I	Analog	Analog-to-digital converter input 4.
AIN5	63	PD2	I	Analog	Analog-to-digital converter input 5.
AIN6	62	PD1	I	Analog	Analog-to-digital converter input 6.
AIN7	61	PD0	I	Analog	Analog-to-digital converter input 7.
AIN8	60	PE5	I	Analog	Analog-to-digital converter input 8.
AIN9	59	PE4	I	Analog	Analog-to-digital converter input 9.
AIN10	58	PB4	I	Analog	Analog-to-digital converter input 10.
AIN11	57	PB5	I	Analog	Analog-to-digital converter input 11.

Base addresses of the ADC modules are:

ADC0: 0x4003.8000

ADC1: 0x4003.9000

8.1. Block Diagram of ADC



See Section 9 for register details.

8.2. Sample Sequencers

The sampling control and data capture is handled by the sample sequencers. All of the sequencers are identical in implementation except for the number of samples that can be captured and the depth of the FIFO. Table 2 shows the maximum number of samples that each sequencer can capture and its corresponding FIFO depth. Each sample that is captured is stored in the FIFO. In this implementation, each FIFO entry is a 32-bit word, with the lower 12 bits containing the conversion result.

Table 8-2 Sequencers.

Sequencer	Number of Samples	Depth of FIFO
SS3	1	1
SS2	4	4
SS1	4	4
SS0	8	8

For a given sample sequence, each sample is defined by bit fields in the ADC Sample Sequence Input Multiplexer Select (ADCSSMUXn) and ADC Sample Sequence Control (ADCSSCTLn) registers, where "n" corresponds to the sequence number. The ADCSSMUXn fields select the input pin, while the ADCSSCTLn fields contain the sample control bits corresponding to parameters such as temperature sensor selection, interrupt enable, end of sequence, and differential input mode.

When configuring a sample sequence, multiple uses of the same input pin within the same sequence are allowed. In the Sample Sequence Control (ADCSSCTLn) register, the IEn bits can be set for any combination of samples, allowing interrupts to be generated after every sample in the sequence if necessary. Also, the END bit can be set at any point within a sample sequence. For example, if Sequencer 0 is used, the END bit can be set in the nibble associated with the fifth sample, allowing Sequencer 0 to complete execution of the sample sequence after the fifth sample.

8.3. TM4C123GH6PM ADC Basic initialization and read

The first step in the initialization is configuring the physical pins to be analog inputs.

System Control Registers(SYSCTL), GPIO Registers (GPIO):

1. Enable desired port in (**RCGCGPIO)**
2. Poll the GPIO Peripheral Ready Register (**PRGPIO**).
3. Configure the pin as an input (**DIR**)
4. Configure the pin as an analog input (**AMSEL**)
5. Configure the pin as an alternate function (**AFSEL**).

System Control Registers(SYSCTL), ADC Registers (ADC0): (Assume we use ADC0 and Sample Sequencer #3 (SS3). SS3 only allows a single analog pin in the sequence.

1. Enable the clock for ADC0 (**RCGCADC**)
2. Wait until ADC0 is ready (**PRADC**)
3. Disable the Sample Sequencers (**ACTSS**)
4. Set the Event Multiplexer Select to processor trigger for SS3 (**EMUX)**
5. Configure Sample Sequencer Control 3. (**SSCTL3)**

Read data through ADC Registers (ADC0)

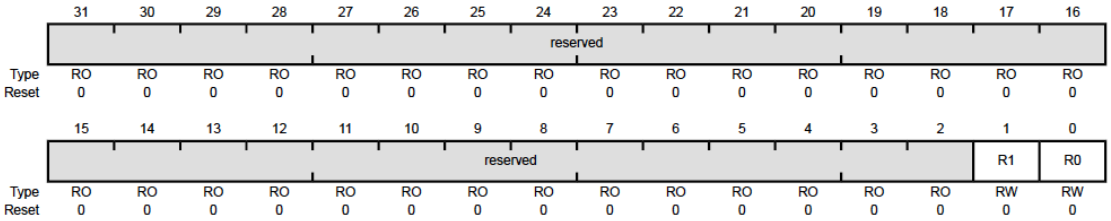
1. Set the Sample Sequencer mux #3 to convert the desired AIN pin (**SSMUX3**)
2. Enable Sample Sequencer #3 (**ACTSS)**
3. Start Sample Sequencer #3 (**PSSI**)
4. Wait for the conversion to complete (**RIS**)
5. Read the results from the Sample Sequencer **#**3′s FIFO (**SSFIFO3**)
6. Acknowledge the conversion (**ISC**)

9. Registers

Registers for Clock Setting

Analog-to-Digital Converter Run Mode Clock Gating Control (RCGCADC)

The RCGCADC register provides software the capability to enable and disable the ADC modules in Run mode. When enabled, a module is provided a clock and accesses to module registers are allowed. When disabled, the clock is disabled to save power and accesses to module registers generate a bus fault.

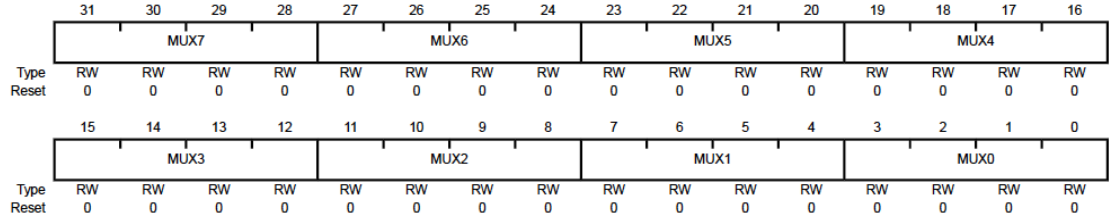


Rn=0 ADC module n is disabled.
Rn=1 Enable and provide a clock to ADC module n in Run mode.

Registers for Setting The Sample Sequencer

ADC Sample Sequence Input Multiplexer Select 0 (ADCSSMUX0), offset 0x040

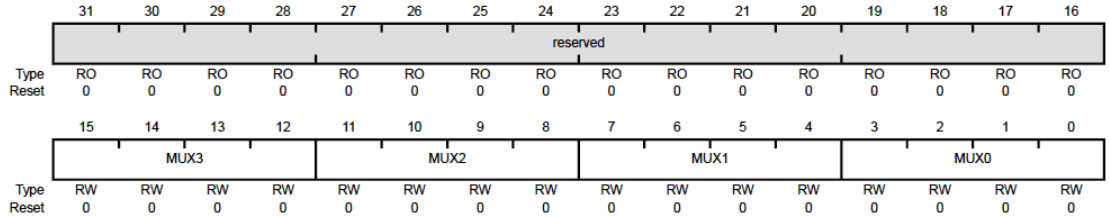
4 bits for each sample specify which input channel (out of 12) is sampled. Number of samples for Sample sequence 0 is eight.



ADC Sample Sequence Input Multiplexer Select 1 (ADCSSMUX1), offset 0x060

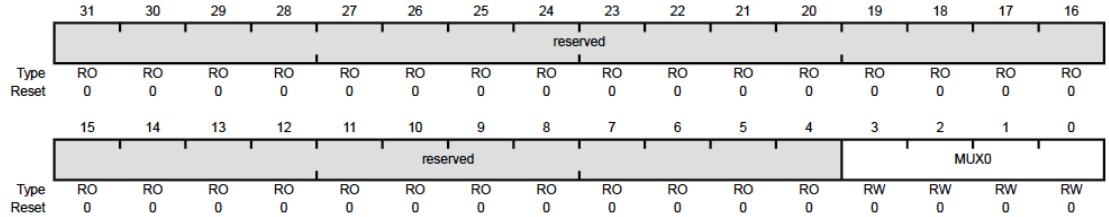
ADC Sample Sequence Input Multiplexer Select 2 (ADCSSMUX2), offset 0x080

Number of samples for Sample sequence 1 and 2 is four.



ADC Sample Sequence Input Multiplexer Select 3 (ADCSSMUX3), offset 0x0A0

Number of samples for Sample sequence 3 is one.



Registers for Configuring The Sample Sequencer

ADC Sample Sequence Control 0 (ADCSSCTL0), offset 0x044

This register contains the configuration information for each sample for a sequence executed with a sample sequencer. This register is 32 bits wide and contains information for eight possible samples of SS0.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	TS7	IE7	END7	D7	TS6	IE6	END6	D6	TS5	IE5	END5	D5	TS4	IE4	END4	D4
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	TS3	IE3	END3	D3	TS2	IE2	END2	D2	TS1	IE1	END1	D1	TS0	IE0	END0	D0
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ADC Sample Sequence Control 1 (ADCSSCTL1), offset 0x064

ADC Sample Sequence Control 2 (ADCSSCTL2), offset 0x084

16 bits for 4 samples of SS1 and SS2.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	TS3	IE3	END3	D3	TS2	IE2	END2	D2	TS1	IE1	END1	D1	TS0	IE0	END0	D0
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ADC Sample Sequence Control 3 (ADCSSCTL3), offset 0x0A4

Four bits for a sample of SS3.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved												TS0	IE0	END0	D0
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

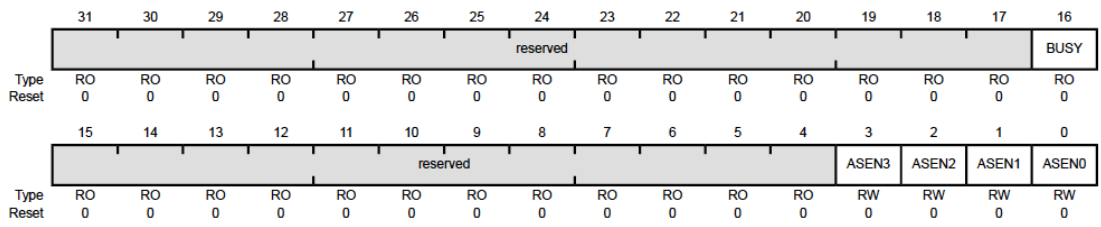
3	TS0	RW	0	1st Sample Temp Sensor Select Value Description 0 The input pin specified by the **ADCSSMUXn** register is read during the first sample of the sample sequence. 1 The temperature sensor is read during the first sample of the sample sequence.
2	IE0	RW	0	1st Sample Interrupt Enable Value Description 0 The raw interrupt is not asserted to the interrupt controller. 1 The raw interrupt signal (INR0 bit) is asserted at the end of the first sample's conversion. If the MASK0 bit in the **ADCIM** register is set, the interrupt is promoted to the interrupt controller. It is legal to have multiple samples within a sequence generate interrupts.
1	END0	RW	0	1st Sample is End of Sequence Value Description 0 Another sample in the sequence is the final sample. 1 The first sample is the last sample of the sequence. It is possible to end the sequence on any sample position. Software must set an ENDn bit somewhere within the sequence. Samples defined after the sample containing a set ENDn bit are not requested for conversion even though the fields may be non-zero.
0	D0	RW	0	1st Sample Differential Input Select Value Description 0 The analog inputs are not differentially sampled. 1 The analog input is differentially sampled. The corresponding **ADCSSMUXn** nibble must be set to the pair number "i", where the paired inputs are "2i and 2i+1". Because the temperature sensor does not have a differential option, this bit must not be set when the TS0 bit is set.

Registers for Enabling The Sample Sequencer And Initiating Sampling

Sample sequencers are enabled by setting the respective ASENn bit in the ADC Active Sample Sequencer (ADCACTSS) register and should be configured before being enabled. Sampling is then initiated by setting the SSn bit in the ADC Processor Sample Sequence Initiate (ADCPSSI) register. In addition, sample sequences may be initiated on multiple ADC modules simultaneously using the GSYNC and SYNCWAIT bits in the ADCPSSI register during the configuration of each ADC module.

ADC Active Sample Sequencer (ADCACTSS), offset 0x000

This register controls the activation of the sample sequencers. Each sample sequencer can be enabled or disabled independently.



ASENn=0, sample sequencer n is disabled,

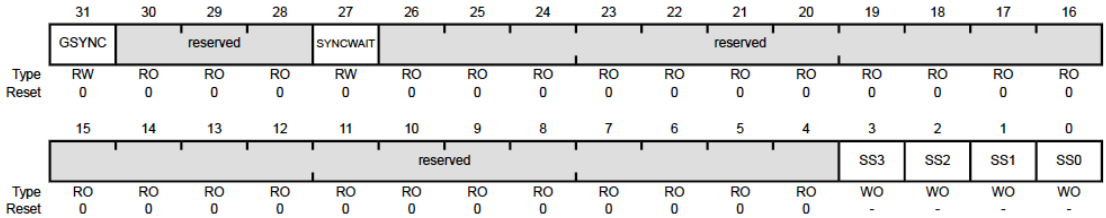
ASENn=1, sample sequencer n is enabled.
BUSY=0, ADC is idle,
BUSY=1, ADC is busy.

The BUSY is used to indicate when the ADC is busy with a current conversion. When there are no triggers pending which may start a new conversion in the immediate cycle or next few cycles, the BUSY bit reads as 0. Software must read the status of the BUSY bit as clear before disabling the ADC clock by writing to the Analog-to-Digital Converter Run Mode Clock Gating Control (RCGCADC) register.

ADC Processor Sample Sequence Initiate (ADCPSSI), offset 0x028

This register provides a mechanism for application software to initiate sampling in the sample sequencers. Sample sequences can be initiated individually or in any combination. When multiple sequences are triggered simultaneously, the priority encodings in ADCSSPRI dictate execution order.

This register also provides a means to configure and then initiate concurrent sampling on all ADC modules. To do this, the first ADC module should be configured. The ADCPSSI register for that module should then be written. The appropriate SS bits should be set along with the SYNCWAIT bit. Additional ADC modules should then be configured following the same procedure. Once the final ADC module is configured, its ADCPSSI register should be written with the appropriate SS bits set along with the GSYNC bit. All of the ADC modules then begin concurrent sampling according to their configuration.



	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	GSYNC	reserved			SYNCWAIT	reserved										
Type	RW	RO	RO	RO	RW	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved												SS3	SS2	SS1	SS0
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	WO	WO	WO	WO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	-	-	-	-

GSYNC bit is cleared once sampling has been initiated.
GSYNC=1 This bit initiates sampling in multiple ADC modules at the same time. Any ADC module that has been initialized by setting an SSn bit and the SYNCWAIT bit starts sampling once this bit is written.
SYNCWAIT=0 Sampling begins when a sample sequence has been initiated.
SYNCWAIT=1 This bit allows the sample sequences to be initiated, but delays sampling until the GSYNC bit is set.
SSn=0 No effect.
SSn=1 Begin sampling on Sample Sequencer n, if the sequencer is enabled in the ADCACTSS register.

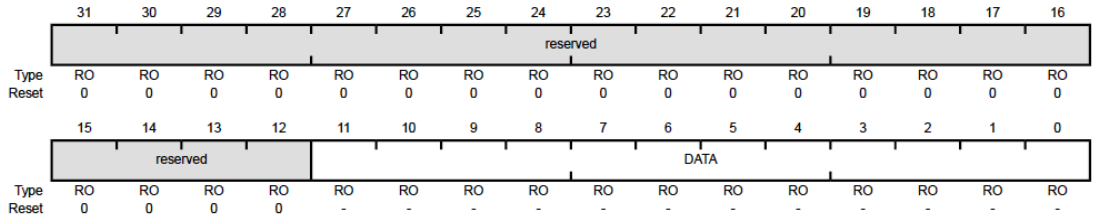
Registers for Reading The Results

After a sample sequence completes execution, the result data can be retrieved from the ADC Sample Sequence Result FIFO (ADCSSFIFOn) registers. The FIFOs are simple circular buffers that read a single address to "pop" result data. For software debug purposes, the positions of the FIFO head and tail pointers are visible in the ADC Sample Sequence FIFO Status (ADCSSFSTATn) registers along with FULL and EMPTY status flags. If a write is

attempted when the FIFO is full, the write does not occur and an overflow condition is indicated. Overflow and underflow conditions are monitored using the ADCOSTAT and ADCUSTAT registers.

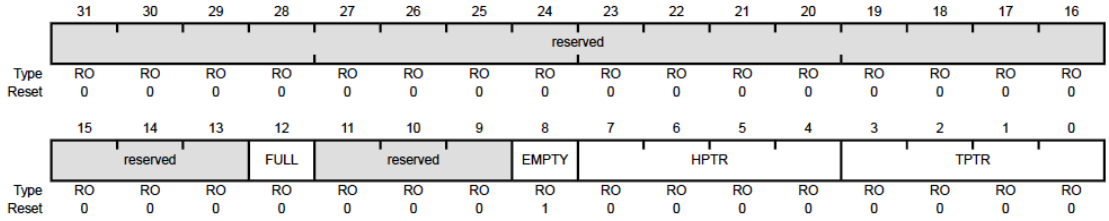
ADC Sample Sequence Result FIFO 0 (ADCSSFIFO0), offset 0x048

ADC Sample Sequence Result FIFO 1 (ADCSSFIFO1), offset 0x068
ADC Sample Sequence Result FIFO 2 (ADCSSFIFO2), offset 0x088
ADC Sample Sequence Result FIFO 3 (ADCSSFIFO3), offset 0x0A8



ADC Sample Sequence FIFO 0 Status (ADCSSFSTAT0), offset 0x04C

ADC Sample Sequence FIFO 1 Status (ADCSSFSTAT1), offset 0x06C
ADC Sample Sequence FIFO 2 Status (ADCSSFSTAT2), offset 0x08C
ADC Sample Sequence FIFO 3 Status (ADCSSFSTAT3), offset 0x0AC

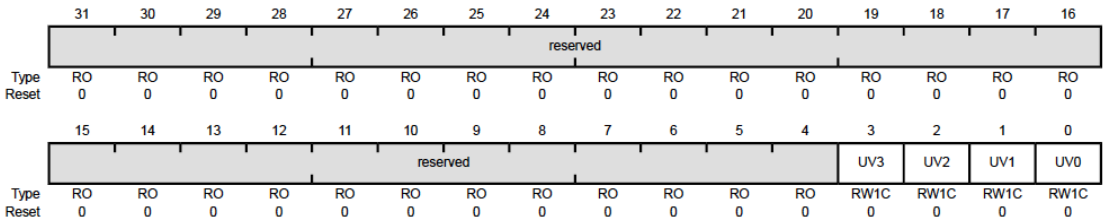


FULL=0 The FIFO is not currently full.
FULL=1 The FIFO is currently full.

EMPTY=0 The FIFO is not currently empty.
EMPTY=1 The FIFO is currently empty.

HPTR / TPTR: the current "head" / "tail" pointer index for the FIFO, that is, the next entry to be written. Valid values are 0x0-0x7 for FIFO0; 0x0-0x3 for FIFO1 and FIFO2; and 0x0 for FIFO3.

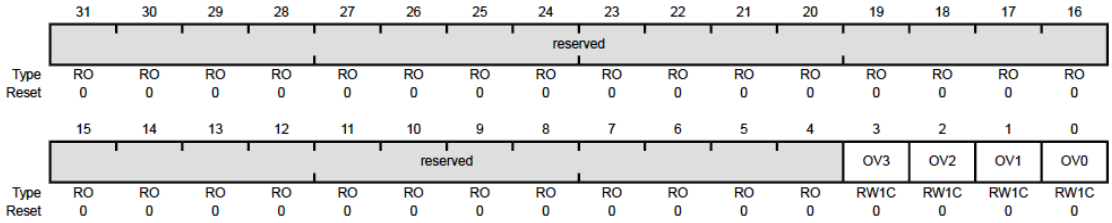
ADC Underflow Status (ADCUSTAT), offset 0x018



UVn=0 The FIFO has not underflowed.
UVn=1 The FIFO for the Sample Sequencer has hit an underflow condition, meaning that the

FIFO is empty and a read was requested. The problematic read does not move the FIFO pointers, and 0s are returned.
This bit is cleared by writing a 1.

ADC Overflow Status (ADCOSTAT), offset 0x010



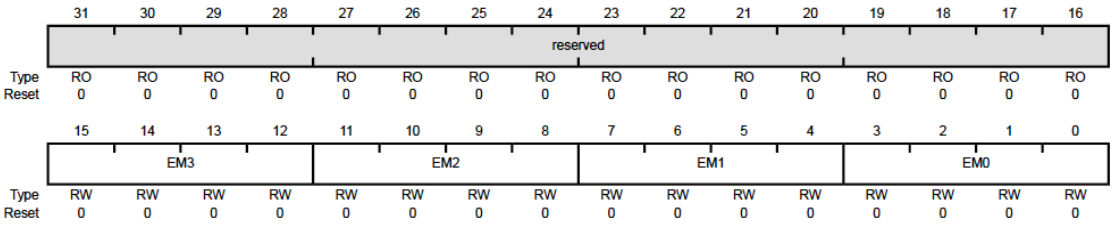
OVn=0 The FIFO has not overflowed.
OVn=1 The FIFO for Sample Sequencer n has hit an overflow condition, meaning that the FIFO is full and a write was requested. When an overflow is detected, the most recent write is dropped.
This bit is cleared by writing a 1.

Registers for Sample Triggering

Sample triggering for each sample sequencer is defined in the ADC Event Multiplexer Select (ADCEMUX) register. Trigger sources include processor (default), analog comparators, an external signal on a GPIO specified by the GPIO ADC Control (GPIOADCCTL) register, a GP Timer, a PWM generator, and continuous sampling. The processor triggers sampling by setting the SSx bits in the ADC Processor Sample Sequence Initiate (ADCPSSI) register.

Care must be taken when using the continuous sampling trigger. If a sequencer's priority is too high, it is possible to starve other lower priority sequencers. Generally, a sample sequencer using continuous sampling should be set to the lowest priority.

ADC Event Multiplexer Select (ADCEMUX), offset 0x014



EMn=0x0 Processor (default)

EMn=0x1 Analog Comparator 0. This trigger is configured by the Analog Comparator Control0 (ACCTL0) register.

EMn=0x2 Analog Comparator 1. This trigger is configured by the Analog Comparator Control1 (ACCTL1) register.

EMn=0x4 External (GPIO Pins) This trigger is connected to the GPIO interrupt for the corresponding GPIO.

EMn=0x5 Timer. In addition, the trigger must be enabled with the TnOTE bit in the GPTMCTL register.

EMn=0x6 PWM generator 0. The PWM generator 0 trigger can be configured with the PWM0 Interrupt and Trigger Enable (PWM0INTEN) register.

EMn=0x7 PWM generator 1. The PWM generator 1 trigger can be configured with the PWM1INTEN register.

EMn=0x8 PWM generator 2. The PWM generator 2 trigger can be configured with the PWM2INTEN register.

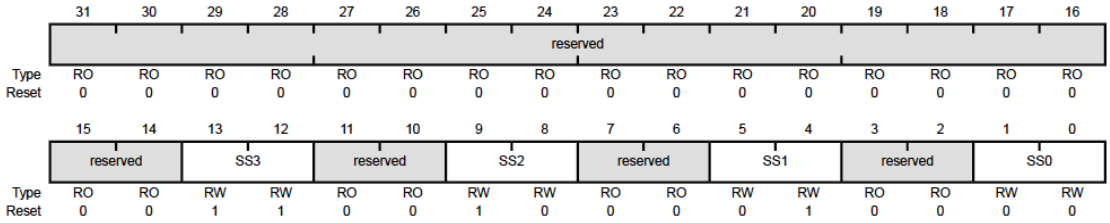
EMn=0x9 PWM generator 3. The PWM generator 3 trigger can be configured with the PWM3INTEN register.

EMn=0xF Always (continuously sample)

Register for Prioritization

When sampling events (triggers) happen concurrently, they are prioritized for processing by the values in the ADC Sample Sequencer Priority (ADCSSPRI) register. Valid priority values are in the range of 0-3, with 0 being the highest priority and 3 being the lowest.

ADC Sample Sequencer Priority (ADCSSPRI), offset 0x020



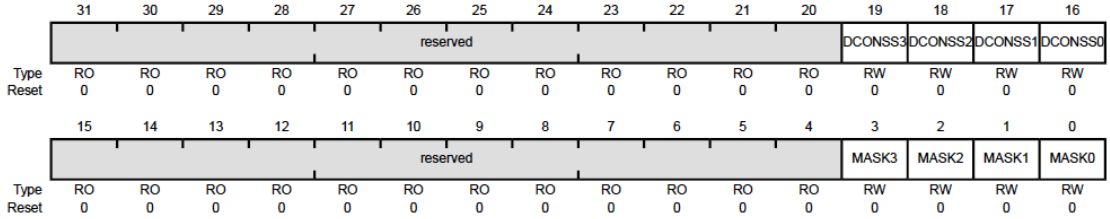
Registers for Interrupt Setting and Control

The register configurations of the sample sequencers and digital comparators dictate which events generate raw interrupts, but do not have control over whether the interrupt is actually sent to the interrupt controller. The ADC module's interrupt signals are controlled by the state of the MASK bits in the ADC Interrupt Mask (ADCIM) register.

Interrupt status can be viewed at two locations: the ADC Raw Interrupt Status (ADCRIS) register, which shows the raw status of the various interrupt signals; and the ADC Interrupt Status and Clear (ADCISC) register, which shows active interrupts that are enabled by the ADCIM register. Sequencer interrupts are cleared by writing a 1 to the corresponding IN bit in ADCISC. Digital comparator interrupts are cleared by writing a 1 to the ADC Digital Comparator Interrupt Status and Clear (ADCDCISC) register.

ADC Interrupt Mask (ADCIM), offset 0x008

This register controls whether the sample sequencer and digital comparator raw interrupt signals are sent to the interrupt controller. Each raw interrupt signal can be masked independently.



DCONSSn=0 The status of the digital comparators does not affect the SSn interrupt status.

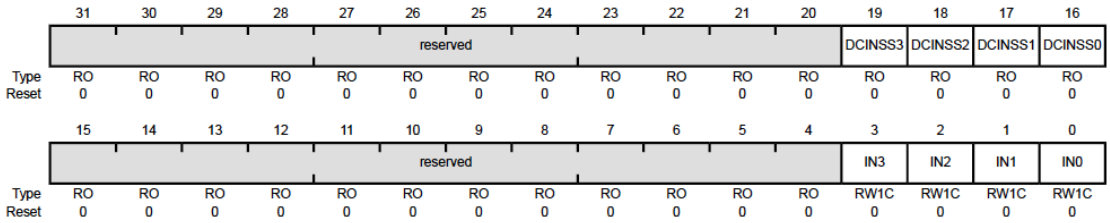
DCONSSn=1 The raw interrupt signal from the digital comparators (INRDC bit in the ADCRIS register) is sent to the interrupt controller on the SSn interrupt line.

MASKn=0 The status of Sample Sequencer n does not affect the SSn interrupt status.

MASKn=1 The raw interrupt signal from Sample Sequencer 3 (ADCRIS register INRn bit) is sent to the interrupt controller.

ADC Interrupt Status and Clear (ADCISC), offset 0x00C

This register provides the mechanism for clearing sample sequencer interrupt conditions and shows the status of interrupts generated by the sample sequencers and the digital comparators which have been sent to the interrupt controller. When read, each bit field is the logical AND of the respective INR and MASK bits. Sample sequencer interrupts are cleared by writing a 1 to the corresponding bit position. Digital comparator interrupts are cleared by writing a 1 to the appropriate bits in the ADCDCISC register. If software is polling the ADCRIS instead of generating interrupts, the sample sequence INRn bits are still cleared via the ADCISC register, even if the INn bit is not set.



DCINSS3=0 No interrupt has occurred or the interrupt is masked. Both the INRDC bit in the ADCRIS register and the DCONSSn bit in the ADCIM register are set, providing a level-based interrupt to the interrupt controller.

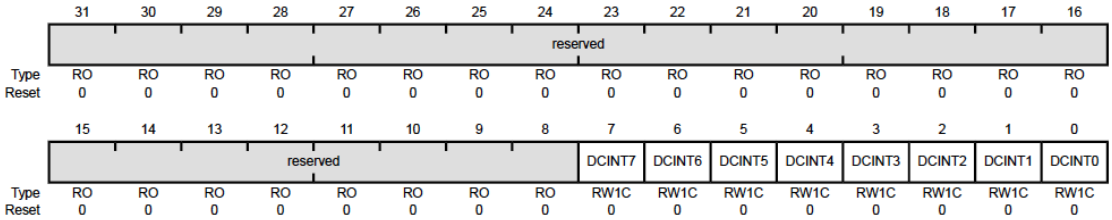
DCINSS3=1 This bit is cleared by writing a 1 to it. Clearing this bit also clears the INRDC bit in the ADCRIS register.

INn=0 No interrupt has occurred or the interrupt is masked. Both the INRn bit in the ADCRIS register and the MASKn bit in the ADCIM register are set, providing a level-based interrupt to the interrupt controller.

INn=1 This bit is cleared by writing a 1. Clearing this bit also clears the INRn bit in the ADCRIS register.

ADC Digital Comparator Interrupt Status and Clear (ADCDCISC), offset 0x034

This register provides status and acknowledgement of digital comparator interrupts. One bit is provided for each comparator.

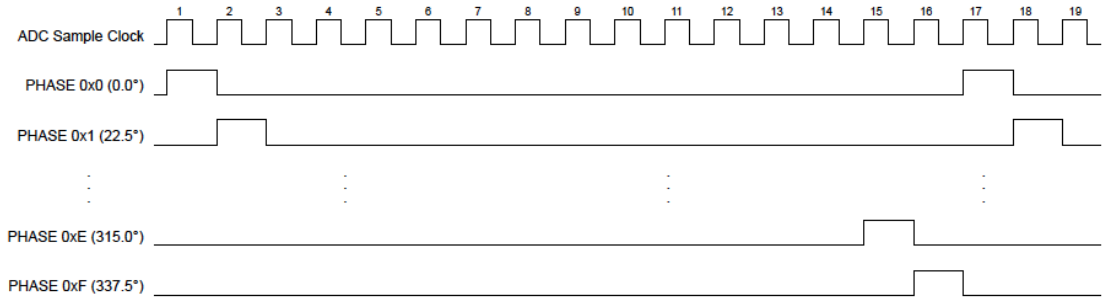


DCINTn=0 No interrupt.

DCINTn=1 Digital Comparator n has generated an interrupt. This bit is cleared by writing a 1.

10. SAMPLE PHASE CONTROL

The trigger source for ADC0 and ADC1 may be independent or the two ADC modules may operate from the same trigger source and operate on the same or different inputs. If the converters are running at the same sample rate, they may be configured to start the conversions coincidentally or with one of 15 different discrete phases relative to each other. The sample time can be delayed from the standard sampling time in 22.5° increments up to 337.5º using the ADC Sample Phase Control (ADCSPC) register.



This feature can be used to double the sampling rate of an input. Both ADC module 0 and ADC module 1 can be programmed to sample the same input. ADC module 0 could sample at the standard position (the PHASE field in the ADCSPC register is 0x0). ADC module 1 can be configured to sample at 180 (PHASE = 0x8). The two modules can be synchronized using the GSYNC and SYNCWAIT bits in the ADC Processor Sample Sequence Initiate (ADCPSSI) register.

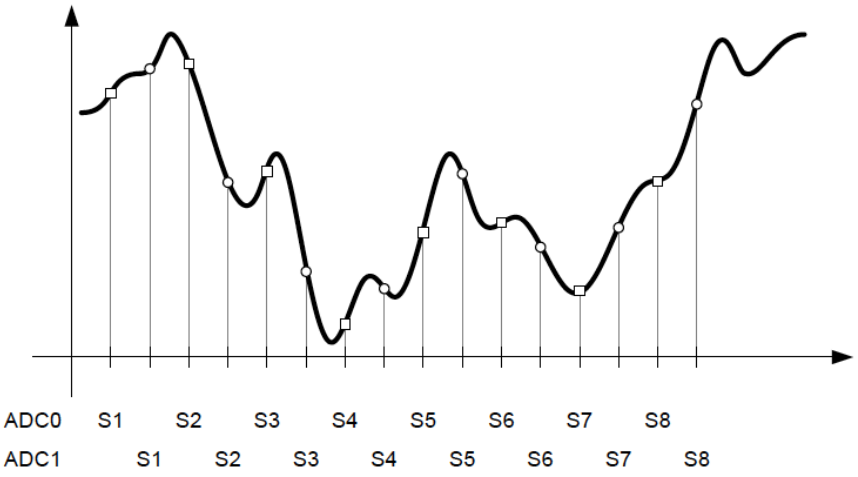


Figure 10-1 ADC0 and ADC1 are used together to increase the sampling rate.

11. Module Clocking

11.1. ADC Clock

The module is clocked by a 16-MHz clock, which can be sourced by a divided version of the PLL (phase lock loop) output, the PIOSC (internal precision Oscillator) or an external source connected to MOSC (the main oscillator).

When the PLL is operating, the ADC clock is derived from the PLL ÷ 25 by default.

To use the PIOSC to clock the ADC, first power up the PLL and then enable the PIOSC in the CS bit field in the ADC Clock Configuration (ADCCC) register, then disable the PLL.

The module clock source attached to the MOSC must be 16 MHz unless the PIOSC is used for the clock source. To use the MOSC to clock the ADC, first power up the PLL and then enable the clock to the ADC module, then disable the PLL and switch to the MOSC for the system clock.

The ADC module can continue to operate in Deep-Sleep mode if the PIOSC is the ADC module clock source.

The system clock must be at the same frequency or higher than the ADC clock.

All ADC modules share the same clock source to facilitate the synchronization of data samples between conversion units, the selection and programming of which is provided by ADC0's ADCCC register. The ADC modules do not run at different conversion rates.

ADC Clock Configuration (ADCCC), offset 0xFC8

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved												CS			
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

CS=0x0 Either the 16-MHz system clock (if the PLL bypass is in effect) or the 16 MHz clock derived from PLL ÷ 25 (default). Note that when the PLL is bypassed, the system clock must be at least 16 MHz.

CS=0x1 PIOSC The PIOSC provides a 16-MHz clock source for the ADC. If the PIOSC is used as the clock source, the ADC module can continue to operate in Deep-Sleep mode.

11.2. System Clock

There are multiple clock sources for use in the microcontroller:

Precision Internal Oscillator (PIOSC): The precision internal oscillator is an on-chip clock source. It does not require the use of any external components and provides a 16-MHz clock with ±1% accuracy with calibration and ±3% accuracy across temperature.

Main Oscillator (MOSC): The main oscillator provides a frequency-accurate clock source by one of two means: an external single-ended clock source is connected to the OSC0 input pin, or an external crystal is connected across the OSC0 input and OSC1 output pins. If the PLL is being used, the crystal value must be one of the supported frequencies between 5 MHz to 25 MHz (inclusive). If the PLL is not being used, the crystal may be any one of the supported frequencies between 4 MHz to 25 MHz.

Low-Frequency Internal Oscillator (LFIOSC): The low-frequency internal oscillator is intended for use during Deep-Sleep power-saving modes. The frequency can have wide variations.

Hibernation Module Clock Source: The Hibernation module is clocked by a 32.768-kHz oscillator connected to the XOSC0 pin. The 32.768-kHz oscillator can be used for the system clock, thus eliminating the need for an additional crystal or oscillator. The Hibernation module clock source is intended to provide the system with a real-time clock source and may also provide an accurate source of Deep-Sleep or Hibernate mode power savings.

The internal system clock (SysClk), is derived from any of the above sources plus two others:

- the output of the main internal PLL,
- the precision internal oscillator divided by four (4 MHz ± 1%).

The frequency of the PLL clock reference must be in the range of 5 MHz to 25 MHz (inclusive).

Various clock sources can be used in a system is summarized below:

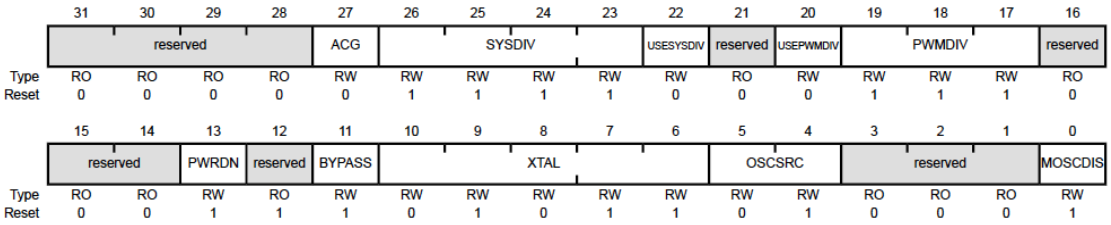
Clock Source	Drive PLL?		Used as SysClk?	
Precision Internal Oscillator	Yes	BYPASS = 0, OSCSRC = 0x1	Yes	BYPASS = 1, OSCSRC = 0x1
Precision Internal Oscillator divide by 4 (4 MHz ± 1%)	No	-	Yes	BYPASS = 1, OSCSRC = 0x2
Main Oscillator	Yes	BYPASS = 0, OSCSRC = 0x0	Yes	BYPASS = 1, OSCSRC = 0x0
Low-Frequency Internal Oscillator (LFIOSC)	No	-	Yes	BYPASS = 1, OSCSRC = 0x3
Hibernation Module 32.768-kHz Oscillator	No	-	Yes	BYPASS = 1, OSCSRC2 = 0x7

The Run-Mode Clock Configuration (RCC) and Run-Mode Clock Configuration 2 (RCC2) registers provide control for the system clock. RCC2 provides for a larger assortment of clock configuration options.

These registers control the following clock functionality:

- Source of clocks in sleep and deep-sleep modes
- System clock derived from PLL or other clock source
- Enabling/disabling of oscillators and PLL
- Clock divisors

Run-Mode Clock Configuration (RCC), offset 0x060



ACG: Auto Clock Gating This bit specifies whether the system uses the Sleep-Mode Clock Gating Control (SCGCn) registers and Deep-Sleep-Mode Clock Gating Control (DCGCn) registers if the microcontroller enters a Sleep or Deep-Sleep mode (respectively).

SYSDIV: System Clock Divisor Specifies which divisor is used to generate the system clock from either the PLL output or the oscillator source (depending on how the BYPASS bit in this register is configured).

USESYSDIV Enable System Clock Divider

USEPWMDIV Enable PWM Clock Divisor

PWMDIV PWM Unit Clock Divisor

PWRDN PLL Power Down

BYPASS PLL Bypass

XTAL Crystal Value This field specifies the crystal value attached to the main oscillator. The encoding for this field is provided below.

0x06	4 MHz	reserved
0x07	4.096 MHz	reserved
0x08	4.9152 MHz	reserved
0x09	5 MHz (USB)	
0x0A	5.12 MHz	
0x0B	6 MHz (USB)	
0x0C	6.144 MHz	
0x0D	7.3728 MHz	
0x0E	8 MHz (USB)	
0x0F	8.192 MHz	
0x10	10.0 MHz (USB)	
0x11	12.0 MHz (USB)	
0x12	12.288 MHz	
0x13	13.56 MHz	
0x14	14.31818 MHz	
0x15	16.0 MHz (USB)	
0x16	16.384 MHz	
0x17	18.0 MHz (USB)	
0x18	20.0 MHz (USB)	
0x19	24.0 MHz (USB)	
0x1A	25.0 MHz (USB)	

OSCSRC Oscillator Source

0x0	MOSC Main oscillator
0x1	PIOSC Precision internal oscillator (default)
0x2	PIOSC/4 Precision internal oscillator / 4
0x3	LFIOSC Low-frequency internal oscillator

MOSCDIS Main Oscillator Disable

12. Extra functionalities

12.1. Dither Enable

The DITHER bit in the ADCCTL register is used to reduce random noise in ADC sampling and keep the ADC operation within the specified performance limits. When taking multiple consecutive samples with the ADC Module, the DITHER bit should be enabled in the ADCCTL register along with hardware averaging in the ADC Sample Averaging Control (ADCSAC) register. The DITHER bit is disabled by default at reset.

ADC Control (ADCCTL), offset 0x038

This register configures the voltage reference. Note that values set in this register apply to all ADC modules, it is not possible to set one module to use internal references and another to use external references.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved									DITHER	reserved					VREF
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RO	RO	RO	RO	RO	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Dither=0/1 Dither mode disabled/enabled.

VREF=0 VDDA and GNDA are the voltage references for all ADC modules.

12.2. Extra Hardware Sample Averaging Circuit

Higher precision results can be generated using the hardware averaging circuit, however, the improved results are at the cost of throughput. For example, if the averaging circuit is configured to average 16 samples, the throughput is decreased by a factor of 16. Up to 64 samples can be accumulated and averaged to form a single data entry in the sequencer FIFO.

By default the averaging circuit is off, and all data from the converter passes through to the sequencer FIFO. The averaging hardware is controlled by the ADC Sample Averaging Control (ADCSAC) register. A single averaging circuit has been implemented, thus all input channels receive the same amount of averaging whether they are single-ended or differential.

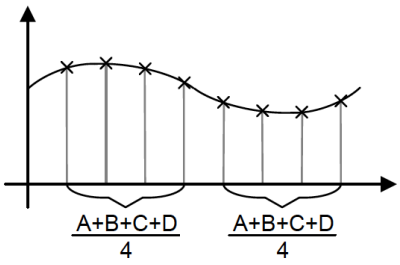


Figure 12-1 ADCSAC register is set to 0x2 for 4x hardware oversampling.

ADC Sample Averaging Control (ADCSAC), offset 0x030

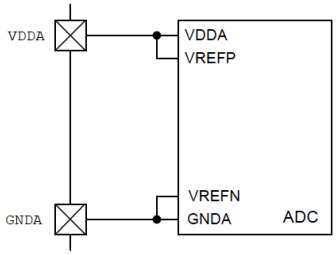
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved													AVG		
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0x0	No hardware oversampling
0x1	2x hardware oversampling
0x2	4x hardware oversampling
0x3	8x hardware oversampling
0x4	16x hardware oversampling
0x5	32x hardware oversampling
0x6	64x hardware oversampling
0x7	reserved

12.3. Voltage Reference

The ADC uses internal signals VREFP and VREFN as references to produce a conversion value from the selected analog input. VREFP is connected to VDDA and VREFN is connected to GNDA.



Parameter	Parameter Name	Min	Nom	Max	Unit
POWER SUPPLY REQUIREMENTS					
V_{DDA}	ADC supply voltage	2.97	3.3	3.63	V
GNDA	ADC ground voltage	-	0	-	V

12.4. Differential Sampling

In addition to traditional single-ended sampling, the ADC module supports differential sampling of two analog input channels. To enable differential sampling, software must set the Dn bit in the ADCSSCTLn register in a step's configuration nibble. When a sequence step is configured for differential sampling, the input pair to sample must be configured in the ADCSSMUXn register.

Differential pairs are listed below:

Differential Pair	Analog Inputs
0	0 and 1
1	2 and 3
2	4 and 5
3	6 and 7
4	8 and 9
5	10 and 11

12.5. Internal Temperature Sensor

The temperature sensor serves two primary purposes:

1) to notify the system that internal temperature is too high or low for reliable operation,

2) to provide temperature measurements for calibration of the Hibernate module RTC trim value.

The temperature sensor does not have a separate enable.

The internal temperature sensor converts a temperature measurement into a voltage. This voltage value, VTSENS, is given by the following equation (where TEMP is the temperature in °C):

VTSENS = 2.7 - ((TEMP + 55) / 75)

The temperature sensor reading can be sampled in a sample sequence by setting the TSn bit in the ADCSSCTLn register. The temperature reading from the temperature sensor can also be given as a function of the ADC value. The following formula calculates temperature (TEMP in ℃) based on the ADC reading (ADCCODE , given as an unsigned decimal number from 0 to 4095) and the maximum ADC voltage range (VREFP - VREFN):

TEMP = 147.5 - ((75 * (VREFP - VREFN) × ADCCODE) / 4096)

13. Initialization Examples

13.1. Module Initialization

Initialization of the ADC module is a simple process with very few steps: enabling the clock to the ADC, disabling the analog isolation circuit associated with all inputs that are to be used, and reconfiguring the sample sequencer priorities (if needed).

1. Enable the ADC clock using the RCGCADC register.
2. Enable the clock to the appropriate GPIO modules via the RCGCGPIO register.
3. Set the GPIO AFSEL bits for the ADC input pins.
4. Configure the AINx signals to be analog inputs by clearing the corresponding DEN bit in the GPIO Digital Enable (GPIODEN) register.
5. Disable the analog isolation circuit for all ADC input pins that are to be used by writing a 1 to the appropriate bits of the GPIOAMSEL register in the associated GPIO block.
6. If required by the application, reconfigure the sample sequencer priorities in the ADCSSPRI register. The default configuration has Sample Sequencer 0 with the highest priority and Sample Sequencer 3 as the lowest priority.

13.2. Sample Sequencer Configuration

Configuration of the sample sequencers is slightly more complex than the module initialization because each sample sequencer is completely programmable.

The configuration for each sample sequencer should be as follows:

1. Ensure that the sample sequencer is disabled by clearing the corresponding ASENn bit in the ADCACTSS register. Programming of the sample sequencers is allowed without having them enabled. Disabling the sequencer during programming prevents erroneous execution if a trigger event were to occur during the configuration process.

2. Configure the trigger event for the sample sequencer in the ADCEMUX register.

3. When using a PWM generator as the trigger source, use the ADC Trigger Source Select (ADCTSSEL) register to specify in which PWM module the generator is located. The default register reset selects PWM module 0 for all generators.

4. For each sample in the sample sequence, configure the corresponding input source in the ADCSSMUXn register.

5. For each sample in the sample sequence, configure the sample control bits in the corresponding nibble in the ADCSSCTLn register. When programming the last nibble, ensure that the END bit is set. Failure to set the END bit causes unpredictable behavior.

6. If interrupts are to be used, set the corresponding MASK bit in the ADCIM register.

7. Enable the sample sequencer logic by setting the corresponding ASENn bit in the ADCACTSS register.

Example 1:

Setup and run ADC sampling on ADC0 and AIN0 = PE3. Store results starting from the address in R5. Use software trigger, sequencer 3. Sampling rate: 125KHz.

- Initialize ADC and GPIO
- Configure ADC
- Set software trigger
    - Write to PSSI bit 3
- Busy-Wait
    - Raw Interrupt Status = RIS bit 3
    - Poll until sample complete
- Read sample
    - Read from SSFIFO3
- Clear sample complete flag
    - Write to ISC bit 3

Initialization and Configuration for ADC

– select rate

Value	Description
0x7	1M samples/second
0x5	500K samples/second
0x3	250K samples/second
0x1	125K samples/second

Speed bits in ADC0_PP_R

– select sequencer (activate at end)

Sequencer	# of Samples	Depth of FIFO
SS3	1	1
SS2	4	4
SS1	4	4
SS0	8	8

Activate with ADC_ACTSS

– select trigger

Value	Event
0x0	Software start
0x1	Analog Comparator 0
0x2	Analog Comparator 1
0x3	Analog Comparator 2
0x4	External (GPIO PB4)
0x5	Timer
0x6	PWM0
0x7	PWM1
0x8	PWM2
0xF	Always (continuously sample)

EM3, EM2, EM1, and EM0 bits in ADC_EMUX_R

– select channel: bits 3-0 in ADC0_SSMUX3 = 0

This field specifies which analog input (in this case AIN0) is sampled

– select sample mode: bits 3-0 in ADC0_SSCTL3 = 6

- TS0=0 (temperature sensor not read)
- IE0=1 (interrupt sent to controller)
- END0=1 (end of sequence)
- D0=0 (not differential)

3	2	1	0
TS0	IE0	END0	D0
RW	RW	RW	RW
0	0	0	0

Disable interrupts: zero bit 3 of ADC0_IM_R
Enable selected sequencer 3: set bit 3 of ADC0_ACTSS_R

```
; ADC Registers
RCGCADC EQU 0x400FE638      ; ADC clock register
                            ; ADC0 base address EQU 0x40038000
ADC0_ACTSS EQU 0x40038000   ; Sample sequencer (ADC0 base address)
ADC0_RIS EQU 0x40038004     ; Interrupt status
ADC0_IM EQU 0x40038008      ; Interrupt select
ADC0_EMUX EQU 0x40038014    ; Trigger select
ADC0_PSSI EQU 0x40038028    ; Initiate sample
ADC0_SSMUX3 EQU 0x400380A0  ; Input channel select
ADC0_SSCTL3 EQU 0x400380A4  ; Sample sequence control
ADC0_SSFIFO3 EQU 0x400380A8 ; Channel 3 results
ADC0_PP EQU 0x40038FC4      ; Sample rate

; GPIO Registers
```

```
RCGCGPIO EQU 0x400FE608      ; GPIO clock register
                             ;PORT E base address EQU 0x40024000
PORTE_DEN EQU 0x4002451C     ; Digital Enable
PORTE_PCTL EQU 0x4002452C    ; Alternate function select
PORTE_AFSEL EQU 0x40024420   ; Enable Alt functions
PORTE_AMSEL EQU 0x40024528   ; Enable analog


; Start clocks for features to be used
LDR R1, =RCGCADC             ; Turn on ADC clock
LDR R0, [R1]
ORR R0, R0, #0x01            ; set bit 0 to enable ADC0 clock
STR R0, [R1]
NOP
NOP
NOP                          ; Let clock stabilize
LDR R1, =RCGCGPIO            ; Turn on GPIO clock
LDR R0, [R1]
ORR R0, R0, #0x10            ; set bit 4 to enable port E clock
STR R0, [R1]
NOP
NOP
NOPs                         ; Let clock stabilize

; Setup GPIO to make PE3 input for ADC0
; Enable alternate functions
LDR R1, =PORTE_AFSEL
LDR R0, [R1]
ORR R0, R0, #0x08            ; set bit 3 to enable alt functions on PE3
STR R0, [R1]
; PCTL does not have to be configured
; since ADC0 is automatically selected when
; port pin is set to analog.
; Disable digital on PE3
LDR R1, =PORTE_DEN
LDR R0, [R1]
BIC R0, R0, #0x08            ; clear bit 3 to disable analog on PE3
STR R0, [R1]
; Enable analog on PE3
LDR R1, =PORTE_AMSEL
LDR R0, [R1]
ORR R0, R0, #0x08            ; set bit 3 to enable analog on PE3
STR R0, [R1]

; Disable sequencer while ADC setup
LDR R1, =ADC0_ACTSS
LDR R0, [R1]
BIC R0, R0, #0x08            ; clear bit 3 to disable seq 3
STR R0, [R1]
```

```
; Select trigger source
LDR R1, =ADC0_EMUX
LDR R0, [R1]
BIC R0, R0, #0xF000                ; clear bits 15:12 to select SOFTWARE
STR R0, [R1] ; trigger
; Select input channel
LDR R1, =ADC0_SSMUX3
LDR R0, [R1]
BIC R0, R0, #0x000F                ; clear bits 3:0 to select AIN0
STR R0, [R1]
; Config sample sequence
LDR R1, =ADC0_SSCTL3
LDR R0, [R1]
ORR R0, R0, #0x06          ; set bits 2:1 (IE0, END0)
STR R0, [R1]
; Set sample rate
LDR R1, =ADC0_PP
LDR R0, [R1]
ORR R0, R0, #0x01          ; set bits 3:0 to 1 for 125k sps
STR R0, [R1]
; Done with setup, enable sequencer
LDR R1, =ADC0_ACTSS
LDR R0, [R1]
ORR R0, R0, #0x08          ; set bit 3 to enable seq 3
STR R0, [R1]               ; sampling enabled but not initiated yet


        ; start sampling routine
        LDR R3, =ADC0_RIS        ; interrupt address
        LDR R4, =ADC0_SSFIFO3    ; result address
        LDR R2, =ADC0_PSSI       ; sample sequence initiate address
        LDR R6,= ADC0_ISC

        ; initiate sampling by enabling sequencer 3 in ADC0_PSSI
Smpl    LDR R0, [R2]
        ORR R0, R0, #0x08        ; set bit 3 for SS3
        STR R0, [R2]
        ; check for sample complete (bit 3 of ADC0_RIS set)
Cont    LDR R0, [R3]
        ANDS R0, R0, #8
        BEQ Cont
        ;branch fails if the flag is set so data can be read and flag is cleared
        LDR R0,[R4]
        STR R0,[R5],#4           ;store the data
        MOV R0, #8
        STR R0, [R6]             ; clear flag
        B Smpl
```

Example 2:

You are required to design a TM4C123 based system, which continuously monitors four analog voltages v0 to v3 (all within 0-3V range) and displays the index of the channel with the smallest analog voltage at a single digit common–anode seven-segment display.

a) Clearly state to which port (and pin) of TM4C123 the display and the analog voltages are connected. (You can use a hex-to-seven-segment-decoder.)

b) Write a complete program using Thumb2 assembly language with labels to perform the required task.

```
MAIN
        B INIT_GPIO
        ; PD 3,2,1,0 for A/D (AIN4,5,6,7)
        ; PB 0,1,2,3 for display
        B INIT_ADC
        ; Seq 1

; start sampling routine
        LDR R3, =ADC0_RIS       ; interrupt address
        LDR R4, =ADC0_SSFIFO1   ; result address
        LDR R2, =ADC0_PSSI      ; sample sequence initiate address
        LDR R5,= ADC0_ISC
        LDR R10,= 0x4000503C    ; addressing PortB pins 3:0
; initiate sampling by enabling sequencer 1 in ADC0_PSSI
Smpl    LDR R0, [R2]
        ORR R0, R0, #0x02       ; set bit 2 for SS1
        ; cleared once sampling is initiated
        STR R0, [R2]
; check for sample complete (bit 2 of ADC0_RIS set)
Cont    LDR R0, [R3]
        AND R0, R0, #2
        BEQ Cont
;branch fails if the flag is set so data can be read and flag is cleared
        MOV R7,#0xFFFFFFFF
        MOV R8,#0X00
C_AD    LDR R0,[R4]
        CMP R0,R7
        BHI CHG
        MOV R7,R0
        MOV R9,R8
CHG     ADDS R8,R8,#1
        CMP R8,#4
        BNE C_AD

        MOV R0, #2
        STR R0, [R5]            ; clear flag
        STR R9, [R10]           ; display
```

```
        B Smpl


INIT_GPIO

; Start clocks for features to be used
        LDR R1, =RCGCGPIO               ; Turn on GPIO clock
        LDR R0, [R1]
        ORR R0, R0, #0x0A               ; enable port B and D clock
        STR R0, [R1]
        NOP
        NOP
        NOP                             ; Let clock stabilize

; Setup GPIO to make PD0-3 input for ADC0
; Enable alternate functions
        LDR R1, =PORTD_AFSEL
        LDR R0, [R1]
        ORR R0, R0, #0x0F
        STR R0, [R1]
; Disable digital on PD0-3
        LDR R1, =PORTD_DEN
        LDR R0, [R1]
        BIC R0, R0, #0x0F
        STR R0, [R1]
; Enable analog on PD0-3
        LDR R1, =PORTD_AMSEL
        LDR R0, [R1]
        ORR R0, R0, #0x0F
        STR R0, [R1]

; Setup GPIO to make PB0-3 output for display
; Disable alternate functions
        LDR R1, =PORTB_AFSEL
        LDR R0, [R1]
        BIC R0, R0, #0x0F
        STR R0, [R1]
; Enable digital on PB0-3
        LDR R1, =PORTB_DEN
        LDR R0, [R1]
        ORR R0, R0, #0x0F
        STR R0, [R1]
; Disable analog on PB0-3
        LDR R1, =PORTB_AMSEL
        LDR R0, [R1]
        BIC R0, R0, #0x0F
        STR R0, [R1]
; PB0-3 direction
```

```
        LDR R1, =PORTB_DIR
        LDR R0, [R1]
        ORR R0, R0, #0x0F
        STR R0, [R1]
        BX LR

INIT_ADC
        LDR R1, =RCGCADC            ; Turn on ADC clock
        LDR R0, [R1]
        ORR R0, R0, #0x01           ; set bit 0 to enable ADC0 clock
        STR R0, [R1]
        NOP
        NOP
        NOP                         ; Let clock stabilize
; Disable sequencer while ADC setup
        LDR R1, =ADC0_ACTSS
        LDR R0, [R1]
        BIC R0, R0, #0x02           ; clear to disable seq 1
        STR R0, [R1]
; Select trigger source
        LDR R1, =ADC0_EMUX
        LDR R0, [R1]
        BIC R0, R0, #0x00F0         ; clear bits 4:7 to select processor trigger
        STR R0, [R1]
; Select input channel
        LDR R1, =ADC0_SSMUX1
        MOV R0,=0x4567
        STR R0,[R1]
; Config sample sequence
        LDR R1, =ADC0_SSCTL1
        MOV R0,#0x2000              ; set bit 1 for sample 4 (END0)
        STR R0, [R1]
; Done with setup, enable sequencer
        LDR R1, =ADC0_ACTSS
        LDR R0, [R1]
        ORR R0, R0, #0x02           ; set to enable seq'r 3
        STR R0, [R1]                ; sampling enabled but not initiated yet
        BX LR
```

Chapter 9

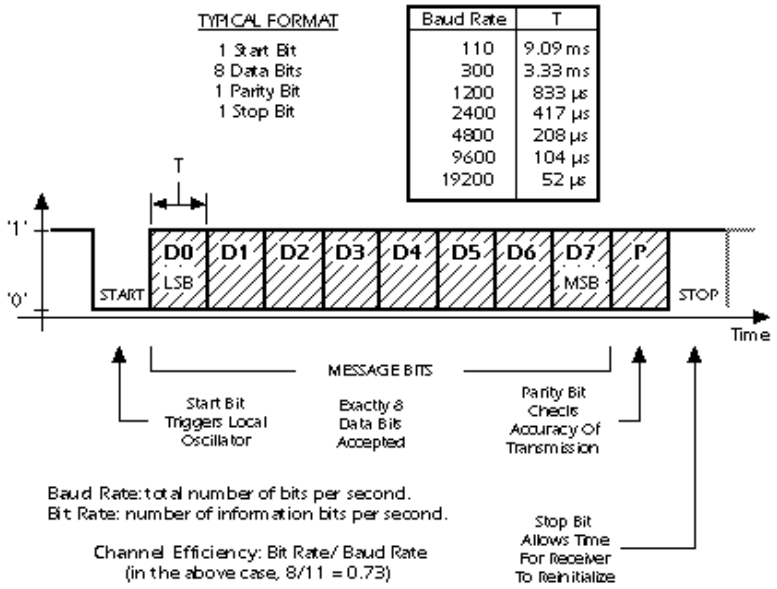
Serial I/O

Serial transmission involves sending one bit at a time, such that the data is spread out over time. Compared to parallel communication, many fewer lines are required to transmit data. This requires fewer pins but adds complexity. Serialized data is not generally sent at a uniform rate through a channel. Instead, there is usually a burst of regularly spaced binary data bits followed by a pause, after which the data flow resumes. Packets of binary data are sent in this manner, possibly with variable-length pauses between packets, until the message has been fully transmitted. In order for the receiving end to know the proper moment to read individual binary bits from the channel, it must know exactly when a packet begins and how much time elapses between bits. When this timing information is known, the receiver is said to be synchronized with the transmitter, and accurate data transfer becomes possible. Failure to remain synchronized throughout a transmission will cause data to be corrupted or lost.

Two basic techniques are employed to ensure correct synchronization: Synchronous and asynchronous sytems.

In synchronous systems, separate channels are used to transmit data and timing information. The timing channel transmits clock pulses to the receiver. Upon receipt of a clock pulse, the receiver reads the data channel and latches the bit value found on the channel at that moment. The data channel is not read again until the next clock pulse arrives. Because the transmitter originates both the data and the timing pulses, the receiver will read the data channel only when told to do so by the transmitter (via the clock pulse), and synchronization is guaranteed.

In asynchronous systems, a separate timing channel is not used. The transmitter and receiver must be preset in advance to an agreed-upon baud rate. A very accurate local oscillator within the receiver will then generate an internal clock signal that is equal to the transmitter's speed within a fraction of a percent. For the most common serial protocol, data is sent in small packets of 10 or 11 bits, eight of which constitute message information. When the channel is idle, the signal voltage corresponds to a continuous logic '1'. A data packet always begins with a logic '0' (the start bit) to signal the receiver that a transmission is starting. The start bit triggers an internal timer in the receiver that generates the needed clock pulses. Following the start bit, eight bits of message data are sent bit by bit at the agreed upon baud rate. The packet is concluded with a parity bit and stop bit(s). One complete packet is illustrated below:



Start and Stop Framing

In asynchronous serial communications, the basic unit (group of bits) of communication is the *character* or *data frame*. The transmitter can send frames at any rate and whenever it feels like it. Hence there is a need for the receiver to know when a frame starts and ends.

Parity

It is used to detect single bit errors in communication. The transmitter generates the parity and the receiver checks it. Parity bit is either a 0 or 1 depending on:

1. the type of parity selected, even or odd
2. the number of 1 bits in the data byte to be transferred

Example: Assume that the transmitter wants to send the seven bit ASCII code 0111000 with even parity. Then the transmitted bits (shown in order of transmission from left to right) are as follows:

0	0001110	1	1
Start Bit	Data bits	Parity bit	Stop bit

The number of one bits in the frame, excluding the start and stop bits but including the parity, is even (even parity convention).

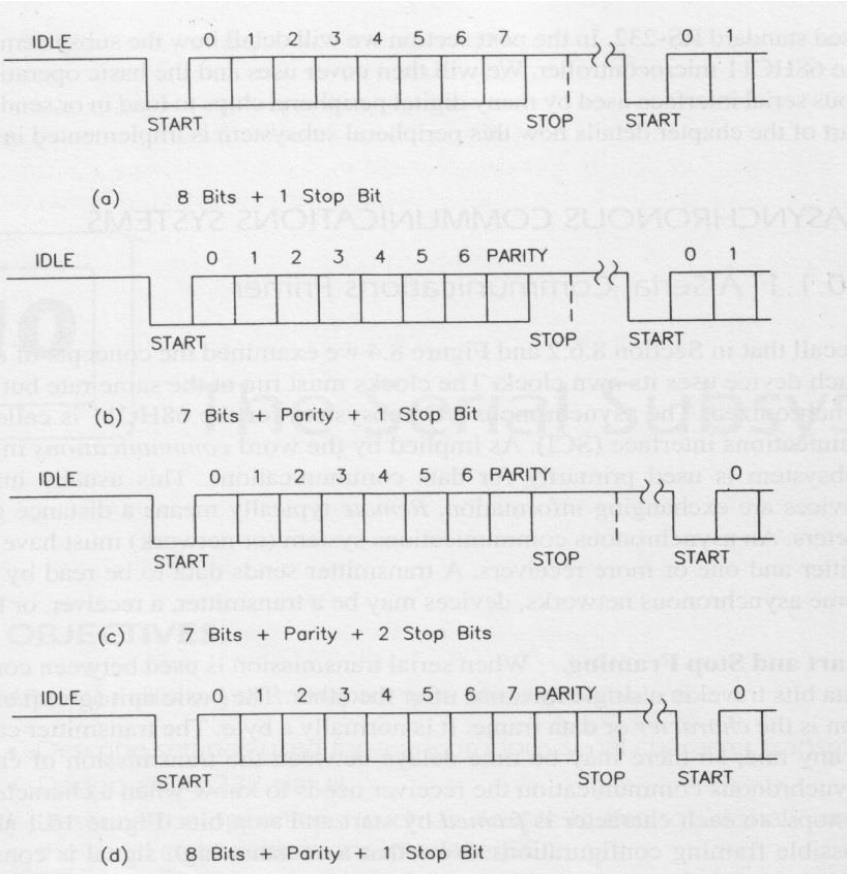


Figure 10.1 Asynchronous Serial Transmission and Framing

Data Speed and Baud Rate

Serial data speed is the number of bits transmitted per second (bps). Baud is the rate at which the signal changes. If each signal element represents more than one bit, then the data speed and the baud rate are not equal to each other. If each signal element represents one bit, then the data speed and baud rate are the same.

In asynchronous serial communications terminology, for example, 9600 baud means that serial data (including the start and stop bits) is transmitted and received at the rate of 9600 bits per second. Since there are overheads in the frame

$$\text{“Actual data transfer speed”} = \frac{\text{number of data bits}}{\text{number of frame bits}} \times \text{baud rate}$$

Flow and error control

A protocol is a set of rules for making connections and transferring messages. Flow control refers to method for stopping data transfers ad restarting them. When transferring data, there have to be aggreed upon rules by both the transmitter and receiver.

ACK/NAK Flow Control: The transmitter waits for the receiver to acknowledge each block of data before the next block can be sent. Upon reception of a block of data, the receiver checks it and send an ACK signal (or message) otherwise NAK is sent.

Cyclic Redundancy Check (CRC): Not a single parity but a longer check field is used for error detection and correction.

XON/XOFF Flow Control: A receiver might initiate or stop a transmitter by sending XON and XOFF signals (messages).

BREAK Signal: used to abort a transfer in midstream.

Communication Channel Types

There are three ways to transmit data along a communication channel:

1. simplex (unidirectional)
2. half-duplex (bidirectional but only direction active at a given time)
3. full-duplex (bidirectional and both directions active at a given time)

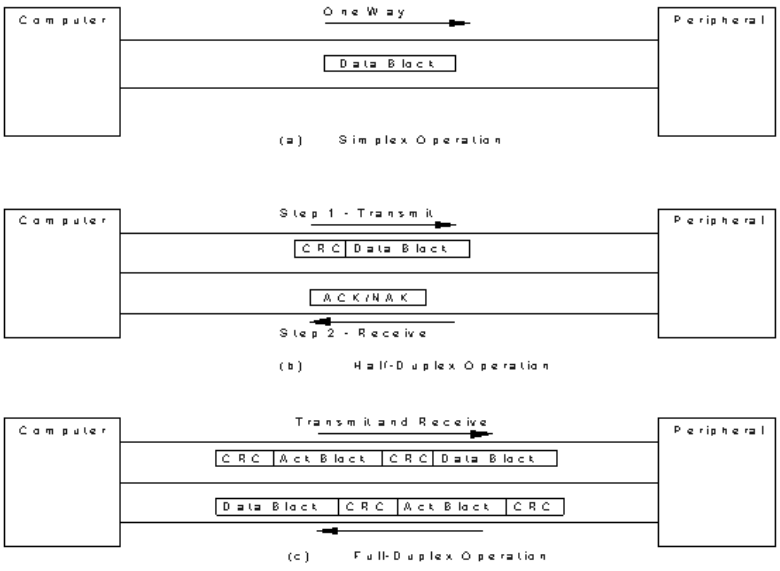


Figure 10.3 Modes of Channel Operation

1. Protocols

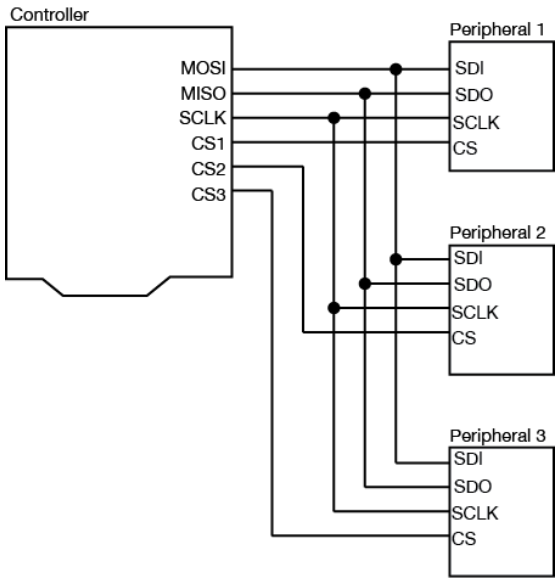
Several protocols exist for both synchronous and asynchronous communication types (UART, SSI, SPI, I2C).

UART: Stands for Universal Asynchronous Receive/Transmit. In simple terms UART relies on a baud rate agreement between receiving and transmitting controllers. The receiving MCU will expect to receive bits into its receive (RX) port at specific baud rate, for example 9600, and the transmitting MCU will transmit bits at the same baud rate, 9600. There is no master clock between the two parties, only the agreement in baud rate. Therefore the

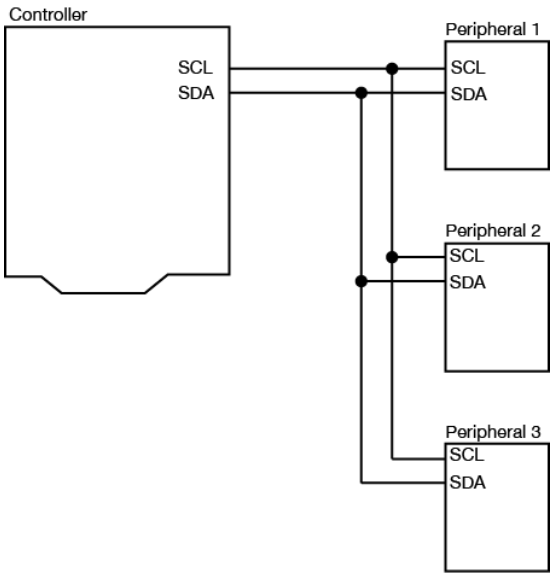
receiving MCU will know the rate that bits will flow into its RX port, and know when it has a full symbol.

SSI: Stands for Synchronous Serial interface. SSI is a synchronous, point to point, serial communication channel for digital data transmission. Synchronous data transmission is one in which the data is transmitted by synchronizing the transmission at the receiving and sending ends using a common clock signal. Since start and stop bits are not present, this allows better use of data transmission bandwidth for more message bits and makes the whole transmission process simpler and easier. There are two common forms of synchronous serial, Inter-Integrated Circuit, or I2C (sometimes also called Two-Wire Interface, or TWI), and Serial Peripheral Interface, or SPI.

SPI: Serial Peripheral Interface involves a master and a slave, or multiple slaves. SPI interfacing involves 3 or more wires, consisting of a clock, serial data out, serial data in, and chip select if necessary. The master MCU basically sets the clock rate for the slaves, asks a specific one to listen up using the chip select port, and sends them commands via its serial data out port, and expects to receive the output from the slave through the serial data in port.



I^2C: Stands for inter-integrated circuit, where every chip in the circuit has a defined address which they can be called up from. Two wires are involved, SDA and CLK. SDA is a bi-directional data flow meaning data can be accepted and sent out based on conditions set. The master sets the clock for communication, a start condition is sent by the master via SDA which tells all the slaves to listen for their address. The master then sends the address of the chip from which interfacing is desired (typically with MSB being a read or write command) and waits for an acknowledgement. After acknowledgement is received the read or write command is sent, followed by some end of transmission.



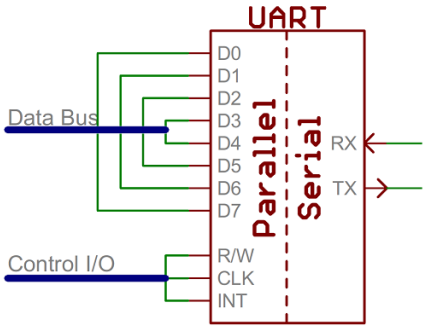
To choose which one is appropriate for your application, the followings can be considered:

- **Speed :** A measure of the number of bits transferred per-second (bps). Some protocols have heavy overhead which means that there can be a significant difference between gross, raw, and net figures; protocol overhead grows proportionately with the amount of data that needs to be sent. Consider sending a temperature measurement between two microcontrollers once per-second vs sending a real-time video stream between two Internet-connected devices.
- **Difficulty:** A fact / product of hooking up the physical wires, writing code, pin counts, and managing actual communications. Expect a tradeoff of speed and device support with variance in difficulty.
- **Devices:** Not an exact figure. More of a general feel for the number of devices you can expect to see that support the specified protocol out-of-the-box.

	Speed	Difficulty	Devices
SPI	☆☆☆	☆☆	☆☆☆
I^2C	☆☆	☆☆☆	☆☆
Serial / UART	☆	☆	☆

2. UART

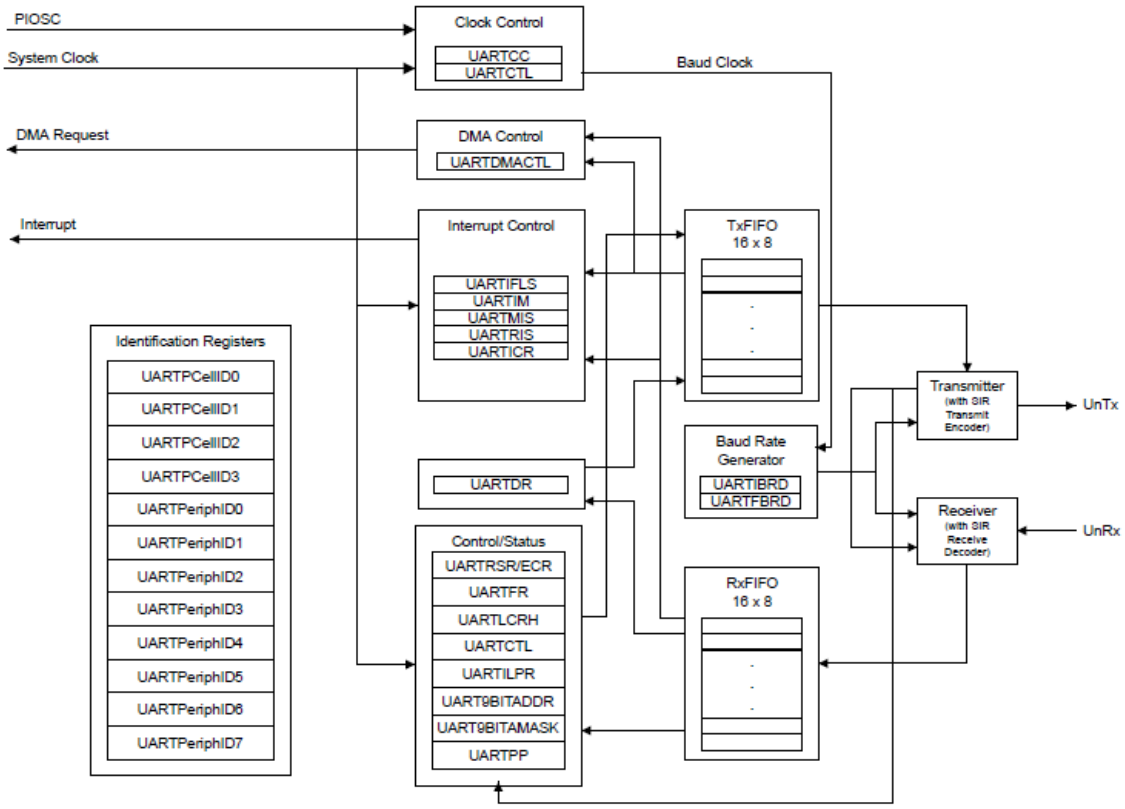
A universal asynchronous receiver/transmitter (UART) is a block of circuitry responsible for implementing serial communication. Essentially, the UART acts as an intermediary between parallel and serial interfaces. On one end of the UART is a bus of eight-or-so data lines (plus some control pins), on the other is the two serial wires - RX and TX.



2.1. TM4C123 UART Features

The TM4C123 controller includes eight Universal AsynchronousReceiver/Transmitter (UART) with the following features:

- Programmable baud-rate generator allowing speeds up to 5 Mbps for regular speed (divide by 16) and 10 Mbps for high speed (divide by 8)
- Separate 16x8 transmit (TX) and receive (RX) FIFOs to reduce CPU interrupt service loading
- Programmable FIFO length, including 1-byte deep operation providing conventional double-buffered interface
- FIFO trigger levels of 1/8, 1/4, 1/2, 3/4, and 7/8
- Standard asynchronous communication bits for start, stop, and parity
- Line-break generation and detection
- Fully programmable serial interface characteristics
    - 5, 6, 7, or 8 data bits
    - Even, odd, stick, or no-parity bit generation/detection
    - 1 or 2 stop bit generation
- Standard FIFO-level and End-of-Transmission interrupts



2.2. UART pin assignment

The GPIO Alternate Function Select (GPIOAFSEL) register should be set to choose the UART function and GPIO Port Control (GPIOPCTL) register should be programmed for the following pins as given in the table.

Pin Name	Pin Number	Pin Mux / Pin Assignment	Pin Type	Buffer Type[a]	Description
U0Rx	17	PA0 (1)	I	TTL	UART module 0 receive.
U0Tx	18	PA1 (1)	O	TTL	UART module 0 transmit.
U1CTS	15 29	PC5 (8) PF1 (1)	I	TTL	UART module 1 Clear To Send modem flow control input signal.
U1RTS	16 28	PC4 (8) PF0 (1)	O	TTL	UART module 1 Request to Send modem flow control output line.
U1Rx	16 45	PC4 (2) PB0 (1)	I	TTL	UART module 1 receive.
U1Tx	15 46	PC5 (2) PB1 (1)	O	TTL	UART module 1 transmit.
U2Rx	53	PD6 (1)	I	TTL	UART module 2 receive.
U2Tx	10	PD7 (1)	O	TTL	UART module 2 transmit.
U3Rx	14	PC6 (1)	I	TTL	UART module 3 receive.
U3Tx	13	PC7 (1)	O	TTL	UART module 3 transmit.
U4Rx	16	PC4 (1)	I	TTL	UART module 4 receive.
U4Tx	15	PC5 (1)	O	TTL	UART module 4 transmit.
U5Rx	59	PE4 (1)	I	TTL	UART module 5 receive.
U5Tx	60	PE5 (1)	O	TTL	UART module 5 transmit.
U6Rx	43	PD4 (1)	I	TTL	UART module 6 receive.
U6Tx	44	PD5 (1)	O	TTL	UART module 6 transmit.
U7Rx	9	PE0 (1)	I	TTL	UART module 7 receive.
U7Tx	8	PE1 (1)	O	TTL	UART module 7 transmit.

2.3. UART Baud Generation

The baud rate divisor is a 22 bit number consisting of a 16 bit integer (UART Integer Baud Rate Divisor - UARTIBRD) and a 6 bit fractional part (UART Fractional Baud Rate Divisor – UARTFBRD). The bit period is determined by these values by the baud rate generator.

The Baud16 clock is created from the system bus clock, with a frequency of (Bus clock frequency)/divider. The baud rate is 16 times slower than Baud16

Baud rate = Baud16/16 = (Bus clock frequency)/(16*divider)

For example, if the bus clock is 80 MHz and the desired baud rate is 19200 bits/sec, then the divider should be 80,000,000/16/19200 or 260.4167. Let m be the integer part, without rounding. We store the integer part (m=260) in IBRD. For the fraction, we find an integer n, such that n/64 is about 0.4167. More simply, we multiply 0.4167*64 = 26.6688 and round to the closest integer, 27. We store this fraction part (n=27) in FBRD. We did approximate the divider, so it is interesting to determine the actual baud rate. Assume the bus clock is 80 MHz.

Baud rate = (80 MHz)/(16* (m+n/64)) = (80 MHz)/(16* (260+27/64)) = 19199.616 bits/sec

The baud rates in the transmitter and receiver must match within 5% for the channel to operate properly. The error for this example is 0.002%.

The three registers LCRH, IBRD, and FBRD form an internal 30-bit register. This internal register is only updated when a write operation to LCRH is performed, so any changes to the baud-rate divisor must be followed by a write to the LCRH register for the changes to take effect. Out of reset, both FIFOs are disabled and act as 1-byte-deep holding registers. The FIFOs are enabled by setting the FEN bit in LCRH.

2.4. UART Data Transmission

The UART is configured for transmit and/or receive via the UART Control (UARTCTL) register.

Data transmitted is stored in 16 byte FIFO. (FIFO level is also configurable. When level=1, it is conventional double buffering). The transmit logic performs parallel to serial conversion on the data read from the transmit FIFO using the 10-bit shift register. The transmitter software will first check to make sure the transmit FIFO is not full (it will wait if TXFF is 1) and then write to the transmit data register (UART0_DR_R). When this data is transferred to Tx FIFO, the TXEF bit in the UART Flag (UARTFR) register becomes 0 meaning that FIFO has data to transmit. If the shift register is empty this data is shifted to the shift register and TXEF bit becomes 1 again if there is no more data in it. The bits are shifted out in this order: start, b0, b1, b2, b3, b4, b5, b6, b7, and then stop, where b0 is the LSB and b7 is the MSB. The transmit data register is write only, which means the software can write to it (to start a new transmission) but cannot read from it. Even though the transmit data register is at the same address as the receive data register, the transmit and receive data registers are two separate registers. The transmission software can write to its data register if its TXFF (transmit FIFO full) flag is zero. TXFF equal to zero means the FIFO is not full and has room.

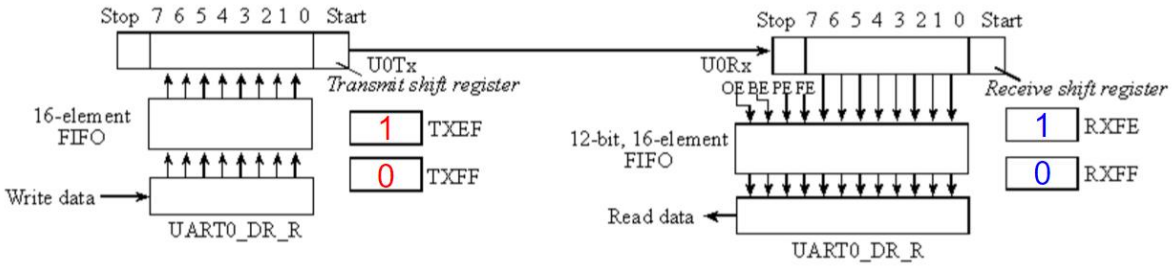
When enabled, the control logic outputs the serial bit stream beginning with a start bit and followed by the data bits (LSB first), parity bit, and the stop bits according to the configuration done in UART Line Control (UARTLCRH) register. Data is transmitted until there is no data left in the transmit FIFO. The BUSY bit in the UART Flag (UARTFR) register is also asserted as soon as data is written to the transmit FIFO. It is negated when transmission FIFO is empty and the last character has been transmitted from the shift register, including the stop bits.

The receiver portion of the UART includes a U0Rx data input pin with digital logic levels. At the input of the microcontroller, true is 3.3V and false is 0V. There is also a 16-element FIFO and a 10-bit shift register, which cannot be directly accessed by the programmer. The receive shift register is 10 bits wide, but the FIFO is 12 bits, 8 bits of data and 4 error flags. Again the receive shift register and receive FIFO are separate from those in the transmitter. The receive data register, UART0_DR_R, is read only, which means write operations to this address have no effect on this register. The receiver obviously cannot start a transmission, but it recognizes a new frame by its start bit. The bits are shifted in using the same order as the transmitter shifted them out: start, b0, b1, b2, b3, b4, b5,b6, b7, and then stop.

There are six status bits generated by receiver activity. The Receive FIFO empty flag, RXFE, is clear when new input data are in the receive FIFO. When the software reads from UART0_DR_R, data are removed from the FIFO. When the FIFO becomes empty, the RXFE flag will be set, meaning there are no more input data. There are other flags associated with the receiver. There is a Receive FIFO full flag RXFF, which is set when the FIFO is full. There are four status bits associated with each byte of data. For this reason, the receive FIFO is 12 bits wide. The overrun error, OE, is set when input data are lost because the FIFO is full and more input frames are arriving at the receiver. An overrun error is caused when the receiver interface latency is too large. i.e. when there are 16 elements in the receive FIFO, and a 17th frame comes into the receiver. The break error, BE, is set when the input is held low for more than a frame. Parity is a mechanism to send one extra bit so the receiver can detect if there were any errors in transmission. With even parity the number of 1's in the data plus parity will be an even number. The PEbit is set on a parity error. The

framing error, FE, is set when the stop bit is incorrect. Framing errors are probably caused by a mismatch in baud rate.

The receiver waits for the 1 to 0 edge signifying a start bit, then shifts in 10 bits of data one at a time from the U0Rx line. The internal clock is 16 times faster than the baud rate. After the 1 to 0 edge, the receiver waits 8 internal clocks and samples the start bit. 16 internal clocks later it samples b0. Every 16 internal clocks it samples another bit until it reaches the stop bit. The UART needs an internal clock faster than the baud rate so it can wait the half a bit time between the 1 to 0 edge beginning the start bit and the middle of the bit window needed for sampling. The start and stop bits are removed (checked for framing errors), the 8 bits of data and 4 bits of status are put into the receive FIFO. The hardware FIFO implements buffering so data is safely stored in the receiver hardware if the software is performing other tasks while data is arriving.



Both FIFOs are accessed via the UART Data (UARTDR) register. Read operations return a 12 bit value consisting of 8 data bits and 4 error flags while write operations place 8 bit data in the transmit FIFO.

FIFOs are enabled by the FEN bit in UARTLCRH. FIFO status can be monitored via the UART Flag (UARTFR) register and the UART Receive Status (UARTRSR) register.

Both FIFOs can be individually configured to trigger interrupts at different levels (1/8, 1/4, 1/2, 3/4, 7/8) via the UART Interrupt FIFO Level Select (UARTIFLS) register. If ¼ is selected for receive FIFO, the UART generates a receive interrupt after 4 data bytes are received.

UART can be configured for 9 bit mode also by UART9BITADDR and UART9BITAMASK registers.

2.5. UART Interrupts

- Overrun Error
- Break Error
- Parity Error
- Framing Error
- Receive Timeout
- Transmit
- Receive

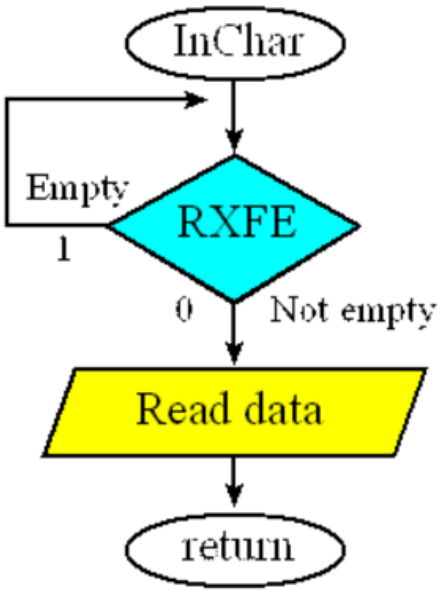
All of the interrupts are ORed together before being set to the interrupt controller. The interrupt events that can trigger a controller level interrupt are defined in the UART Interrupt Mask (UARTIM) register. Masked interrupt status are observed from UARST Masked Interrupt Status (UARTMIS) register. If interrupts are not used, the raw status is visible via

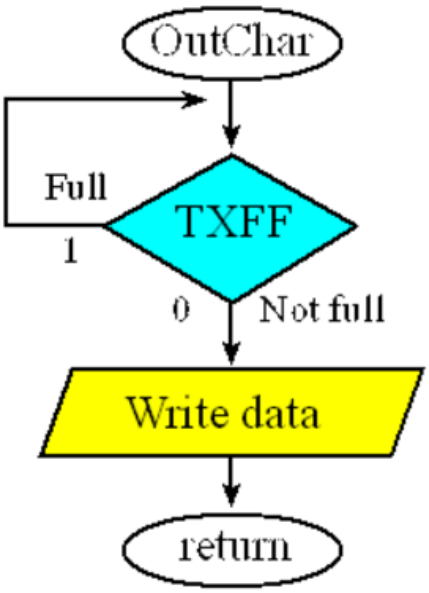
the UART Raw Interrupt Status (UARTRIS) register. Interrupts are cleared by writing a 1 to the corresponding bit in the UART Interrupt Clear (UARTICR) register.

2.6. UART Configuration

The initialization program enables the UART device and selects the baud rate. If UART1 is used PCTL bits 5-4 are set to 0x22 to select U1Tx and U1Rx on PC5 and PC4.

The input routine waits in a loop until RXFE is 0 (FIFO not empty), then reads the data register. The output routine first waits in a loop until TXFF is 0 (FIFO not full), then writes data to the data register. Polling before writing data is an efficient way to perform output.





2.7. UART Registers

Base addresses for UART:

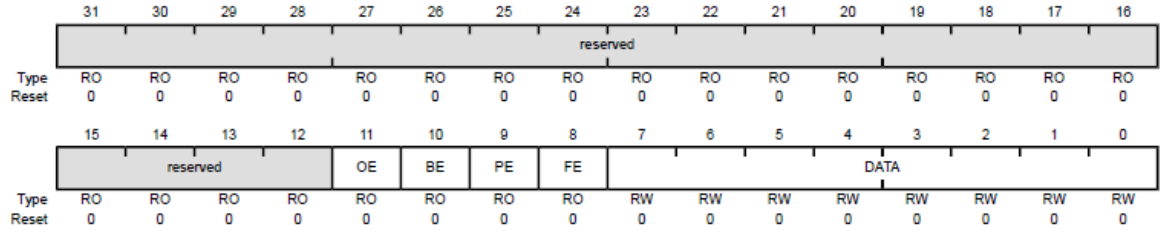
UART1 base: 0x4000.D000
UART2 base: 0x4000.E000
UART3 base: 0x4000.F000
UART4 base: 0x4001.0000
UART5 base: 0x4001.1000
UART6 base: 0x4001.2000
UART7 base: 0x4001.3000

UART Registers and their offset values:

Offset	Name	Type	Reset	Description
0x000	UARTDR	RW	0x0000.0000	UART Data
0x004	UARTRSR/UARTECR	RW	0x0000.0000	UART Receive Status/Error Clear
0x018	UARTFR	RO	0x0000.0090	UART Flag
0x020	UARTILPR	RW	0x0000.0000	UART IrDA Low-Power Register
0x024	UARTIBRD	RW	0x0000.0000	UART Integer Baud-Rate Divisor
0x028	UARTFBRD	RW	0x0000.0000	UART Fractional Baud-Rate Divisor
0x02C	UARTLCRH	RW	0x0000.0000	UART Line Control
0x030	UARTCTL	RW	0x0000.0300	UART Control
0x034	UARTIFLS	RW	0x0000.0012	UART Interrupt FIFO Level Select
0x038	UARTIM	RW	0x0000.0000	UART Interrupt Mask
0x03C	UARTRIS	RO	0x0000.0000	UART Raw Interrupt Status
0x040	UARTMIS	RO	0x0000.0000	UART Masked Interrupt Status
0x044	UARTICR	W1C	0x0000.0000	UART Interrupt Clear
0x048	UARTDMACTL	RW	0x0000.0000	UART DMA Control
0x0A4	UART9BITADDR	RW	0x0000.0000	UART 9-Bit Self Address
0x0A8	UART9BITAMASK	RW	0x0000.00FF	UART 9-Bit Self Address Mask
0xFC0	UARTPP	RO	0x0000.0003	UART Peripheral Properties
0xFC8	UARTCC	RW	0x0000.0000	UART Clock Configuration
0xFD0	UARTPeriphID4	RO	0x0000.0000	UART Peripheral Identification 4
0xFD4	UARTPeriphID5	RO	0x0000.0000	UART Peripheral Identification 5
0xFD8	UARTPeriphID6	RO	0x0000.0000	UART Peripheral Identification 6
0xFDC	UARTPeriphID7	RO	0x0000.0000	UART Peripheral Identification 7
0xFE0	UARTPeriphID0	RO	0x0000.0060	UART Peripheral Identification 0
0xFE4	UARTPeriphID1	RO	0x0000.0000	UART Peripheral Identification 1
0xFE8	UARTPeriphID2	RO	0x0000.0018	UART Peripheral Identification 2
0xFEC	UARTPeriphID3	RO	0x0000.0001	UART Peripheral Identification 3
0xFF0	UARTPCellID0	RO	0x0000.000D	UART PrimeCell Identification 0
0xFF4	UARTPCellID1	RO	0x0000.00F0	UART PrimeCell Identification 1
0xFF8	UARTPCellID2	RO	0x0000.0005	UART PrimeCell Identification 2
0xFFC	UARTPCellID3	RO	0x0000.00B1	UART PrimeCell Identification 3

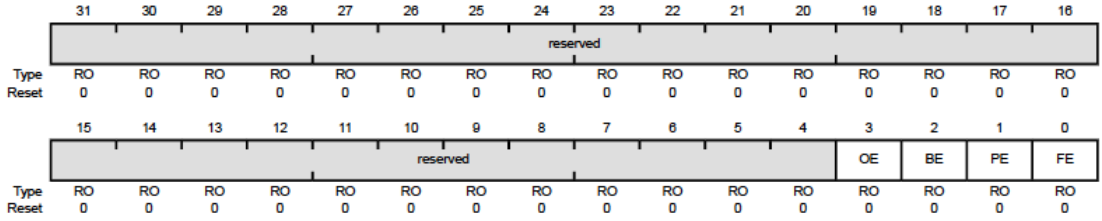
UART Data (UARTDR), offset 0x000

This register is the data register (the interface to the FIFOs). For transmitted data, if the FIFO is enabled, data written to this location is pushed onto the transmit FIFO. If the FIFO is disabled, data is stored in the transmitter holding register (the bottom word of the transmit FIFO). A write to this register initiates a transmission from the UART. For received data, if the FIFO is enabled, the data byte and the 4-bit status (break, frame, parity, and overrun) is pushed onto the 12-bit wide receive FIFO. If the FIFO is disabled, the data byte and status are stored in the receiving holding register (the bottom word of the receive FIFO). The received data can be retrieved by reading this register.



UART Receive Status/Error Clear (UARTRSR/UARTECR), offset 0x004

The UARTRSR/UARTECR register is the receive status register/error clear register. In addition to the UARTDR register, receive status can also be read from the UARTRSR register.



3	OE	RO	0	UART Overrun Error

Value	Description
0	No data has been lost due to a FIFO overrun.
1	New data was received when the FIFO was full, resulting in data loss.

This bit is cleared by a write to **UARTECR**.

The FIFO contents remain valid because no further data is written when the FIFO is full, only the contents of the shift register are overwritten. The CPU must read the data in order to empty the FIFO.

2	BE	RO	0	UART Break Error

Value	Description
0	No break condition has occurred
1	A break condition has been detected, indicating that the receive data input was held Low for longer than a full-word transmission time (defined as start, data, parity, and stop bits).

This bit is cleared to 0 by a write to **UARTECR**.

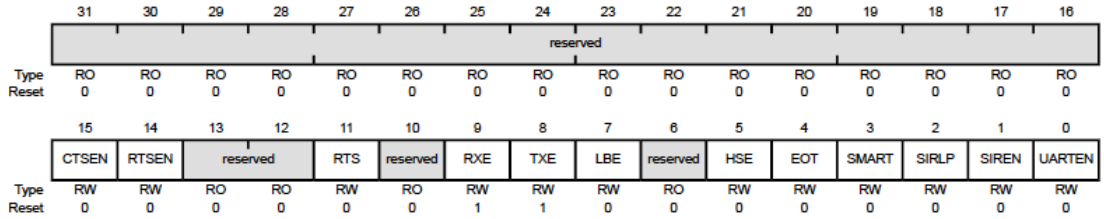
In FIFO mode, this error is associated with the character at the top of the FIFO. When a break occurs, only one 0 character is loaded into the FIFO. The next character is only enabled after the receive data input goes to a 1 (marking state) and the next valid start bit is received.

1	PE	RO	0	UART Parity Error Value Description 0 No parity error has occurred 1 The parity of the received data character does not match the parity defined by bits 2 and 7 of the **UARTLCRH** register. This bit is cleared to 0 by a write to **UARTECR**.
0	FE	RO	0	UART Framing Error Value Description 0 No framing error has occurred 1 The received character does not have a valid stop bit (a valid stop bit is 1). This bit is cleared to 0 by a write to **UARTECR**. In FIFO mode, this error is associated with the character at the top of the FIFO.

Write-Only Error Clear Register: A write to this register of any data clears the framing, parity, break, and overrun flags.

UART Control (UARTCTL), offset 0x030

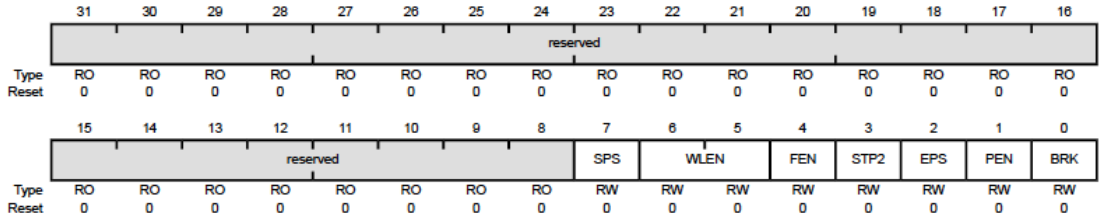
The UARTCTL register should not be changed while the UART is enabled or else the results are unpredictable.



15	CTSEN	RW	0	Enable Clear To Send Value Description 0 CTS hardware flow control is disabled. 1 CTS hardware flow control is enabled. Data is only transmitted when the `U1CTS` signal is asserted.
14	RTSEN	RW	0	Enable Request to Send Value Description 0 RTS hardware flow control is disabled. 1 RTS hardware flow control is enabled. Data is only requested (by asserting `U1RTS`) when the receive FIFO has available entries.
11	RTS	RW	0	Request to Send When `RTSEN` is clear, the status of this bit is reflected on the `U1RTS` signal. If `RTSEN` is set, this bit is ignored on a write and should be ignored on read.

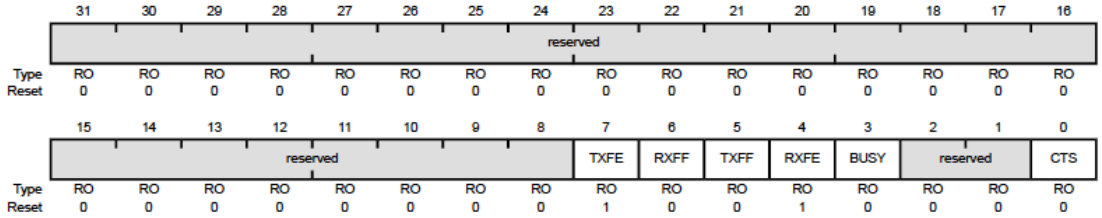
9	RXE	RW	1	UART Receive Enable Value Description 0 The receive section of the UART is disabled. 1 The receive section of the UART is enabled. If the UART is disabled in the middle of a receive, it completes the current character before stopping. **Note:** To enable reception, the UARTEN bit must also be set.
8	TXE	RW	1	UART Transmit Enable Value Description 0 The transmit section of the UART is disabled. 1 The transmit section of the UART is enabled. If the UART is disabled in the middle of a transmission, it completes the current character before stopping. **Note:** To enable transmission, the UARTEN bit must also be set.
7	LBE	RW	0	UART Loop Back Enable Value Description 0 Normal operation. 1 The UnTx path is fed through the UnRx path.
5	HSE	RW	0	High-Speed Enable Value Description 0 The UART is clocked using the system clock divided by 16. 1 The UART is clocked using the system clock divided by 8. **Note:** System clock used is also dependent on the baud-rate divisor configuration (see page 914) and page 915). The state of this bit has no effect on clock generation in ISO 7816 smart card mode (the SMART bit is set).
4	EOT	RW	0	End of Transmission This bit determines the behavior of the TXRIS bit in the **UARTRIS** register. Value Description 0 The TXRIS bit is set when the transmit FIFO condition specified in **UARTIFLS** is met. 1 The TXRIS bit is set only after all transmitted data, including stop bits, have cleared the serializer.
0	UARTEN	RW	0	UART Enable Value Description 0 The UART is disabled. 1 The UART is enabled. If the UART is disabled in the middle of transmission or reception, it completes the current character before stopping.

UART Line Control (UARTLCRH), offset 0x02C



7	SPS	RW	0	UART Stick Parity Select When bits 1, 2, and 7 of **UARTLCRH** are set, the parity bit is transmitted and checked as a 0. When bits 1 and 7 are set and 2 is cleared, the parity bit is transmitted and checked as a 1. When this bit is cleared, stick parity is disabled.
6:5	WLEN	RW	0x0	UART Word Length The bits indicate the number of data bits transmitted or received in a frame as follows: Value Description 0x0 5 bits (default) 0x1 6 bits 0x2 7 bits 0x3 8 bits
4	FEN	RW	0	UART Enable FIFOs Value Description 0 The FIFOs are disabled (Character mode). The FIFOs become 1-byte-deep holding registers. 1 The transmit and receive FIFO buffers are enabled (FIFO mode).
3	STP2	RW	0	UART Two Stop Bits Select Value Description 0 One stop bit is transmitted at the end of a frame. 1 Two stop bits are transmitted at the end of a frame. The receive logic does not check for two stop bits being received. When in 7816 smartcard mode (the SMART bit is set in the **UARTCTL** register), the number of stop bits is forced to 2.
2	EPS	RW	0	UART Even Parity Select Value Description 0 Odd parity is performed, which checks for an odd number of 1s. 1 Even parity generation and checking is performed during transmission and reception, which checks for an even number of 1s in data and parity bits. This bit has no effect when parity is disabled by the PEN bit.
1	PEN	RW	0	UART Parity Enable Value Description 0 Parity is disabled and no parity bit is added to the data frame. 1 Parity checking and generation is enabled.
0	BRK	RW	0	UART Send Break Value Description 0 Normal use. 1 A Low level is continually output on the UnTx signal, after completing transmission of the current character. For the proper execution of the break command, software must set this bit for at least two frames (character periods).

UART Flag (UARTFR), offset 0x018

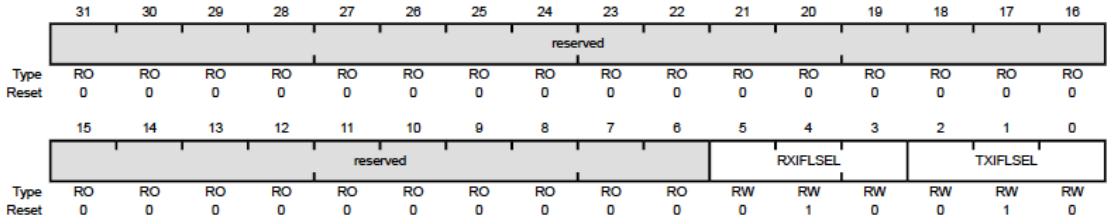


7	TXFE	RO	1	UART Transmit FIFO Empty The meaning of this bit depends on the state of the FEN bit in the **UARTLCRH** register. Value Description 0 The transmitter has data to transmit. 1 If the FIFO is disabled (FEN is 0), the transmit holding register is empty. If the FIFO is enabled (FEN is 1), the transmit FIFO is empty.
6	RXFF	RO	0	UART Receive FIFO Full The meaning of this bit depends on the state of the FEN bit in the **UARTLCRH** register. Value Description 0 The receiver can receive data. 1 If the FIFO is disabled (FEN is 0), the receive holding register is full. If the FIFO is enabled (FEN is 1), the receive FIFO is full.
5	TXFF	RO	0	UART Transmit FIFO Full The meaning of this bit depends on the state of the FEN bit in the **UARTLCRH** register. Value Description 0 The transmitter is not full. 1 If the FIFO is disabled (FEN is 0), the transmit holding register is full. If the FIFO is enabled (FEN is 1), the transmit FIFO is full.
4	RXFE	RO	1	UART Receive FIFO Empty The meaning of this bit depends on the state of the FEN bit in the **UARTLCRH** register. Value Description 0 The receiver is not empty. 1 If the FIFO is disabled (FEN is 0), the receive holding register is empty. If the FIFO is enabled (FEN is 1), the receive FIFO is empty.
3	BUSY	RO	0	UART Busy Value Description 0 The UART is not busy. 1 The UART is busy transmitting data. This bit remains set until the complete byte, including all stop bits, has been sent from the shift register. This bit is set as soon as the transmit FIFO becomes non-empty (regardless of whether UART is enabled).

0	CTS	RO	0	Clear To Send

Value	Description
0	The `ulCTS` signal is not asserted.
1	The `ulCTS` signal is asserted.

UART Interrupt FIFO Level Select (UARTIFLS), offset 0x034



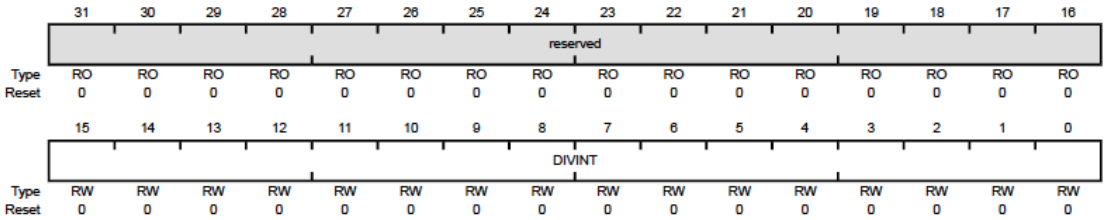
5:3	RXIFLSEL	RW	0x2	UART Receive Interrupt FIFO Level Select The trigger points for the receive interrupt are as follows:

Value	Description
0x0	RX FIFO ≥ ⅛ full
0x1	RX FIFO ≥ ¼ full
0x2	RX FIFO ≥ ½ full (default)
0x3	RX FIFO ≥ ¾ full
0x4	RX FIFO ≥ ⅞ full
0x5-0x7	Reserved

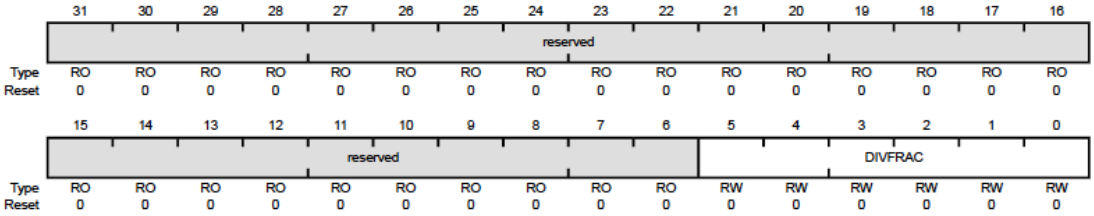
2:0	TXIFLSEL	RW	0x2	UART Transmit Interrupt FIFO Level Select The trigger points for the transmit interrupt are as follows:

Value	Description
0x0	TX FIFO ≤ ⅞ empty
0x1	TX FIFO ≤ ¾ empty
0x2	TX FIFO ≤ ½ empty (default)
0x3	TX FIFO ≤ ¼ empty
0x4	TX FIFO ≤ ⅛ empty
0x5-0x7	Reserved

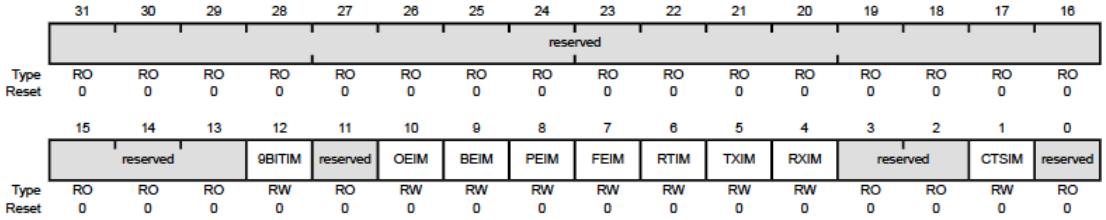
UART Integer Baud-Rate Divisor (UARTIBRD), offset 0x024



UART Fractional Baud-Rate Divisor (UARTFBRD), offset 0x028



UART Interrupt Mask (UARTIM), offset 0x038



12	9BITIM	RW	0	9-Bit Mode Interrupt Mask

Value	Description
0	The `9BITRIS` interrupt is suppressed and not sent to the interrupt controller.
1	An interrupt is sent to the interrupt controller when the `9BITRIS` bit in the **UARTRIS** register is set.

10	OEIM	RW	0	UART Overrun Error Interrupt Mask

Value	Description
0	The `OERIS` interrupt is suppressed and not sent to the interrupt controller.
1	An interrupt is sent to the interrupt controller when the `OERIS` bit in the **UARTRIS** register is set.

9	BEIM	RW	0	UART Break Error Interrupt Mask Value Description 0 The BERIS interrupt is suppressed and not sent to the interrupt controller. 1 An interrupt is sent to the interrupt controller when the BERIS bit in the **UARTRIS** register is set.
8	PEIM	RW	0	UART Parity Error Interrupt Mask Value Description 0 The PERIS interrupt is suppressed and not sent to the interrupt controller. 1 An interrupt is sent to the interrupt controller when the PERIS bit in the **UARTRIS** register is set.
7	FEIM	RW	0	UART Framing Error Interrupt Mask Value Description 0 The FERIS interrupt is suppressed and not sent to the interrupt controller. 1 An interrupt is sent to the interrupt controller when the FERIS bit in the **UARTRIS** register is set.
6	RTIM	RW	0	UART Receive Time-Out Interrupt Mask Value Description 0 The RTRIS interrupt is suppressed and not sent to the interrupt controller. 1 An interrupt is sent to the interrupt controller when the RTRIS bit in the **UARTRIS** register is set.
5	TXIM	RW	0	UART Transmit Interrupt Mask Value Description 0 The TXRIS interrupt is suppressed and not sent to the interrupt controller. 1 An interrupt is sent to the interrupt controller when the TXRIS bit in the **UARTRIS** register is set.
4	RXIM	RW	0	UART Receive Interrupt Mask Value Description 0 The RXRIS interrupt is suppressed and not sent to the interrupt controller. 1 An interrupt is sent to the interrupt controller when the RXRIS bit in the **UARTRIS** register is set.
1	CTSIM	RW	0	UART Clear to Send Modem Interrupt Mask Value Description 0 The CTSRIS interrupt is suppressed and not sent to the interrupt controller. 1 An interrupt is sent to the interrupt controller when the CTSRIS bit in the **UARTRIS** register is set. This bit is implemented only on UART1 and is reserved for UART0 and UART2.

UART Raw Interrupt Status (UARTRIS), offset 0x03C

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved			9BITRIS	reserved	OERIS	BERIS	PERIS	FERIS	RTRIS	TXRIS	RXRIS	reserved		CTSRIS	reserved
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

UART Interrupt Clear (UARTICR), offset 0x044

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved			9BITIC	reserved	OEIC	BEIC	PEIC	FEIC	RTIC	TXIC	RXIC	reserved		CTSMIC	reserved
Type	RO	RO	RO	RW	RO	W1C	W1C	W1C	W1C	W1C	W1C	W1C	RO	RO	W1C	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

2.8. Programming Example

Ex: Write a program to echo every character you hit until you hit SPACEBAR (ASCII code for the SPACEBAR = 0x20)

```
;main

get     BL      InChar
        CMP     R5,#0x20
        BEQ     done
        BL      OutChar
        B       get
done    B       done

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

InChar
        PUSH {R0-R4}

        ; Enable UART clock
        LDR         R1, =RCGCUART
        LDR         R0, [R1]
        ORR         R0, R0, #0x01               ; set bit 0 to enable UART0 clock
        STR         R0, [R1]
        NOP                                     ; Let clock stabilize
        NOP
        NOP
```

```
; Setup GPIO
        ; Enable GPIO clock to use debug USB as com port (PA0, PA1)
        LDR         R1, =RCGCGPIO
        LDR         R0, [R1]
        ORR         R0, R0, #0x01                   ; set bit 0 to enable port A clock
        STR         R0, [R1]
        NOP                                                 ; Let clock stabilize
        NOP
        NOP

        ; Make PA0, PA1 digital
        LDR         R1, =PORTA_DEN
        LDR         R0, [R1]
        ORR         R0, R0, #0x03       ; set bits 1,0 to enable alt functions on PA0, PA1
        STR         R0, [R1]

        ; Disable analog on PA0, PA1
        LDR         R1, =PORTA_AMSEL
        LDR         R0, [R1]
        BIC         R0, R0, #0x03       ; clear bits 1,0 to disable analog on PA0, PA1
        STR         R0, [R1]

        ; Enable alternate functions selected
        LDR         R1, =PORTA_AFSEL
        LDR         R0, [R1]
        ORR         R0, R0, #0x03       ; set bits 1,0 to enable alt functions on PA0, PA1
        STR         R0, [R1]


        ; Select alternate function to be used (UART on PA0, PA1)
        LDR         R1, =PORTA_PCTL
        LDR         R0, [R1]
        ORR         R0, R0, #0x11               ; set bits 4,0 to select UART Rx, Tx
        STR         R0, [R1]

; Setup UART
        ; Disable UART while setting up
        LDR         R1, =UART0_CTL
        LDR         R0, [R1]
        BIC         R0, R0, #0x01                   ; Clear bit 0 to disable UART0
        STR         R0, [R1]

        ; Set baud rate to 9600. Divisor = 16MHz/(16*9600)= 104.16666
        ; Set integer part to 104
        LDR         R1, =UART0_IBRD
        MOV         R0, #104                                ;
        STR         R0, [R1]

        ; Set fractional part
        ;       0.16666*64+0.5 = 11.16666 => Integer = 11
        LDR         R1, =UART0_FBRD
        MOV         R0, #11
        STR         R0, [R1]
```

```
; Set serial parameters
        LDR         R1, =UART0_LCRH
        MOV         R0, #0x70                   ; No stick parity, 8bit, FIFO enabled,
        STR         R0, [R1]                    ; One stop bit, Disable parity, Normal use


        ; Enable UART, TX, RX
        LDR         R1, =UART0_CTL
        LDR         R0, [R1]
        MOV         R2, #0x00000301                     ; Set bits 9,8,0
        ORR         R0, R0, R2                  ; you can also enable just rx
        STR         R0, [R1]


        ; Preload R4 with UART data address
        LDR         R4, =UART0_DR

check
   ; check for incoming character
        LDR    R1, =UART0_FR                    ; load UART status register address
        LDR         R0, [R1]                            ;
        ANDS  R0,R0,#0x10                       ; check if char received (RXFE is 0)
        BNE         check                               ; if no character, check again, else
        LDR         R5, [R4]                            ; load received char into R5

        POP         {R0-R4}

        BX          LR

;.....................................................................
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

OutChar
        PUSH        {R0-R4}                             ; store registers

        ; Enable UART clock
        LDR         R1, =RCGCUART
        LDR         R0, [R1]
        ORR         R0, R0, #0x01                       ; set bit 0 to enable UART0 clock
        STR         R0, [R1]
        NOP                                                     ; Let clock stabilize
        NOP
        NOP

; Setup GPIO
        ; Enable GPIO clock to use debug USB as com port (PA0, PA1)
        LDR         R1, =RCGCGPIO
        LDR         R0, [R1]
        ORR         R0, R0, #0x01                       ; set bit 0 to enable port A clock
        STR         R0, [R1]
        NOP                                                     ; Let clock stabilize
         NOP
        NOP

        ; Make PA0, PA1 digital
        LDR         R1, =PORTA_DEN
```

```
        LDR     R0, [R1]
        ORR     R0, R0, #0x03       ; set bits 1,0 to enable alt functions on PA0, PA1
        STR     R0, [R1]

        ; Disable analog on PA0, PA1
        LDR     R1, =PORTA_AMSEL
        LDR     R0, [R1]
        BIC     R0, R0, #0x03       ; clear bits 1,0 to disable analog on PA0, PA1
        STR     R0, [R1]

        ; Enable alternate functions selected
        LDR     R1, =PORTA_AFSEL
        LDR     R0, [R1]
        ORR     R0, R0, #0x03       ; set bits 1,0 to enable alt functions on PA0, PA1
        STR     R0, [R1]

        ; Select alternate function to be used (UART on PA0, PA1)
        LDR     R1, =PORTA_PCTL
        LDR     R0, [R1]
        ORR     R0, R0, #0x11               ; set bits 4,0 to select UART Rx, Tx
        STR     R0, [R1]


; Setup UART
        ; Disable UART while setting up
        LDR     R1, =UART0_CTL
        LDR     R0, [R1]
        BIC     R0, R0, #0x01               ; Clear bit 0 to disable UART0 while
        STR     R0, [R1]                                ; setting up

        ; Set baud rate to 9600. Divisor = 16MHz/(16*9600)= 104.16666
        ; Set integer part to 104
        LDR     R1, =UART0_IBRD
        MOV     R0, #104                                ;
        STR     R0, [R1]

        ; Set fractional part
        ;       0.16666*64+0.5 = 11.16666 => Integer = 11
        LDR     R1, =UART0_FBRD
        MOV     R0, #11
        STR     R0, [R1]

        ; Set serial parameters
        LDR     R1, =UART0_LCRH
        MOV     R0, #0x70               ; No stick parity, 8bit, FIFO enabled,
        STR     R0, [R1]                ; One stop bit, Disable parity, Normal use

        ;; Enable UART, TX, RX
        ;LDR    R1, =UART0_CTL
        ;LDR    R0, [R1]
        ;MOV    R2, #0x00000301                 ; Set bits 9,8,0
        ;ORR    R0, R0, R2   ; you can also enable just transmit
        ;STR    R0, [R1]
        ;POP    {R0-R4}                                 ; restore registers
        ;BX     LR
```

```
        ; load flag register location
        LDR    R1, =UART0_FR

waitR
        ; check if UART is ready (buffer is empty)
                                ; load UART status register address
        LDR             R0, [R1]
         ANDS R0,R0,#0x20               ; check if TXFF = 1
        BNE    waitR                   ; If so, UART is full, so wait / check again

        ; load R4 with UART data address
        LDR             R4, =UART0_DR
        STR             R5,[R4]                         ; Put character in UART data register


waitD
        ; check if UART is done transmitting
        LDR             R0, [R1]
        ANDS  R0,R0,#0x08               ; check if BUSY = 1
        BNE    waitD                    ; If so, UART is busy, so wait / check again

        ; Disable UART, TX, Disable RX
        LDR             R1, =UART0_CTL
        MOV             R2, #0x00000000                         ; clear bits 8,0
        STR             R2, [R1]



        POP             {R0-R4}                                         ; restore registers
        BX      LR
```

3. SPI (Serial Peripheral Interface)

The Serial Peripheral Interface, or SPI, is a synchronous serial interface. One of the key benefits of the SPI interface is that it provides much higher data transfer rates than UART and I^2C interfaces. Increased data speeds make SPI a popular interface for embedded systems that interface with large banks of non-volatile storage (SD and microSD). The SPI bus is also used by modern PCs to load the BIOS. While the SPI bus provides increased data throughput over I^2C and UART, it also requires more pins than the UART and I^2C interfaces.

So what makes SPI faster than a UART? The SPI bus achieves higher data transfer rates because two interconnected devices share a common clock. The two devices synchronize data transfers using a common clock and data is transmitted on every clock edge. There is no over sampling of the SPI bus. This differs from a UART interface that must sample the data for several clock cycles to determine the logic value of each bit.

3.1. Master/Slave Configuration

A SPI bus is designed as a point-to-point interface. This means that only two devices are interconnected by a single SPI bus. The synchronous nature of SPI also requires that one of the devices generates a clock that will be used to determine when data is shifted out of one device and into the other. The device that generates the clock is said to be the master device. The device that does not generate the clock is said to be the slave device. Because the master device is the only device that can generate a clock, the master device determines when all data transactions occur. In most cases, the master device is a microprocessor and the slave device is a device that provides a specific function to the embedded system (LCD screen, non-volatile storage, accelerometer, etc).

The master-slave topology of the SPI bus is fundamentally different than that of a UART. A UART interface is asynchronous. Either end of a UART communication channel can initiate data transfer because transmitting and receiving data are completely independent and neither device is responsible for generating a shared clock signal. The asynchronous nature of the UART interface is very handy for sensors that inform the microprocessor that an input to the system has changed. A UART based sensor can transmit data to the microprocessor at any time. A SPI based sensor does not have this ability. Since the sensor is almost always a slave device, it cannot generate the synchronizing clock to transmit data to the master. So how does the data get to the master? There are two solutions to this problem. The first is that the master to periodically poll the sensor. Like any polling based scheme, this method is simple but it also consumes clock cycles trying to determine if the sensor has important data. The second approach is for the sensor to generate an interrupt on one of the external pins of the microprocessor. When the sensor is trying to report data, it asserts the interrupt indicating that it requires attention. The microprocessor must be configured to generate an interrupt when the sensor asserts a specific pin. The interrupt service routine for the pin initiates a SPI data transfer and reads the data.

3.2. SPI Bus Specifics

The SPI interface consists of 4 dedicated pins.

CLK

The SPI clock determines the rate at which data transfer occur. The maximum speed of the clock is determined by analyzing both the master and slave devices. Each device will support a maximum clock rate. The device with the slowest maximum clock rate determines the maximum clock speed. The SPI clock is only active during data transfers. When data is not being transferred, the clock will be idle.

MOSI

The Master Out/Slave In signal is used by the master device to transfer data to the slave device. Data is transmitted most significant bit first.

MISO

The Master In/Slave Out signal is used by the master device to receive data from the slave device. Data is transmitted most significant bit first.

CS

The SPI interface requires an **active low** chip select to frame the number of bytes being transmitted. Unlike the UART interface, there are no start or stop bits used to determine when data is valid. The chip select frames the data being transmitted. By eliminating the start and stop bits allow us to transmit 8 bits of data in 8 clock cycles. The UART interface has to transmit 10 symbols for every 8 bits of data due to the start and stop bits.

A SPI transaction consists of all the data being sent while the CS line remains low. The master SPI device controls the CS line, but here may be situations where the CS line is configured as a normal GPIO pin and software will determine the value on the line. When sending a large amount of data in a single transaction, software may control the CS line because the amount of data being transferred exceeds the capacity of the transmit FIFO.

3.3. SPI Modes

When interfacing with another device, the SPI bus must be configured so that both devices agree on two things: how the clock behaves during periods of inactivity and when to sample the data. These two characteristics are commonly referred to as the clock phase and clock polarity.

The clock polarity determines if the clock is brought to logic 0 or logic 1 when data is not being sent. If the polarity is set to 0, then the clock will be pulled to logic 0 when inactive. When the polarity is set to 1, the clock will be pulled to logic 1 when inactive.

The clock phase indicates on which edge the device will sample the first byte of data. If the phase is equal to 0, the data is sampled on the first edge of the clock. If the phase is equal to 1, the data is sampled on the second edge of the clock.

Using both the polarity and phase, we can define 4 different SPI modes (0 through 3). Some devices support all 4 SPI modes, other will only support a single SPI mode. The microprocessor will need to be configured for a SPI mode that is supported by the slave device.

3.4. TM4C123 SPI Features

The SPI peripheral on the TM4C123 is called the SSI peripheral or Synchronous Serial Interface. It supports three different variants of SPI (Freescale SPI, MICROWIRE, and the TI SSI interface). Most devices use the Freescale SPI configuration.

The SPI interface has an 8 entry FIFO. The length of the data being transmitted can be anywhere from 4 to 16 bits. Data is placed in the transmit FIFO by writing to the data register (DR). Data is removed from the receive FIFO by reading from the data register (DR).

3.5. SPI pin assignment

Pins should be configured using GPIO Alternate Function Select (GPIOAFSEL) and GPIO Port Control (GPIOPCTL) registers.

Pin Name	Pin Number	Pin Mux / Pin Assignment	Pin Type	Buffer Type[a]	Description
SSI0Clk	19	PA2 (2)	I/O	TTL	SSI module 0 clock
SSI0Fss	20	PA3 (2)	I/O	TTL	SSI module 0 frame signal
SSI0Rx	21	PA4 (2)	I	TTL	SSI module 0 receive
SSI0Tx	22	PA5 (2)	O	TTL	SSI module 0 transmit
SSI1Clk	30 61	PF2 (2) PD0 (2)	I/O	TTL	SSI module 1 clock.
SSI1Fss	31 62	PF3 (2) PD1 (2)	I/O	TTL	SSI module 1 frame signal.
SSI1Rx	28 63	PF0 (2) PD2 (2)	I	TTL	SSI module 1 receive.
SSI1Tx	29 64	PF1 (2) PD3 (2)	O	TTL	SSI module 1 transmit.
SSI2Clk	58	PB4 (2)	I/O	TTL	SSI module 2 clock.
SSI2Fss	57	PB5 (2)	I/O	TTL	SSI module 2 frame signal.
SSI2Rx	1	PB6 (2)	I	TTL	SSI module 2 receive.
SSI2Tx	4	PB7 (2)	O	TTL	SSI module 2 transmit.
SSI3Clk	61	PD0 (1)	I/O	TTL	SSI module 3 clock.
SSI3Fss	62	PD1 (1)	I/O	TTL	SSI module 3 frame signal.
SSI3Rx	63	PD2 (1)	I	TTL	SSI module 3 receive.
SSI3Tx	64	PD3 (1)	O	TTL	SSI module 3 transmit.

3.6. Bit Rate Generation

The SPI includes a programmable bit rate clock divider and prescaler to generate the serial output clock. Bit rates are supported to 2 MHz and higher, although maximum bit rate is determined by peripheral devices.

The serial bit rate is derived by dividing down the input clock (SysClk). The clock is first divided by an even prescale value CPSDVSR from 2 to 256, which is programmed in SSI Clock Prescale (SSICPSR) register. The clock is further divided by a value from 1 to 256, which is 1+ SCR, where SCR is the value programmed in the SSI Control 0 (SSICR0) register.

The frequency of the output clock SSInClk is defined by:

SSInClk = SysClk / (CPSDVSR * (1 + SCR))

3.7. FIFO

Transmit FIFO:

The common transmit FIFO is a 16-bit wide, 8-locations deep, first-in, first-out memory buffer. The CPU writes data to the FIFO by writing the SSI Data (SSIDR) register, and data is stored in the FIFO until it is read out by the transmission logic.

When configured as a master or a slave, parallel data is written into the transmit FIFO prior to serial conversion and transmission to the attached slave or master, respectively, through the SSInTx pin.

In slave mode, the SSI transmits data each time the master initiates a transaction. If the transmit FIFO is empty and the master initiates, the slave transmits the 8th most recent value in the transmit FIFO. If less than 8 values have been written to the transmit FIFO since the SSI module clock was enabled using the Rn bit in the RCGCSSI register, then 0 is transmitted. Care should be taken to ensure that valid data is in the FIFO as needed. The SSI can be configured to generate an interrupt or a µDMA request when the FIFO is empty.

ReceiveFIFO:

The common receive FIFO is a 16-bit wide, 8-locations deep, first-in, first-out memory buffer. Received data from the serial interface is stored in the buffer until read out by the CPU, which accesses the read FIFO by reading the SSIDR register.

When configured as a master or slave, serial data received through the SSInRx pin is registered prior to parallel loading into the attached slave or master receive FIFO, respectively.

3.8. Interrupts

The SSI can generate interrupts when the following conditions are observed:

- Transmit FIFO service (when the transmit FIFO is half full or less)
- Receive FIFO service (when the receive FIFO is half full or more)
- Receive FIFO time-out
- Receive FIFO overrun
- End of transmission

All of the interrupt events are ORed together before being sent to the interrupt controller, so the SSI generates a single interrupt request to the controller regardless of the number of active interrupts.

Each of the four individual maskable interrupts can be masked by clearing the appropriate bit in the SSI Interrupt Mask (SSIIM) register. Setting the appropriate mask bit enables the interrupt. The individual outputs, along with a combined interrupt output, allow use of either a global interrupt service routine or modular device drivers to handle interrupts. The transmit and receive dynamic data flow interrupts have been separated from the status interrupts so that data can be read or written in response to the FIFO trigger levels.

The status of the individual interrupt sources can be read from the SSI Raw Interrupt Status (SSIRIS) and SSI Masked Interrupt Status (SSIMIS) registers. The receive FIFO has a time-out period that is 32 periods at the rate of SSInClk (whether or not SSInClk is currently active) and is started when the RX FIFO goes from EMPTY to not-EMPTY. If the RXFIFO is emptied before 32 clocks have passed, the time-out period is reset. As a result, the ISR should clear the Receive FIFO Time-out Interrupt just after reading out the RXFIFO by writing a 1 to the RTIC bit in the SSI Interrupt Clear (SSIICR) register. The interrupt should not be cleared so late that the ISR returns before the interrupt is actually cleared, or the ISR may be re-activated unnecessarily.

The End-of-Transmission (EOT) interrupt indicates that the data has been transmitted completely and is only valid for Master mode devices/operations. This interrupt can be used to indicate when it is safe to turn off the SSImoduleclock or enter sleep mode. In addition, because transmitted data and received data complete at exactly the same time, the interrupt can also indicate that read data is ready immediately, without waiting for the receive FIFO time-out period to complete.

3.9. SPI Configuration

GPIO Configuration

1. Enable the clock for GPIOx (**RCGCGPIO**)
2. Wait until GPIOx is ready (**PRGPIO**)
3. Configure the CLK, CS (FSS), MOSI (Tx), and MISO(Rx) pins as a digital pin (**DEN**)
4. Configure the CLK, CS (FSS), MOSI (Tx), and MISO(Rx) pins for their alternate function (**AFSEL**)
5. Configure the CLK, CS (FSS), MOSI (Tx), and MISO(Rx) port control pins to route the SSI interface to the pins (**PCTL**)

SPI Configuration (Polled)

1. Enable the clock for SSIx (**RCGCSSI**)
2. Wait for the SSI peripheral to be ready (**PRSSI**)
3. Disable the SPI interface (**CR1**)
4. Set the clock rate of the SPI Clock (**CPSR, CR0**)
5. Set the data size to be 8-bits and Freescale mode (**CR0**)
6. Set the SPI mode (**CR0**)
7. Re-enable the SPI interface (**CR1**)

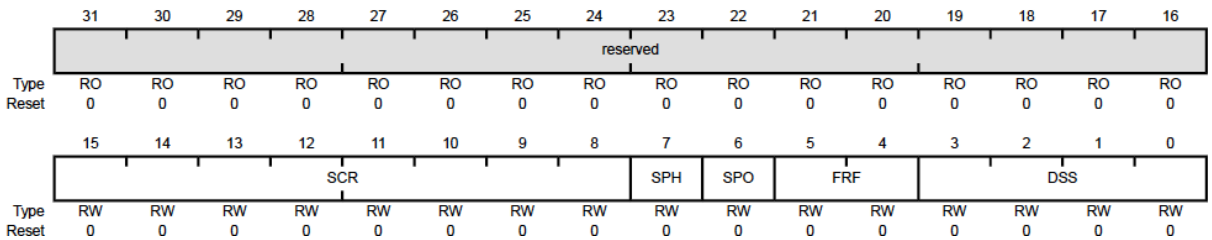
3.10. SPI Registers

Base addresses of the modules:

SSI0: 0x4000.8000
SSI1: 0x4000.9000
SSI2: 0x4000.A000
SSI3: 0x4000.B000

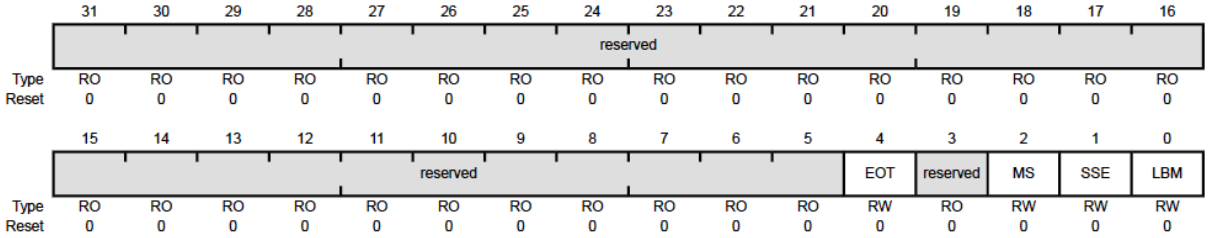
Offset	Name	Type	Reset	Description
0x000	SSICR0	RW	0x0000.0000	SSI Control 0
0x004	SSICR1	RW	0x0000.0000	SSI Control 1
0x008	SSIDR	RW	0x0000.0000	SSI Data
0x00C	SSISR	RO	0x0000.0003	SSI Status
0x010	SSICPSR	RW	0x0000.0000	SSI Clock Prescale
0x014	SSIIM	RW	0x0000.0000	SSI Interrupt Mask
0x018	SSIRIS	RO	0x0000.0008	SSI Raw Interrupt Status
0x01C	SSIMIS	RO	0x0000.0000	SSI Masked Interrupt Status
0x020	SSIICR	W1C	0x0000.0000	SSI Interrupt Clear
0x024	SSIDMACTL	RW	0x0000.0000	SSI DMA Control
0xFC8	SSICC	RW	0x0000.0000	SSI Clock Configuration
0xFD0	SSIPeriphID4	RO	0x0000.0000	SSI Peripheral Identification 4
0xFD4	SSIPeriphID5	RO	0x0000.0000	SSI Peripheral Identification 5
0xFD8	SSIPeriphID6	RO	0x0000.0000	SSI Peripheral Identification 6
0xFDC	SSIPeriphID7	RO	0x0000.0000	SSI Peripheral Identification 7
0xFE0	SSIPeriphID0	RO	0x0000.0022	SSI Peripheral Identification 0
0xFE4	SSIPeriphID1	RO	0x0000.0000	SSI Peripheral Identification 1
0xFE8	SSIPeriphID2	RO	0x0000.0018	SSI Peripheral Identification 2
0xFEC	SSIPeriphID3	RO	0x0000.0001	SSI Peripheral Identification 3
0xFF0	SSIPCellID0	RO	0x0000.000D	SSI PrimeCell Identification 0
0xFF4	SSIPCellID1	RO	0x0000.00F0	SSI PrimeCell Identification 1
0xFF8	SSIPCellID2	RO	0x0000.0005	SSI PrimeCell Identification 2
0xFFC	SSIPCellID3	RO	0x0000.00B1	SSI PrimeCell Identification 3

SSI Control 0 (SSICR0), offset 0x000



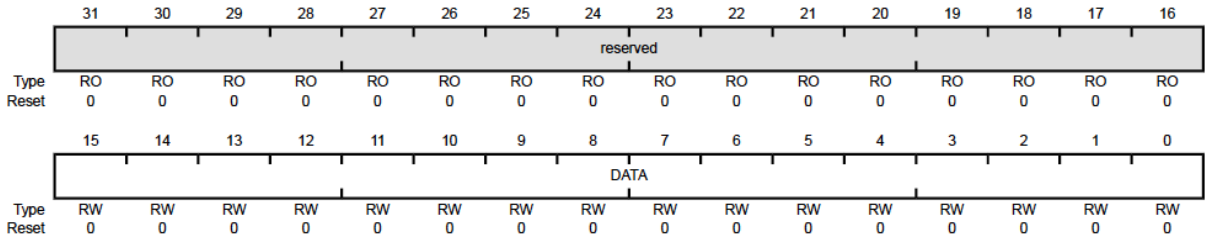
15:8	SCR	RW	0x00	SSI Serial Clock Rate This bit field is used to generate the transmit and receive bit rate of the SSI. The bit rate is: `BR=SysClk/(CPSDVSR * (1 + SCR))` where `CPSDVSR` is an even value from 2-254 programmed in the **SSICPSR** register, and `SCR` is a value from 0-255.
7	SPH	RW	0	SSI Serial Clock Phase This bit is only applicable to the Freescale SPI Format. The `SPH` control bit selects the clock edge that captures data and allows it to change state. This bit has the most impact on the first bit transmitted by either allowing or not allowing a clock transition before the first data capture edge. Value Description 0 Data is captured on the first clock edge transition. 1 Data is captured on the second clock edge transition.
6	SPO	RW	0	SSI Serial Clock Polarity Value Description 0 A steady state Low value is placed on the `SSInClk` pin. 1 A steady state High value is placed on the `SSInClk` pin when data is not being transferred. **Note:** If this bit is set, then software must also configure the GPIO port pin corresponding to the `SSInClk` signal as a pull-up in the **GPIO Pull-Up Select (GPIOPUR)** register.
5:4	FRF	RW	0x0	SSI Frame Format Select Value Frame Format 0x0 Freescale SPI Frame Format 0x1 Texas Instruments Synchronous Serial Frame Format 0x2 MICROWIRE Frame Format 0x3 Reserved
3:0	DSS	RW	0x0	SSI Data Size Select Value Data Size 0x0-0x2 Reserved 0x3 4-bit data 0x4 5-bit data 0x5 6-bit data 0x6 7-bit data 0x7 8-bit data 0x8 9-bit data 0x9 10-bit data 0xA 11-bit data 0xB 12-bit data 0xC 13-bit data 0xD 14-bit data 0xE 15-bit data 0xF 16-bit data

SSI Control 0 (SSICR0), offset 0x000

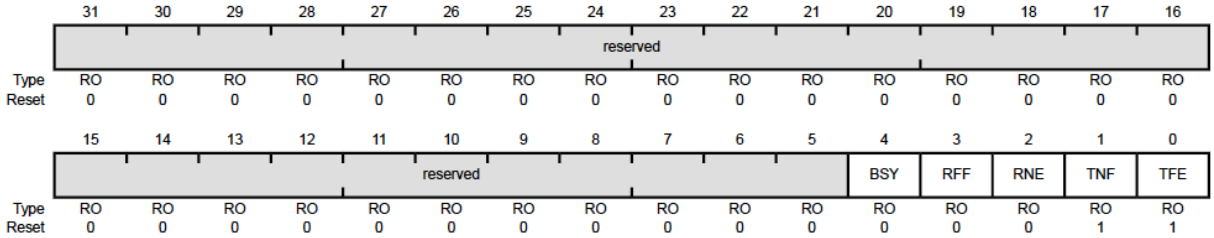


4	EOT	RW	0	End of Transmission This bit is only valid for Master mode devices and operations (MS = 0x0). Value Description 0 The TXRIS interrupt indicates that the transmit FIFO is half full or less. 1 The End of Transmit interrupt mode for the TXRIS interrupt is enabled.
2	MS	RW	0	SSI Master/Slave Select This bit selects Master or Slave mode and can be modified only when the SSI is disabled (SSE=0). Value Description 0 The SSI is configured as a master. 1 The SSI is configured as a slave.
1	SSE	RW	0	SSI Synchronous Serial Port Enable Value Description 0 SSI operation is disabled. 1 SSI operation is enabled. **Note:** This bit must be cleared before any control registers are reprogrammed.
0	LBM	RW	0	SSI Loopback Mode Value Description 0 Normal serial port operation enabled. 1 Output of the transmit serial shift register is connected internally to the input of the receive serial shift register.

SSI Data (SSIDR), offset 0x008



SSI Status (SSISR), offset 0x00C



Bit/Field	Name	Type	Reset	Description
4	BSY	RO	0	SSI Busy Bit

Value	Description
0	The SSI is idle.
1	The SSI is currently transmitting and/or receiving a frame, or the transmit FIFO is not empty.

Bit/Field	Name	Type	Reset	Description
3	RFF	RO	0	SSI Receive FIFO Full

Value	Description
0	The receive FIFO is not full.
1	The receive FIFO is full.

Bit/Field	Name	Type	Reset	Description
2	RNE	RO	0	SSI Receive FIFO Not Empty

Value	Description
0	The receive FIFO is empty.
1	The receive FIFO is not empty.

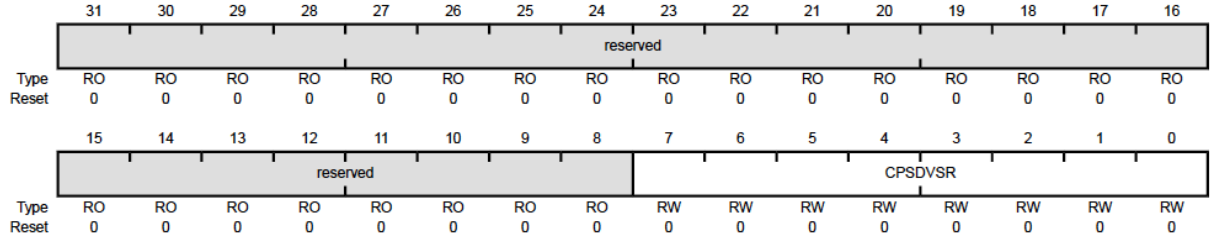
Bit/Field	Name	Type	Reset	Description
1	TNF	RO	1	SSI Transmit FIFO Not Full

Value	Description
0	The transmit FIFO is full.
1	The transmit FIFO is not full.

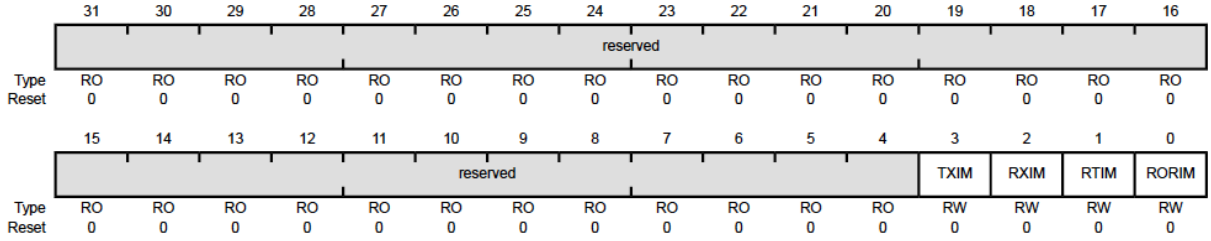
Bit/Field	Name	Type	Reset	Description
0	TFE	RO	1	SSI Transmit FIFO Empty

Value	Description
0	The transmit FIFO is not empty.
1	The transmit FIFO is empty.

SSI Clock Prescale (SSICPSR), offset 0x010



SSI Interrupt Mask (SSIIM), offset 0x014



3	TXIM	RW	0	SSI Transmit FIFO Interrupt Mask

Value	Description
0	The transmit FIFO interrupt is masked.
1	The transmit FIFO interrupt is not masked.

2	RXIM	RW	0	SSI Receive FIFO Interrupt Mask

Value	Description
0	The receive FIFO interrupt is masked.
1	The receive FIFO interrupt is not masked.

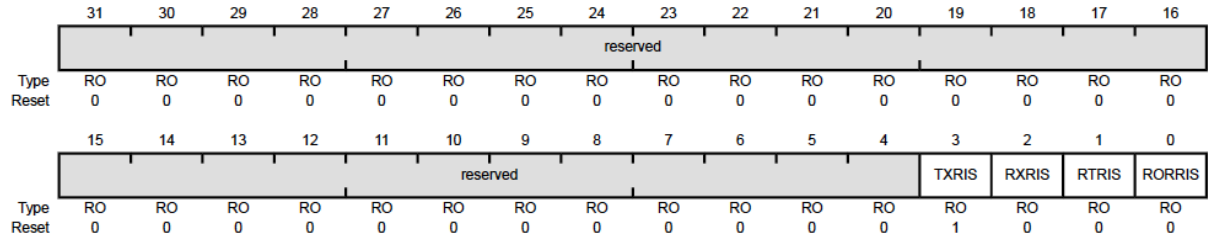
1	RTIM	RW	0	SSI Receive Time-Out Interrupt Mask

Value	Description
0	The receive FIFO time-out interrupt is masked.
1	The receive FIFO time-out interrupt is not masked.

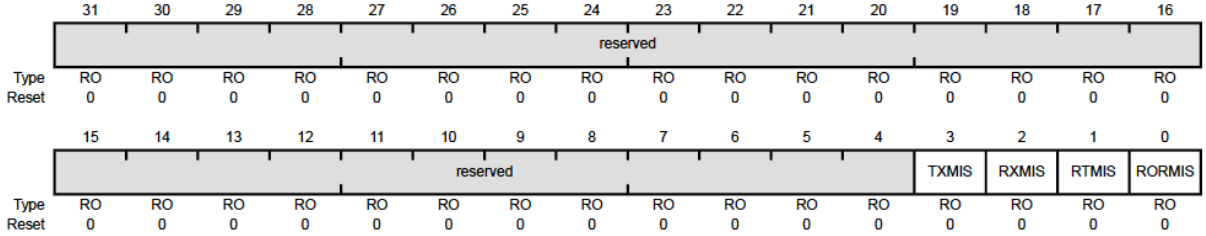
0	RORIM	RW	0	SSI Receive Overrun Interrupt Mask

Value	Description
0	The receive FIFO overrun interrupt is masked.
1	The receive FIFO overrun interrupt is not masked.

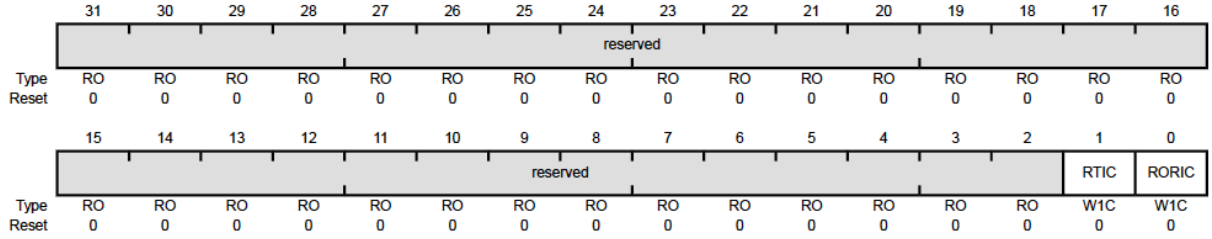
SSI Raw Interrupt Status (SSIRIS), offset 0x018



SSI Masked Interrupt Status (SSIMIS), offset 0x01C

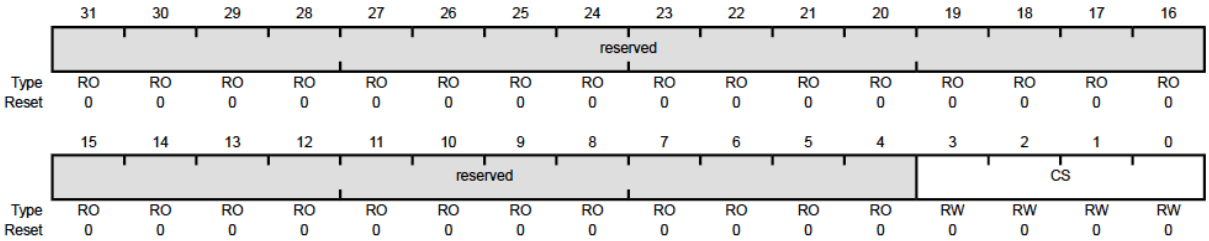


SSI Interrupt Clear (SSIICR), offset 0x020



1	RTIC	W1C	0	SSI Receive Time-Out Interrupt Clear Writing a 1 to this bit clears the `RTRIS` bit in the **SSIRIS** register and the `RTMIS` bit in the **SSIMIS** register.
0	RORIC	W1C	0	SSI Receive Overrun Interrupt Clear Writing a 1 to this bit clears the `RORRIS` bit in the **SSIRIS** register and the `RORMIS` bit in the **SSIMIS** register.

SSI Clock Configuration (SSICC), offset 0xFC8



3:0	CS	RW	0	SSI Baud Clock Source The following table specifies the source that generates for the SSI baud clock:

Value	Description
0x0	System clock (based on clock source and divisor factor)
0x1-0x4	reserved
0x5	PIOSC
0x6 - 0xF	Reserved

APPENDIX

A1. Supplementary material to Chapter 1

I/O mapping

Memory Mapped I/O and Isolated I/O are two methods of performing input-output operations between CPU and installed peripherals in the system. Memory mapped I/O uses the same address bus to connect both primary memory and memory of hardware devices. Thus the instruction to address a section or portion or segment of RAM can also be used to address a memory location of a hardware device.

On the other hand, isolated I/O uses separate instruction classes to access primary memory and device memory. In this case, I/O devices have separate address space either by separate I/O pin on CPU or by entire separate bus. As it separates general memory addresses with I/O devices, it is called isolated I/O.

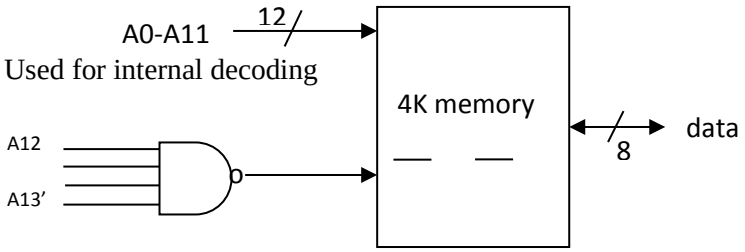
Address Decoding

MCU enables a memory chip or I/O device only when it wants to address it. To do this, MCU uses an address decoding circuit. The signals on the address pins determine which memory location, I/O device is selected or addressed. Address decoding methods are:

1. Full (exhaustive) decoding: Each peripheral is assigned to a unique address. All address bits are used to define the referenced location.
2. Partial decoding: Not all address bits are used in the decoding process. Peripherals may respond to more than one address. Reduced complexity in the decoder.

Basic address decoder uses a NAND gate

Example: You have a 4Kx8 bit memory chip and 16 address lines coming out of the MCU. Then 12 address lines should be used for internal decoding and 4 address lines should be used for chip select. Assume that you want to decode the addresses between $D000 and $DFFF.



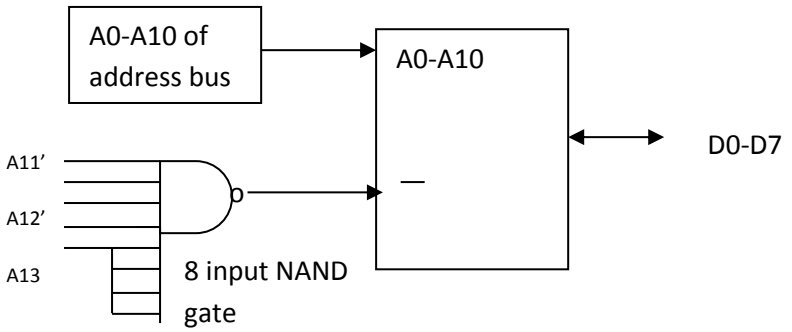
A15	A14	A13	A12	A11	A10	……..	A1	A0	
1	1	0	1	0	0		0	0	:$D000
1	1	0	1	1	1		1	1	:$DFFF

used for chip selection used for internal memory addressing

Example: For a particular application it is required to use a 2716 (2Kx8bit) EPROM for memory locations $A000 thru $A7FF. Design a decoder circuit using NAND gates.

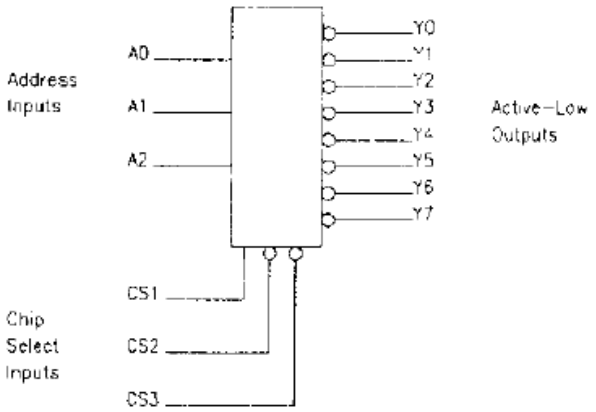
A15	A14	A13	A12	A11	A10	……..	A1	A0	
1	0	1	0	0	0		0	0	:$A000
1	0	1	0	0	1		1	1	:$A7FF

used for chip selection used for internal EPROM addressing



Integrated decoder circuits for address decoding:

In most applications, more than one memory device is usually required for system operation. For many applications 74HC138 (3-to-8 line decoder) is used.



Chip Select Conditions
CS1 = 1
CS2 = 0
CS3 = 0

All outputs high for other chip select conditions

Inputs			Outputs							
A2	A1	A0	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
0	0	0	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	1	1
0	1	0	1	1	0	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1
1	0	0	1	1	1	1	0	1	1	1
1	0	1	1	1	1	1	1	0	1	1
1	1	0	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	0

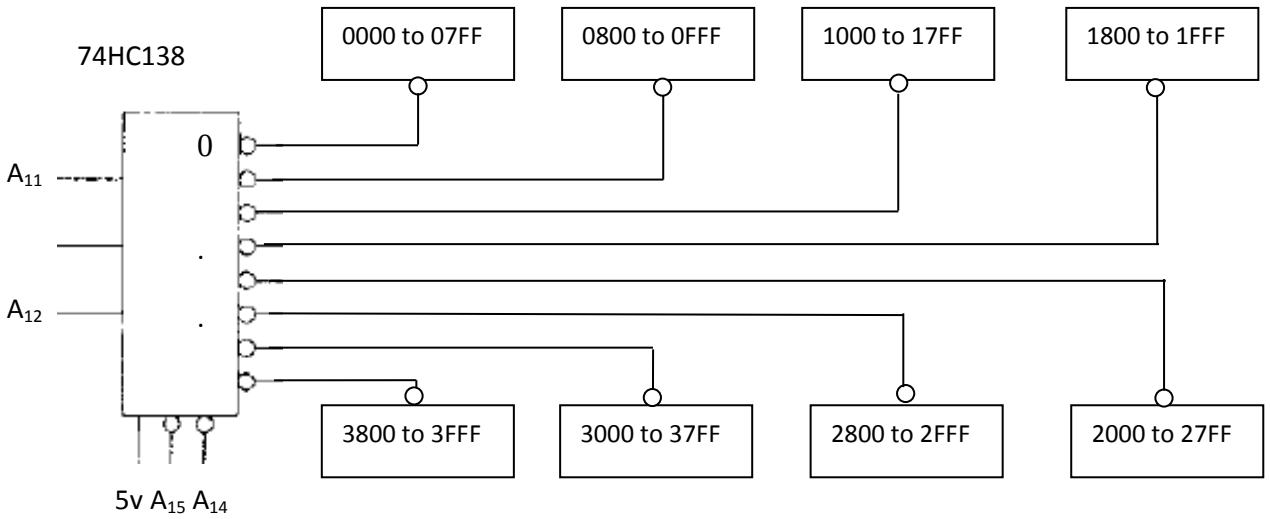
74HC138 (3-to-8 line decoder)

Example: Connect eight 2716 (2K) EPROMs at memory addresses $0000 thru $3FFF.

This can be achieved using eight 8-input NAND gates and a number of inverters or a single 74HC138 decoder.

A15	A14	A13	A12	A11	A10	A9	……..	A1	A0	
0	0	0	0	0	x	x		x	x	EPROM1
0	0	0	0	1	x	x		x	x	EPROM2
.										
.										
.										
0	0	1	1	1	x	x		x	x	EPROM8

Bit 15,14: Used for the selection of this eight-chip portion in the whole 64K address space.
Bit 13,12,11: Used for 1 out of 8 decoder (EPROM chip) selection
Bit 10,..,0: Used for internal decoding of the selected EPROM chip

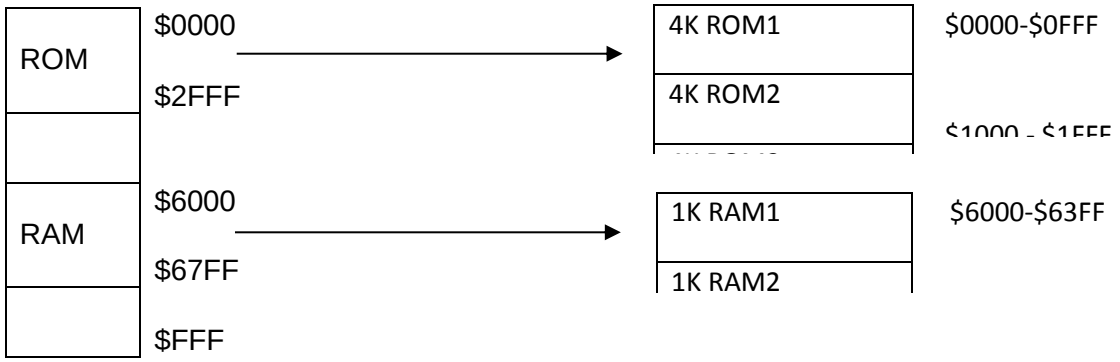


Decoding circuit

Static memory systems

A memory map illustrates which segments are to be used for RAM, ROM, and in some cases where the I/O resides.

Example: The following memory map is given and it is required to be implemented using three 4K ROM and two 1K RAM chips available.



For RAM internal addressing we need 10 bits (A0-A9). A10-A15 are left for RAM selection.
For ROM internal addressing we need 12 bits (A0-A11). A12-A15 are left for ROM selection.

A15	A14	A13	A12	A11	A10	A9	……..	A1	A0	
0	0	0	0	x	x	x		x	x	ROM1
0	0	0	1	x	x	x		x	x	ROM2
0	0	1	0	x	x	x		x	x	ROM3
0	1	1	0	0	0	x		x	x	RAM1
0	1	1	0	0	1	x		x	x	RAM2

1 of 3 ROM selection

1 of 2 RAM selection

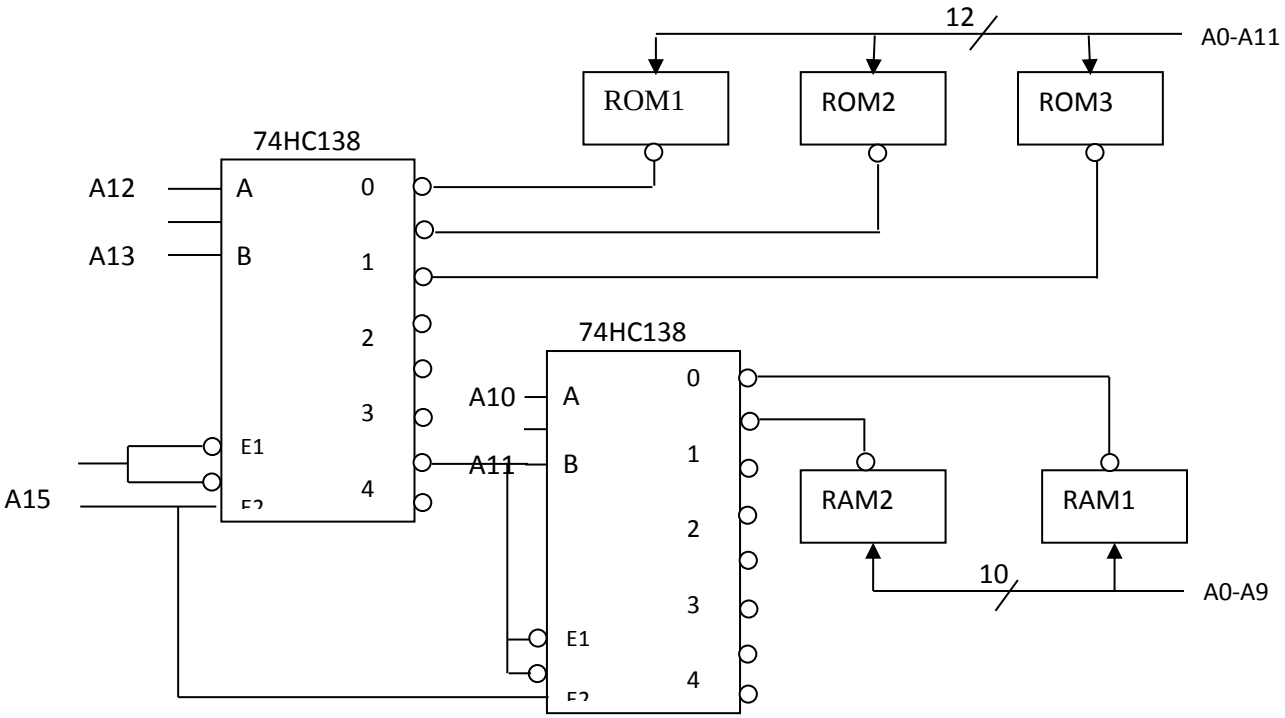
Internal decoding for RAMs

Internal decoding for ROMs

RAM/ROM selection

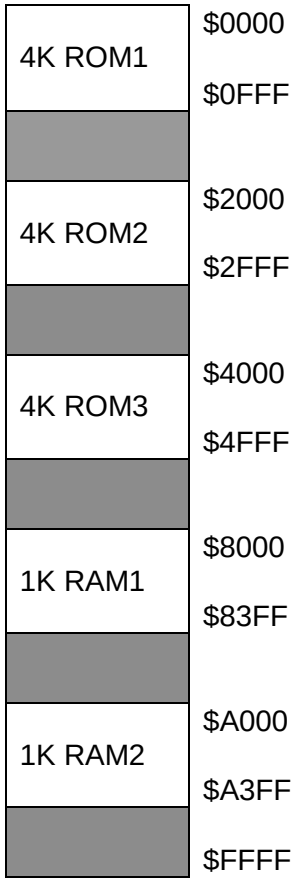
May be used for active enable input if a PROM decoder is used.

Using 74HC138 decoders



Memory Overlaying

Example: Design a decoder circuit for the following memory map.



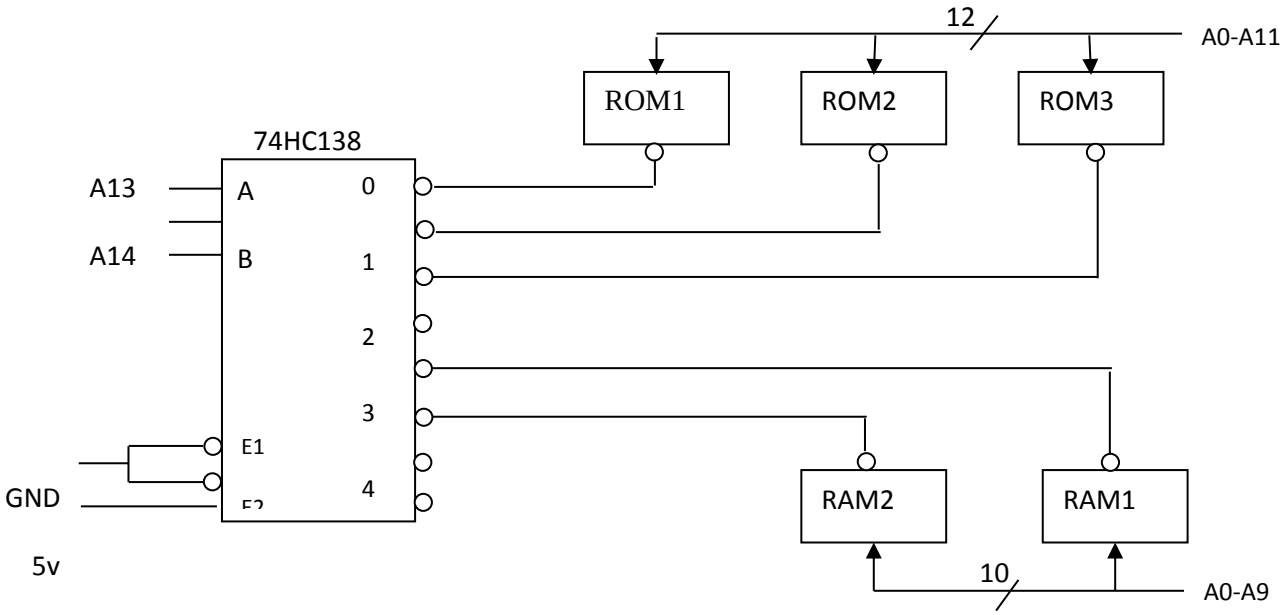
Overlaying:

Each 4 Kbyte ROM may overlay 8 Kbytes of memory

A12 is always 0 and not used in the following decoding circuit

A15	A14	A13	A12	A11	A10	A9	……..	A1	A0	
0	0	0	0	x	x	x		x	x	ROM1
0	0	1	0	x	x	x		x	x	ROM2
0	1	0	0	x	x	x		x	x	ROM3
1	0	0	0	0	0	x		x	x	RAM1
1	0	1	0	0	0	x		x	x	RAM2

Can be used for chip selection with overlays



A2. Thumb2 Instruction Set

REV16	Rd, Rn	Reverse Byte Order in a Halfword, Rd[15:8]←Rn[7:0], Rd[7:0]←Rn[15:8], Rd[31:24]←Rn[23:16], Rd[23:16]←Rn[31:24]
REVSH	Rd, Rn	Reverse Byte order in Low Halfword and sign extend, Rd[15:8]←Rn[7:0], Rd[7:0]←Rn[15:8], Rd[31:16]←Rn[7]*&FFFF
ROR, RORS	Rd, Rm, <Rs|#n>	Rotate Right, Rd ← ROR(Rm, Rs|n), RORS updates N,Z,C
RRX, RRXS	Rd, Rm	Rotate Right with Extend, Rd ← RRX(Rm), RRXS updates N,Z,C
RSB, RSBS	{Rd,} Rn, Op2	Reverse Subtract, Rd ← Op2 – Rn, RSBS updates N,Z,C,V
SBC, SBCS	{Rd,} Rn, Op2	Subtract with Carry, Rd ← Rn-Op2-NOT(Carry), updates NZCV
SBFX	Rd, Rn, #lsb, #width	Signed Bit Field Extract, Rd[(width-1):0] = Rn[(width+lsb-1):lsb], Rd[31:width] = Replicate(Rn[width+lsb-1])
SDIV	{Rd,} Rn, Rm	Signed Divide, Rd ← Rn/Rm
SEV	-	Send Event
SMLAL	RdLo, RdHi, Rn, Rm	Signed Multiply with Accumulate, RdHi,RdLo ← signed(RdHi,RdLo + Rn*Rm)
SMULL	RdLo, RdHi, Rn, Rm	Signed Multiply, RdHi,RdLo ← signed(Rn*Rm)
SSAT	Rd, #n, Rm{,shift #s}	Signed Saturate, Rd ← SignedSat((Rm shift s), n). Update Q
STM	Rn{!}, reglist	Store Multiple Registers
STMDB, STMEA	Rn{!}, reglist	Store Multiple Registers Decrement Before
STMFD, STMIA	Rn{!}, reglist	Store Multiple Registers Increment After
STR	Rt, [Rn, #offset]	Store Register with Word, mem[Rn+offset] = Rt
STRB, STRBT	Rt, [Rn, #offset]	Store Register with Byte, mem[Rn+offset] = Rt
STRD	Rt, Rt, [Rn,#offset]	Store Register with two Words, mem[Rn+offset] = Rt, mem[Rn+offset+4] = Rt2
STREX	Rd, Rt, [Rn,#offset]	Store Register Exclusive, If allowed, mem[Rn + offset] ← Rt, clear exclusive tag, Rd ← 0. Else Rd ← 1.
STREXB	Rd, Rt, [Rn]	Store Register Exclusive Byte, mem[Rn] ← Rt[15:0] or mem[Rn] ← Rt[7:0], clear exclusive tag, Rd ← 0. Else Rd ← 1
STREXH	Rd, Rt, [Rn]	Store Register Exclusive Halfword, mem[Rn] ← Rt[15:0] or mem[Rn] ← Rt[7:0], clear exclusive tag, Rd ← 0. Else Rd ← 1
STRH, STRHT	Rt, [Rn, #offset]	Store Halfword, mem[Rn + offset] ← Rt[15:0]
STRT	Rt, [Rn, #offset]	Store Register with Translation, mem[Rn + offset] = Rt
SUB, SUBS	{Rd,} Rn, Op2	Subtraction, Rd ← Rn – Op2, SUBS updates N,Z,C,V
SUB, SUBS	{Rd,} Rn, #imm12	Subtraction, Rd ← Rn-imm12, SUBS updates N,Z,C,V
SVC	#imm	Supervisor Call
SXTB	{Rd,} Rm {,ROR #n}	Sign Extend Byte, Rd ← SignExtend((Rm ROR (8*n))[7:0])
SXTH	{Rd,} Rm {,ROR #n}	Sign Extend Halfword, Rd ← SignExtend((Rm ROR (8*n))[15:0])
TBB	[Rn, Rm]	Table Branch Byte, PC ← PC+ZeroExtend(Memory(Rn+Rm,1)<<1)
TBH	[Rn, Rm, LSL #1]	Table Branch Halfword, PC ← PC + ZeroExtend(Memory(Rn+Rm<<1, 2)<<1)
TEQ	Rn, Op2	Test Equivalence, Update N,Z,C,V on Rn EOR Operand2
TST	Rn, Op2	Test, Update N,Z,C,V on Rn AND Op2
UBFX	Rd, Rn, #lsb, #width	Unsigned Bit Field Extract, Rd[(width-1):0] = Rn[(width+lsb-1):lsb], Rd[31:width] = Replicate(0)
UDIV	{Rd,} Rn, Rm	Unsigned Divide, Rd ← Rn/Rm
UMLAL	RdLo, RdHi, Rn, Rm	Unsigned Multiply with Accumulate, RdHi,RdLo ← unsigned(RdHi,RdLo + Rn*Rm)
UMULL	RdLo, RdHi, Rn, Rm	Unsigned Multiply, RdHi,RdLo ← unsigned(Rn*Rm)
USAT	Rd, #n, Rm{,shift #s}	Unsigned Saturate, Rd←UnsignedSat((Rm shift s),n), Update Q
UXTB	{Rd,} Rm {,ROR #n}	Unsigned Extend Byte, Rd ← ZeroExtend((Rm ROR (8*n))[7:0])
UXTH	{Rd,} Rm {,ROR #n}	Unsigned Extend Halfword, Rd ← ZeroExtend((Rm ROR (8*n))[15:0])
WFE	-	Wait For Event
WFI	-	Wait for Interrupt

Instruction	Operands	Description and Action
ADC, ADCS	{Rd,} Rn, Op2	Add with Carry, Rd ← Rn + Op2 + Carry, ADCS updates N,Z,C,V
ADD, ADDS	{Rd,} Rn, Op2	Add, Rd ← Rn + Op2, ADDS updates N,Z,C,V
ADD, ADDS	{Rd,} Rn, #imm12	Add Immediate, Rd ← Rn + imm12, ADDS updates N,Z,C,V
ADR	Rd, label	Load PC-relative Address, Rd ← <label>
AND, ANDS	{Rd,} Rn, Op2	Logical AND, Rd ← Rn AND Op2, ANDS updates N,Z,C
ASR, ASRS	Rd, Rm, <Rs|#n>	Arithmetic Shift Right, Rd ← Rm>>(Rs|n), ASRS updates N,Z,C
B	label	Branch, PC ← label
BFC	Rd, #lsb, #width	Bit Field Clear, Rd[(width+lsb-1):lsb] ← 0
BFI	Rd, Rn, #lsb, #width	Bit Field Insert, Rd[(width+lsb-1):lsb] ← Rn[(width-1):0]
BIC, BICS	{Rd,} Rn, Op2	Bit Clear, Rd ← Rn AND NOT Op2, BICS updates N,Z,C
BKPT	#imm	Breakpoint, prefetch abort or enter debug state
BL	label	Branch with Link, LR ← address of next instruction, PC ← label
BLX	Rm	Branch register with link, LR ← address of next instruction, PC ← Rm[31:1]
BX	Rm	Branch register, PC ← Rm
CBNZ	Rn, label	Compare and Branch if Non-zero; PC ← label if Rn != 0
CBZ	Rn, label	Compare and Branch if Zero; PC ← label if Rn == 0
CLREX	-	Clear local processor exclusive tag
CLZ	Rd, Rm	Count Leading Zeroes, Rd ← number of leading zeroes in Rm
CMN	Rn, Op2	Compare Negative, Update N,Z,C,V flags on Rn + Op2
CMP	Rn, Op2	Compare, Update N,Z,C,V flags on Rn − Op2
CPSID	i	Disable specified (i) interrupts, optional change mode
CPSIE	i	Enable specified (i) interrupts, optional change mode
DMB	-	Data Memory Barrier, ensure memory access order
DSB	-	Data Synchronization Barrier, ensure completion of access
EOR, EORS	{Rd,} Rn, Op2	Exclusive OR, Rd ← Rn XOR Op2, EORS updates N,Z,C
ISB	-	Instruction Synchronization Barrier
IT	-	If-Then Condition Block
LDM	Rn{!}, reglist	Load Multiple Registers increment after, <reglist> = mem[Rn], Rn increments after each memory access
LDMDB, LDMEA	Rn{!}, reglist	Load Multiple Registers Decrement Before, <reglist> = mem[Rn], Rn decrements before each memory access
LDMFD, LDMIA	Rn{!}, reglist	<reglist> = mem[Rn], Rn increments after each memory access
LDR	Rt, [Rn, #offset]	Load Register with Word, Rt ← mem[Rn + offset]
LDRB, LDRBT	Rt, [Rn, #offset]	Load Register with Byte, Rt ← mem[Rn + offset]
LDRD	Rt, Rt2, [Rn,#offset]	Load Register with two words, Rt ← mem[Rn + offset], Rt2 ← mem[Rn + offset + 4]
LDREX	Rt, [Rn, #offset]	Load Register Exclusive, Rt ← mem[Rn + offset]
LDREXB	Rt, [Rn]	Load Register Exclusive with Byte, Rt ← mem[Rn]
LDREXH	Rt, [Rn]	Load Register Exclusive with Halfword, Rt ← mem[Rn]
LDRH, LDRHT	Rt, [Rn, #offset]	Load Register with Halfword, Rt ← mem[Rn + offset]
LDRSB, LDRSBT	Rt, [Rn, #offset]	Load Register with Signed Byte, Rt ← mem[Rn + offset]
LDRSH, LDRSHT	Rt, [Rn, #offset]	Load Register with Signed Halfword, Rt ← mem[Rn + offset]
LDRT	Rt, [Rn, #offset]	Load Register with Word, Rt ← mem[Rn + offset]
LSL, LSLS	Rd, Rm, <Rs|#n>	Logic Shift Left, Rd ← Rm << Rs|n, LSLS update N,Z,C
LSR, LSRS	Rd, Rm, <Rs|#n>	Logic Shift Right, Rd ← Rm >> Rs|n, LSRS update N,Z,C
MLA	Rd, Rn, Rm, Ra	Multiply with Accumulate, Rd ← (Ra + (Rn*Rm))[31:0]
MLS	Rd, Rn, Rm, Ra	Multiply with Subtract, Rd ← (Ra - (Rn*Rm))[31:0]
MOV, MOVS	Rd, Op2	Move, Rd ← Op2, MOVS updates N,Z,C
MOVT	Rd, #imm16	Move Top, Rd[31:16] ← imm16, Rd[15:0] unaffected
MOVW, MOVWS	Rd, #imm16	Move 16-bit Constant, Rd ← imm16, MOVWS updates N,Z,C
MRS	Rd, spec_reg	Move from Special Register, Rd ← spec_reg
MSR	spec_reg, Rm	Move to Special Register, spec_reg ← Rm, Updates N,Z,C,V
MUL, MULS	{Rd,} Rn, Rm	Multiply, Rd ← (Rn*Rm)[31:0], MULS updates N,Z
MVN, MVNS	Rd, Op2	Move NOT, Rd ← 0xFFFFFFFF EOR Op2, MVNS updates N,Z,C
NOP	-	No Operation
ORN, ORNS	{Rd,} Rn, Op2	Logical OR NOT, Rd ← Rn OR NOT Op2, ORNS updates N,Z,C
ORR, ORRS	{Rd,} Rn, Op2	Logical OR, Rd ← Rn OR Op2, ORRS updates N,Z,C
POP	reglist	Canonical form of LDM SP!, <reglist>
PUSH	reglist	Canonical form of STMDB SP!, <reglist>
RBIT	Rd, Rn	Reverse Bits, for (i = 0; i < 32; i++): Rd[i] = RN[31-i]
REV	Rd, Rn	Reverse Byte Order in a Word, Rd[31:24]←Rn[7:0], Rd[23:16]←Rn[15:8], Rd[15:8]←Rn[23:16], Rd[7:0]←Rn[31:24]

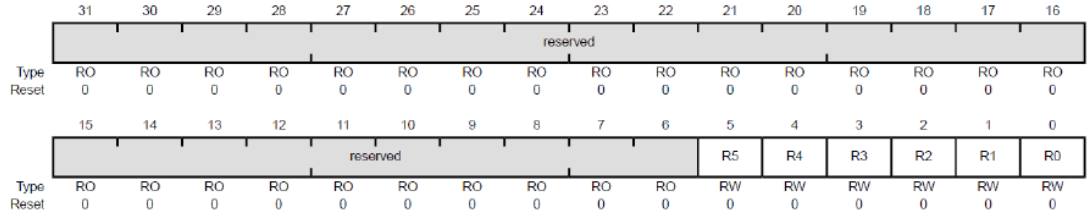
A3. TM4C123 Register Set

GPIO

GPIO pins and alternate functions

IO	Pin	Analog Function	Digital Function (GPIOPCTL PMCx Bit Field Encoding)[a]										
			1	2	3	4	5	6	7	8	9	14	15
PA0	17	-	U0Rx	-	-	-	-	-	-	CAN1Rx	-	-	-
PA1	18	-	U0Tx	-	-	-	-	-	-	CAN1Tx	-	-	-
PA2	19	-	-	SSI0Clk	-	-	-	-	-	-	-	-	-
PA3	20	-	-	SSI0Fss	-	-	-	-	-	-	-	-	-
PA4	21	-	-	SSI0Rx	-	-	-	-	-	-	-	-	-
PA5	22	-	-	SSI0Tx	-	-	-	-	-	-	-	-	-
PA6	23	-	-	-	I2C1SCL	-	M1PWM2	-	-	-	-	-	-
PA7	24	-	-	-	I2C1SDA	-	M1PWM3	-	-	-	-	-	-
PB0	45	USB0ID	U1Rx	-	-	-	-	-	T2CCP0	-	-	-	-
PB1	46	USB0VBUS	U1Tx	-	-	-	-	-	T2CCP1	-	-	-	-
PB2	47	-	-	-	I2C0SCL	-	-	-	T3CCP0	-	-	-	-
PB3	48	-	-	-	I2C0SDA	-	-	-	T3CCP1	-	-	-	-
PB4	58	AIN10	-	SSI2Clk	-	M0PWM2	-	-	T1CCP0	CAN0Rx	-	-	-
PB5	57	AIN11	-	SSI2Fss	-	M0PWM3	-	-	T1CCP1	CAN0Tx	-	-	-
PB6	1	-	-	SSI2Rx	-	M0PWM0	-	-	T0CCP0	-	-	-	-
PB7	4	-	-	SSI2Tx	-	M0PWM1	-	-	T0CCP1	-	-	-	-
PC0	52	-	TCK SWCLK	-	-	-	-	-	T4CCP0	-	-	-	-
PC1	51	-	TMS SWDIO	-	-	-	-	-	T4CCP1	-	-	-	-
PC2	50	-	TDI	-	-	-	-	-	T5CCP0	-	-	-	-
PC3	49	-	TDO SWO	-	-	-	-	-	T5CCP1	-	-	-	-
PC4	16	C1-	U4Rx	U1Rx	-	M0PWM6	-	IDX1	WT0CCP0	U1RTS	-	-	-
PC5	15	C1+	U4Tx	U1Tx	-	M0PWM7	-	PhA1	WT0CCP1	U1CTS	-	-	-
PC6	14	C0+	U3Rx	-	-	-	-	PhB1	WT1CCP0	USB0EPEN	-	-	-
PC7	13	C0-	U3Tx	-	-	-	-	-	WT1CCP1	USB0PFLT	-	-	-
PD0	61	AIN7	SSI3Clk	SSI1Clk	I2C3SCL	M0PWM6	M1PWM0	-	WT2CCP0	-	-	-	-
PD1	62	AIN6	SSI3Fss	SSI1Fss	I2C3SDA	M0PWM7	M1PWM1	-	WT2CCP1	-	-	-	-
PD2	63	AIN5	SSI3Rx	SSI1Rx	-	M0FAULT0	-	-	WT3CCP0	USB0EPEN	-	-	-
PD3	64	AIN4	SSI3Tx	SSI1Tx	-	-	-	IDX0	WT3CCP1	USB0PFLT	-	-	-
PD4	43	USB0DM	U6Rx	-	-	-	-	-	WT4CCP0	-	-	-	-
PD5	44	USB0DP	U6Tx	-	-	-	-	-	WT4CCP1	-	-	-	-
PD6	53	-	U2Rx	-	-	M0FAULT0	-	PhA0	WT5CCP0	-	-	-	-
PD7	10	-	U2Tx	-	-	-	-	PhB0	WT5CCP1	NMI	-	-	-
PE0	9	AIN3	U7Rx	-	-	-	-	-	-	-	-	-	-
PE1	8	AIN2	U7Tx	-	-	-	-	-	-	-	-	-	-
PE2	7	AIN1	-	-	-	-	-	-	-	-	-	-	-
PE3	6	AIN0	-	-	-	-	-	-	-	-	-	-	-
PE4	59	AIN9	U5Rx	-	I2C2SCL	M0PWM4	M1PWM2	-	-	CAN0Rx	-	-	-
PE5	60	AIN8	U5Tx	-	I2C2SDA	M0PWM5	M1PWM3	-	-	CAN0Tx	-	-	-
PF0	28	-	U1RTS	SSI1Rx	CAN0Rx	-	M1PWM4	PhA0	T0CCP0	NMI	C0o	-	-
PF1	29	-	U1CTS	SSI1Tx	-	-	M1PWM5	PhB0	T0CCP1	-	C1o	TRD1	-
PF2	30	-	-	SSI1Clk	-	M0FAULT0	M1PWM6	-	T1CCP0	-	-	TRD0	-
PF3	31	-	-	SSI1Fss	CAN0Tx	-	M1PWM7	-	T1CCP1	-	-	TRCLK	-
PF4	5	-	-	-	-	-	M1FAULT0	IDX0	T2CCP0	USB0EPEN	-	-	-

Run Clock Gate Control Register for GPIO (RCGCGPIO) (address is 0x400FE608)



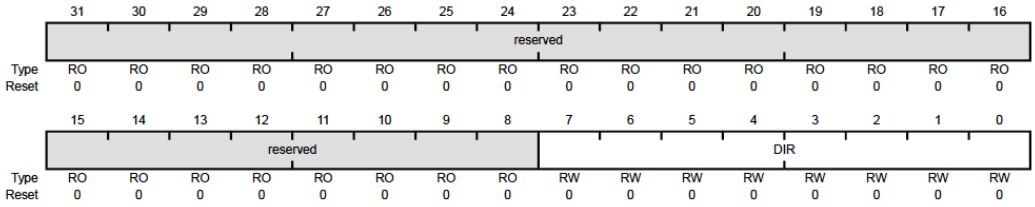
Bit/Field	Name	Type	Reset	Description
5	R5	RW	0	GPIO Port F Run Mode Clock Gating Control Value Description 0 GPIO Port F is disabled. 1 Enable and provide a clock to GPIO Port F in Run mode.
. . .				
0	R0	RW	0	GPIO Port A Run Mode Clock Gating Control Value Description 0 GPIO Port A is disabled. 1 Enable and provide a clock to GPIO Port A in Run mode.

Base addresses for GPIO ports

Port A: 0x40004000
Port B: 0x40005000
Port C: 0x40006000
Port D: 0x40007000
Port E: 0x40024000
Port F: 0x40025000

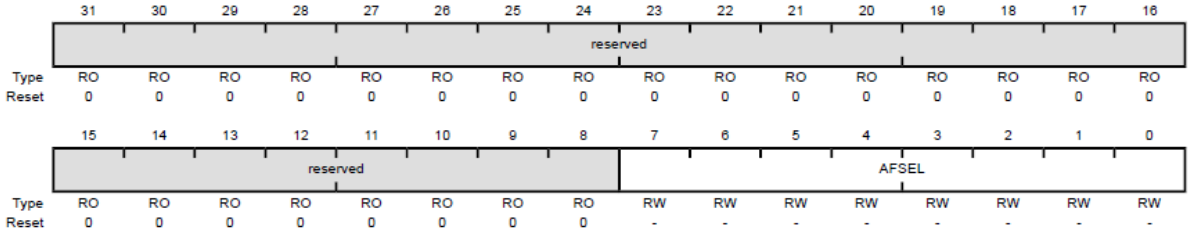
Data Register: GPIODATA, offset 0x000

Data direction register: GPIODIR, offset 0x400



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	DIR	RW	0x00	GPIO Data Direction Value Description 0 Corresponding pin is an input. 1 Corresponding pins is an output.

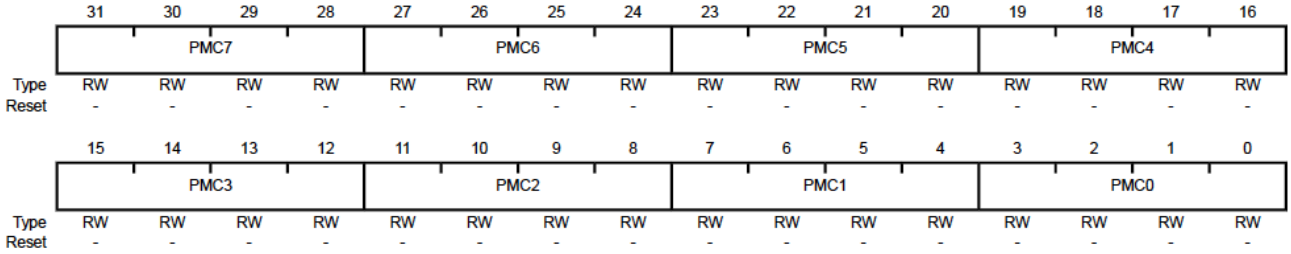
GPIO Alternate Function Select: GPIOAFSEL, offset 0x420



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	AFSEL	RW	-	GPIO Alternate Function Select

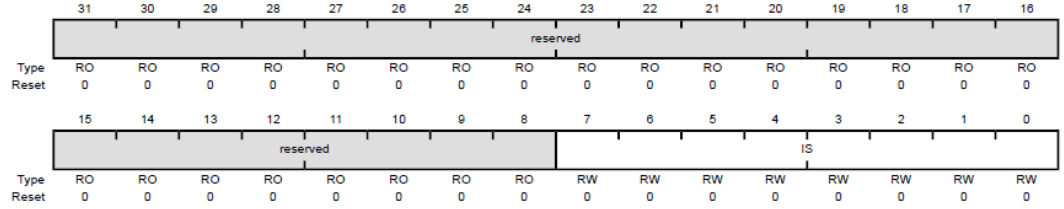
Value	Description
0	The associated pin functions as a GPIO and is controlled by the GPIO registers.
1	The associated pin functions as a peripheral signal and is controlled by the alternate hardware function.

GPIO Port Control: GPIOPCTL, offset 0x52C



Bit/Field	Name	Type	Reset	Description
31:28	PMC7	RW	-	Port Mux Control 7 This field controls the configuration for GPIO pin 7.
27:24	PMC6	RW	-	Port Mux Control 6 This field controls the configuration for GPIO pin 6.
23:20	PMC5	RW	-	Port Mux Control 5 This field controls the configuration for GPIO pin 5.
19:16	PMC4	RW	-	Port Mux Control 4 This field controls the configuration for GPIO pin 4.
15:12	PMC3	RW	-	Port Mux Control 3 This field controls the configuration for GPIO pin 3.
11:8	PMC2	RW	-	Port Mux Control 2 This field controls the configuration for GPIO pin 2.
7:4	PMC1	RW	-	Port Mux Control 1 This field controls the configuration for GPIO pin 1.
3:0	PMC0	RW	-	Port Mux Control 0 This field controls the configuration for GPIO pin 0.

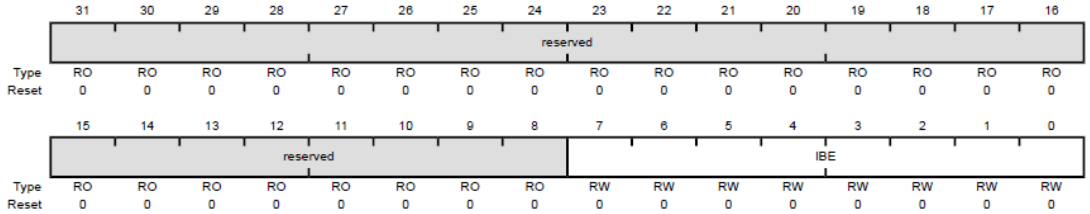
GPIO Interrupt Sense (GPIOIS), offset 0x404



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IS	RW	0x00	GPIO Interrupt Sense

Value	Description
0	The edge on the corresponding pin is detected (edge-sensitive).
1	The level on the corresponding pin is detected (level-sensitive).

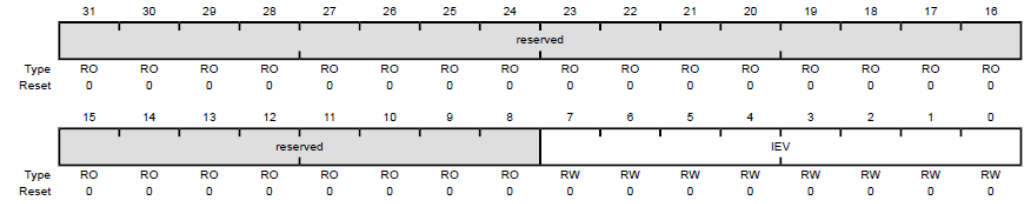
GPIO Interrupt Both Edges:GPIOIBE, offset 0x408



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IBE	RW	0x00	GPIO Interrupt Both Edges

Value	Description
0	Interrupt generation is controlled by the **GPIO Interrupt Event (GPIOIEV)** register (see page 666).
1	Both edges on the corresponding pin trigger an interrupt.

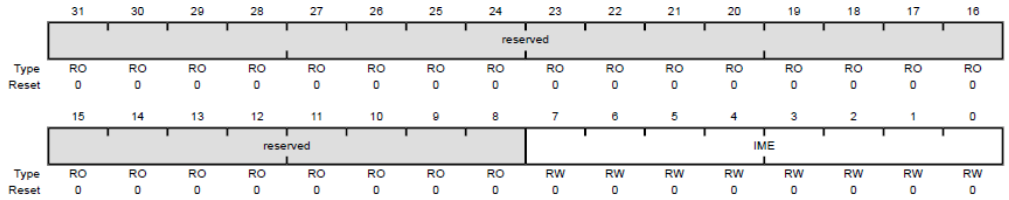
GPIO Interrupt Event : GPIOIEV, offset 0x40C



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IEV	RW	0x00	GPIO Interrupt Event

Value	Description
0	A falling edge or a Low level on the corresponding pin triggers an interrupt.
1	A rising edge or a High level on the corresponding pin triggers an interrupt.

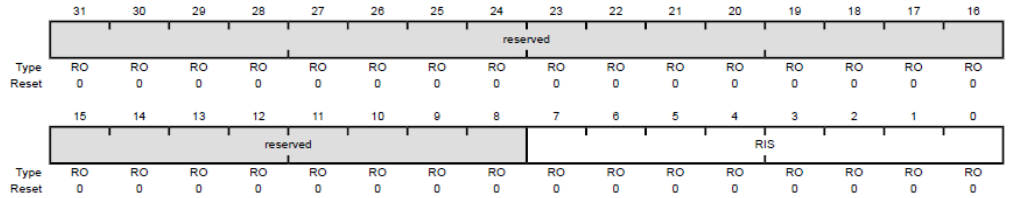
GPIO Interrupt Mask: GPIOIM, offset 0x410



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IME	RW	0x00	GPIO Interrupt Mask Enable

Value	Description
0	The interrupt from the corresponding pin is masked.
1	The interrupt from the corresponding pin is sent to the interrupt controller.

GPIO Raw Interrupt Status: GPIORIS, offset 0x414 (read only)



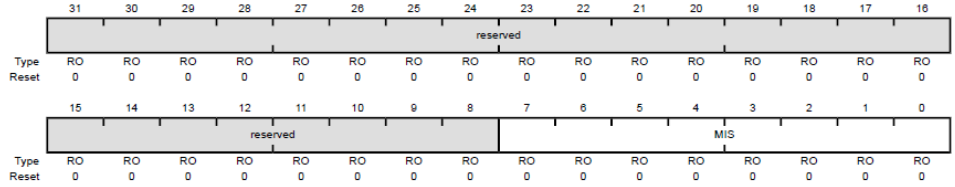
Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	RIS	RO	0x00	GPIO Interrupt Raw Status

Value	Description
0	An interrupt condition has not occurred on the corresponding pin.
1	An interrupt condition has occurred on the corresponding pin.

For edge-detect interrupts, this bit is cleared by writing a 1 to the corresponding bit in the **GPIOICR** register.

For a GPIO level-detect interrupt, the bit is cleared when the level is deasserted.

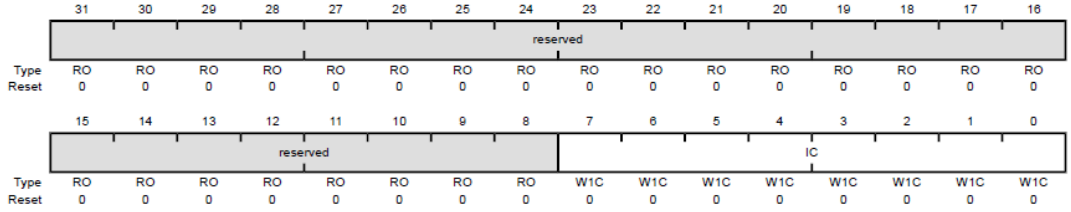
GPIO Masked Interrupt Status: GPIOMIS, offset 0x418 (read only)



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	MIS	RO	0x00	GPIO Masked Interrupt Status

Value	Description
0	An interrupt condition on the corresponding pin is masked or has not occurred.
1	An interrupt condition on the corresponding pin has triggered an interrupt to the interrupt controller.

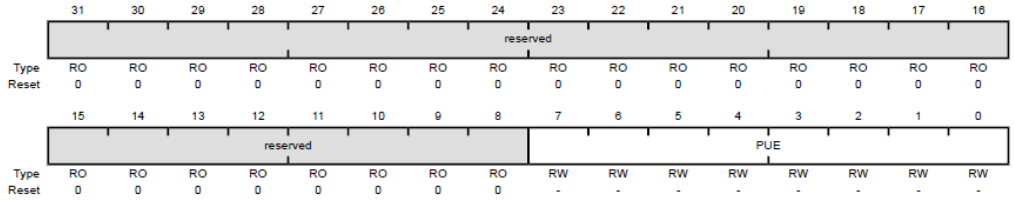
GPIO Interrupt Clear: GPIOICR, offset 0x41C



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IC	W1C	0x00	GPIO Interrupt Clear

Value	Description
0	The corresponding interrupt is unaffected.
1	The corresponding interrupt is cleared.

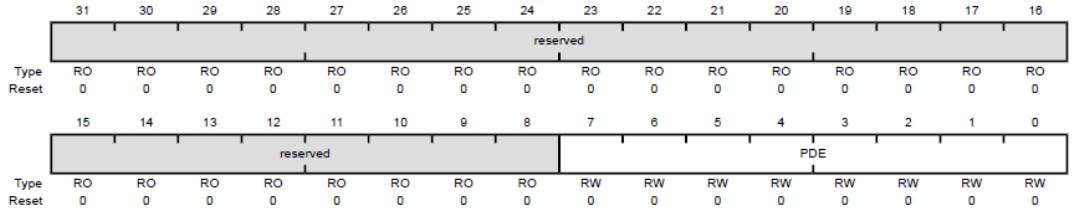
GPIO Pull-Up Select: GPIOPUR, offset 0x510



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	PUE	RW	-	Pad Weak Pull-Up Enable

Value	Description
0	The corresponding pin's weak pull-up resistor is disabled.
1	The corresponding pin's weak pull-up resistor is enabled.

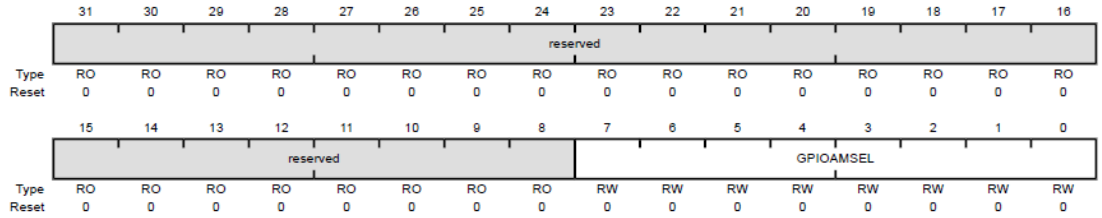
GPIO Pull-Down Select (GPIOPDR), offset 0x514



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	PDE	RW	0x00	Pad Weak Pull-Down Enable

Value	Description
0	The corresponding pin's weak pull-down resistor is disabled.
1	The corresponding pin's weak pull-down resistor is enabled.

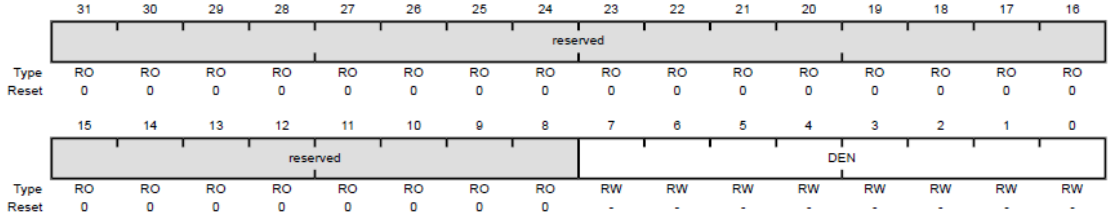
GPIO Analog Mode Select: GPIOAMSEL, offset 0x528



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	GPIOAMSEL	RW	0x00	GPIO Analog Mode Select

Value	Description
0	The analog function of the pin is disabled, the isolation is enabled, and the pin is capable of digital functions as specified by the other GPIO configuration registers.
1	The analog function of the pin is enabled, the isolation is disabled, and the pin is capable of analog functions.

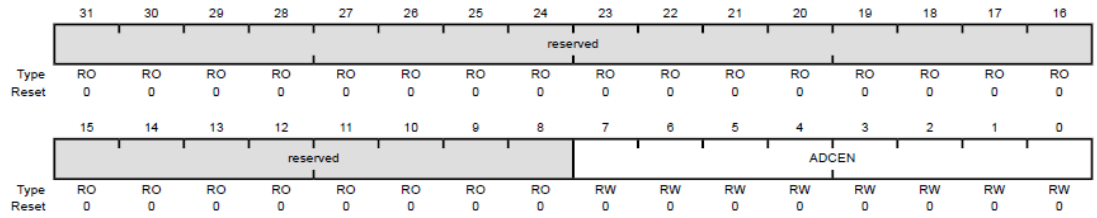
GPIO Digital Enable: GPIODEN, offset 0x51C



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	DEN	RW	-	Digital Enable

Value	Description
0	The digital functions for the corresponding pin are disabled.
1	The digital functions for the corresponding pin are enabled.

GPIO ADC Control: GPIOADCCTL, offset 0x530



Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	ADCEN	RW	0x00	ADC Trigger Enable

Value	Description
0	The corresponding pin is not used to trigger the ADC.
1	The corresponding pin is used to trigger the ADC.

TIMER

SysTick

Address	31-24	23-17	16	15-3	2	1	0	Name
$E000E010	0	0	COUNT	0	CLK_SRC	INTEN	ENABLE	STCTRL
$E000E014	0	24-bit RELOAD value						STRELOAD
$E000E018	0	24-bit CURRENT value of SysTick counter						STCURRENT

SysTick Control and Status Register(STCTRL)

Enable: 0 timer disabled, 1 timer enabled

INTEN: 0 interrupt disabled, 1 interrupt is generated to the NVIC when SysTick counts to 0.

CLK_SRC: 0 (POSC) divided by 4, 1 system clock

COUNT: 0 SysTick timer has not counted to 0 yet, 1 SysTick timer has counted to 0. This bit is cleared by a read of the register or if the STCURRENT register is written with any value.

SysTick Reload Value Register (STRELOAD)

This register specifies the start value to load into the SysTick Current Value (STCURRENT) register when the counter reaches 0.

Start value can be between 0x1 and 0x00FFFFFF (only 24 bit value an be written)

SysTick Current Value Register (STCURRENT)

This register is write-clear. Writing to it with any value clears the register. Clearing this register also clears the COUNT bit of the STCTRL register.

General Purpose Timers

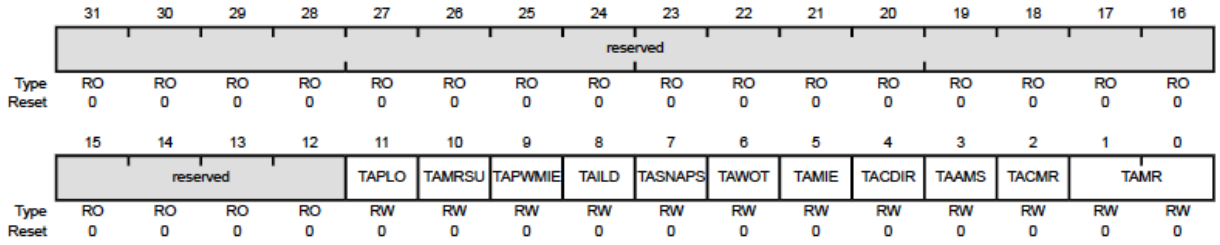
Base addresses of six 16/32 bit Timers and six 32/64 bit Timers:

- 16/32-bit Timer 0: 0x4003.0000
- 16/32-bit Timer 1: 0x4003.1000
- 16/32-bit Timer 2: 0x4003.2000
- 16/32-bit Timer 3: 0x4003.3000
- 16/32-bit Timer 4: 0x4003.4000
- 16/32-bit Timer 5: 0x4003.5000
- 32/64-bit Wide Timer 0: 0x4003.6000
- 32/64-bit Wide Timer 1: 0x4003.7000
- 32/64-bit Wide Timer 2: 0x4004.C000
- 32/64-bit Wide Timer 3: 0x4004.D000
- 32/64-bit Wide Timer 4: 0x4004.E000
- 32/64-bit Wide Timer 5: 0x4004.F000

Timer Configuration : CFG, offset 0x000

Value of Bit field 2:0	Description
0x0	For a 16/32-bit timer, this value selects the 32-bit timer configuration. For a 32/64-bit wide timer, this value selects the 64-bit timer configuration.
0x1	For a 16/32-bit timer, this value selects the 32-bit real-time clock (RTC) counter configuration. For a 32/64-bit wide timer, this value selects the 64-bit real-time clock (RTC) counter configuration.
0x4	For a 16/32-bit timer, this value selects the 16-bit timer configuration. For a 32/64-bit wide timer, this value selects the 32-bit timer configuration.

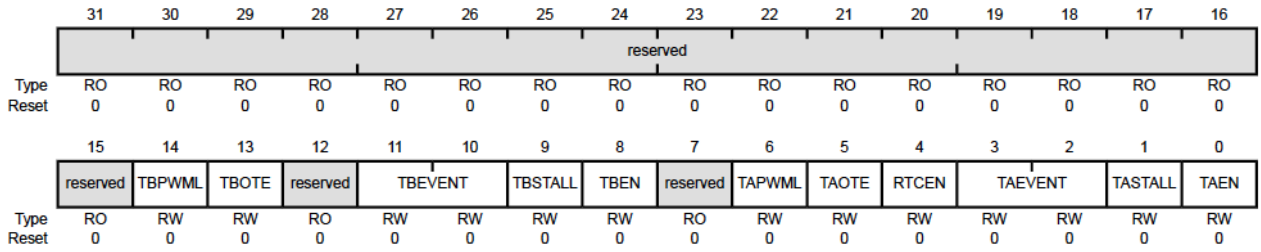
Timer A/B Mode : TAMR/ TBMR, offset 0x004/0x008



Bit field	Value	Description
1:0	0x1	One shot timer mode
	0x2	Periodic timer mode
	0x3	Capture mode
2	0	Edge count mode
	1	Edge time mode
3	0	Capture or compare mode

	1	PWM
4	0	Timer counts down
	1	Timer counts up
5	0	The match interrupt is disabled for match events.
	1	An interrupt is generated when the match value in the TAMATCHR register is reached in the one-shot and periodic modes.
6	0	Timer A begins counting as soon as it is enabled
	1	If Timer A is enabled (TAEN is set in the CTL register), Timer A does not begin counting until it receives a trigger from the timer in the previous position in the daisy chain
7	0	Snap shot mode is disabled
	1	If Timer A is configured in the periodic mode, the actual free-running, capture or snapshot value of Timer A is loaded at the time-out event/capture or snapshot event into the Timer A (TAR) register. If the timer prescaler is used, the prescaler snapshot is loaded into the Timer A (TAPR).
8	0	Update the TAR and TAV registers with the value in the TAILR register on the next cycle. Also update the TAPS and TAPV registers with the value in the TAPR register on the next cycle.
	1	Update the TAR and TAV registers with the value in the TAILR register on the next timeout. Also update the TAPS and TAPV registers with the value in the TAPR register on the next timeout.
9	0	Capture event interrupt is disabled.
	1	Capture event interrupt is enabled.
10	0	Update the TAMATCHR register and the TAPR register, if used, on the next cycle.
	1	Update the TAMATCHR register and the TAPR register, if used, on the next time out.
11	0	Legacy operation with CCP pin driven Low when the TAILR is reloaded after the timer reaches 0.
	1	CCP is driven High when the TAILR is reloaded after the timer reaches 0.

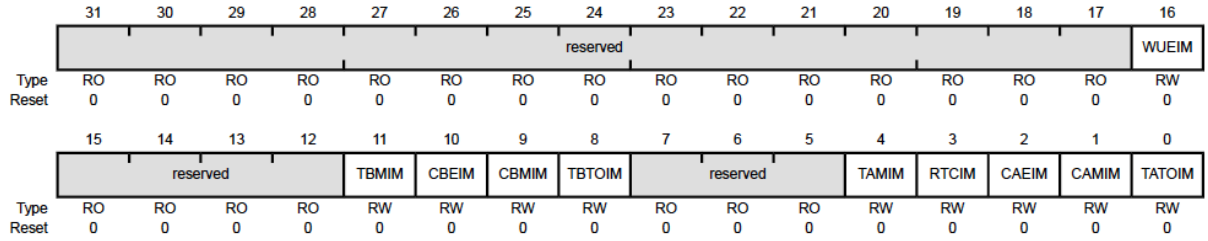
Timer Control : CTL, offset 0x00C



Bit field	Value	Description
14	0	Output is unaffected.
	1	Output is inverted.
13	0	The output Timer B ADC trigger is disabled.
	1	The output Timer B ADC trigger is enabled.

11:10	0x0	Timer B Event Mode, Positive edge
	0x1	Timer B Event Mode, Negative edge
	0x3	Timer B Event Mode, both edges
9	0	Timer B continues counting while the processor is halted by the debugger.
	1	Timer B freezes counting while the processor is halted by the debugger.
8	0	Timer B is disabled.
	1	Timer B is enabled and begins counting or the capture logic is enabled based on the CFG register.
6	0	Timer A PWM Output Level is unaffected.
	1	Timer A PWM Output Level is inverted.
5	0	The output Timer A ADC trigger is disabled.
	1	The output Timer A ADC trigger is enabled.
4	0	RTC counting freezes while the processor is halted by the debugger.
	1	RTC counting continues while the processor is halted by the debugger.
3:2	0x0	Timer A Event Mode, Positive edge
	0x1	Timer A Event Mode, Negative edge
	0x3	Timer A Event Mode, both edges
1	0	Timer A continues counting while the processor is halted by the debugger.
	1	Timer A freezes counting while the processor is halted by the debugger.
0	0	Timer A is disabled.
	1	Timer A is enabled and begins counting or the capture logic is enabled based on the CFG register.

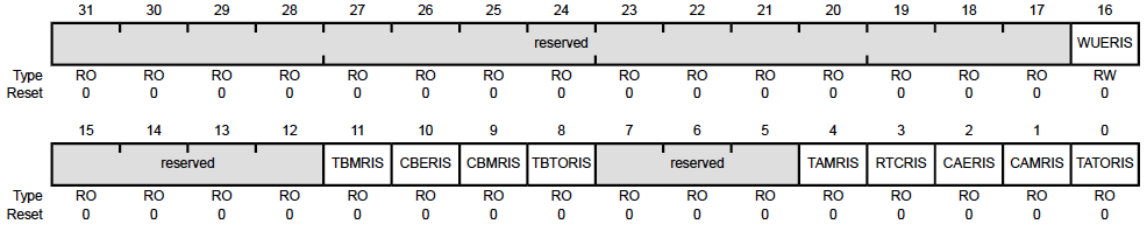
Timer Interrupt Mask: IMR, offset 0x018



bit	function
16	32/64-Bit Wide Write Update (WUE) Error Interrupt Mask
11	Timer B Match (TBM) Interrupt Mask
10	Timer B Capture Mode Event (CBE) Interrupt Mask
9	Timer B Capture Mode Match (CBM) Interrupt Mask
8	Timer B Time-Out (TBTO) Interrupt Mask
4	Timer A Match (TAM) Interrupt Mask
3	RTC Interrupt Mask
2	Timer A Capture Mode Event (CAE) Interrupt Mask
1	Timer A Capture Mode Match (CAM) Interrupt Mask
0	Timer A Time-Out (TATO) Interrupt Mask

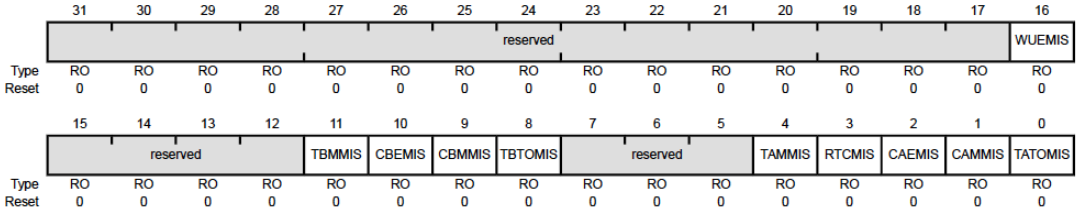
0: Interrupt is disabled, 1: Interrupt is enabled

Timer Raw Interrupt Status : RIS, offset 0x01C



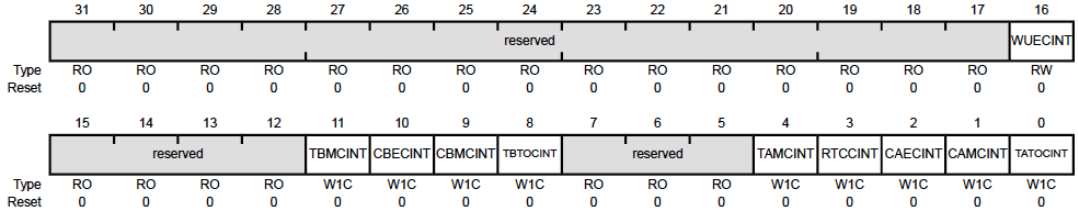
0: Match/capture/time-out etc. events not occurred.
1: Match/capture/time-out etc. events occurred.
(TBM, CBE, etc. corresponds to the same abbreviations as the above table)

Timer Masked Interrupt Status : MIS, offset 0x020



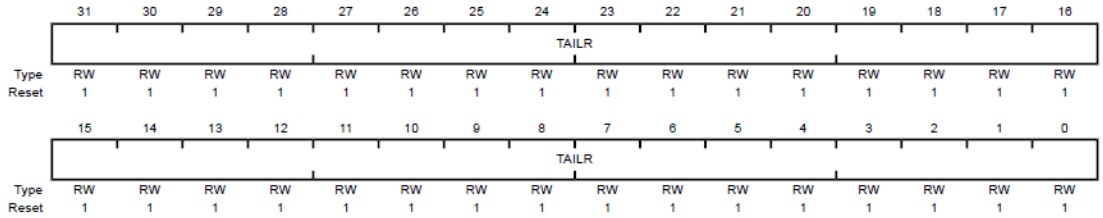
0: Match/capture/time-out etc. interrupts has not occurred or masked.
1: Unmasked Match/capture/time-out etc. interrupt has occurred.
(TBM, CBE, etc. corresponds to the same abbreviations as the above table)

Timer Interrupt Clear: ICR, offset 0x024



Writing a 1 to this a bit clears the respective bit in RIS and MRIS registers.

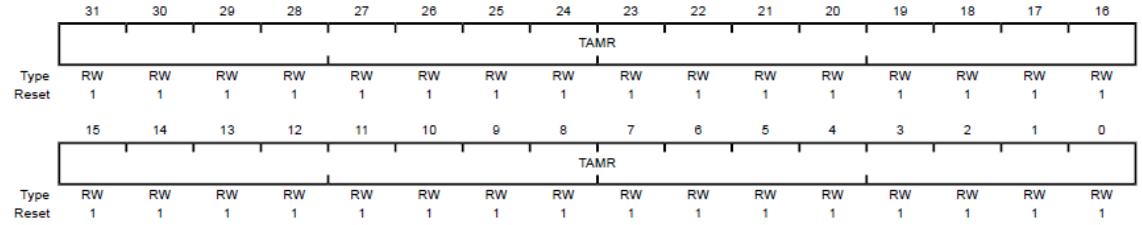
Timer A/B Interval Load: TAILR/ TBILR, offset 0x028/0x02C



Bit/Field	Name	Type	Reset	Description
31:0	TAILR	RW	0xFFFF.FFFF	GPTM Timer A Interval Load Register Writing this field loads the counter for Timer A.

(Same for Timer B)

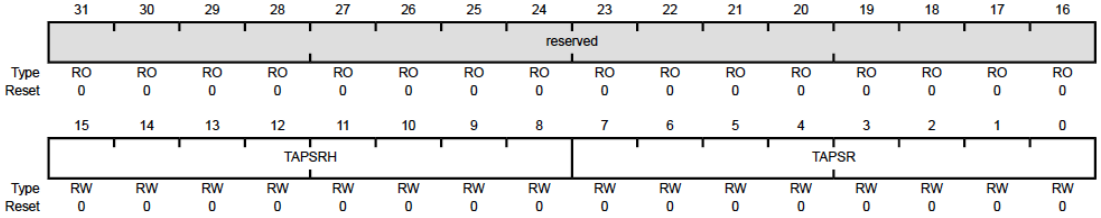
Timer A/B Match: TAMATCHR/ TBMATCHR, offset 0x030/0x034



Bit/Field	Name	Type	Reset	Description
31:0	TAMR	RW	0xFFFF.FFFF	GPTM Timer A Match Register This value is compared to the **GPTMTAR** register to determine match events.

(Same for Timer B)

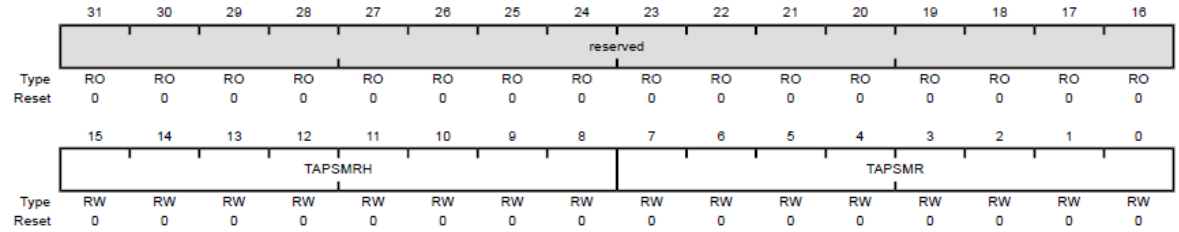
Timer A/B Prescale: TAPR/ TBPR, offset 0x038/0x03C



TAPSRH: GPTM Timer A Prescale High Byte. For the 16/32-bit GPTM, this field is reserved. For the 32/64-bit Wide GPTM, this field contains the upper 8-bits of the 16-bit prescaler.

TAPSRL: GPTM Timer A Prescale Low Byte. For the 16/32-bit GPTM, this field contains the entire 8-bit prescaler. For the 32/64-bit Wide GPTM, this field contains the lower 8-bits of the 16-bit prescaler.
(Same for Timer B)

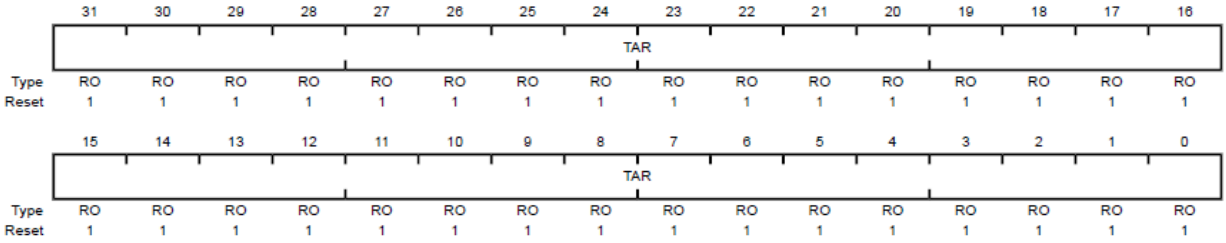
Timer A/B Prescale Match: TAPMR/TBPMR, offset 0x040/0x044



TAPSRH: GPTM Timer A Match Prescale High Byte. For the 16/32-bit GPTM, this field is reserved. For the 32/64-bit Wide GPTM, this field contains the upper 8-bits of the 16-bit match prescaler.

TAPSRL: GPTM Timer A Match Prescale Low Byte. For the 16/32-bit GPTM, this field contains the entire 8-bit match prescaler. For the 32/64-bit Wide GPTM, this field contains the lower 8-bits of the 16-bit match prescaler.
(Same for Timer B)

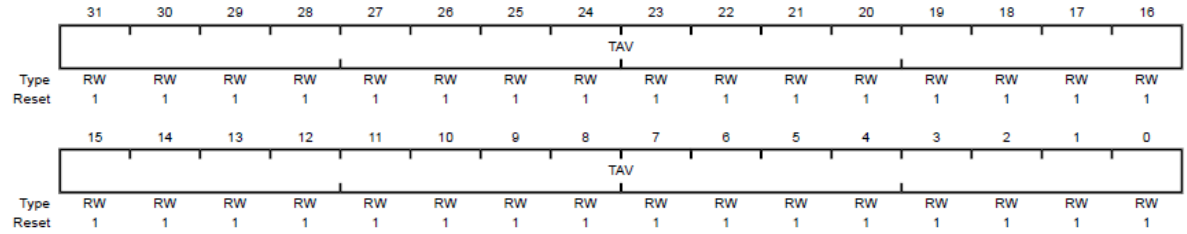
Timer A/B: TAR/ TBR, offset 0x048/0x04C



Bit/Field	Name	Type	Reset	Description
31:0	TAR	RO	0xFFFF.FFFF	GPTM Timer A Register A read returns the current value of the **GPTM Timer A Count Register**, in all cases except for Input Edge Count and Time modes. In the Input Edge Count mode, this register contains the number of edges that have occurred. In the Input Edge Time mode, this register contains the time at which the last edge event took place.

(Same for Timer B)

Timer A/B Value : TAV/ TBV, offset 0x050/0x054



Bit/Field	Name	Type	Reset	Description
31:0	TAV	RW	0xFFFF.FFFF	GPTM Timer A Value A read returns the current, free-running value of Timer A in all modes. When written, the value written into this register is loaded into the **GPTMTAR** register on the next clock cycle.

(Same for Timer B)

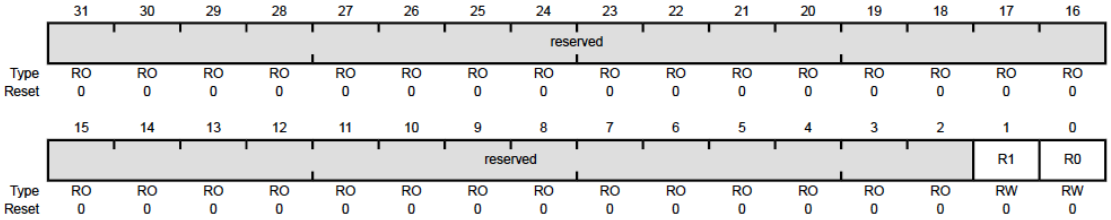
Analog-to-Digital Converter

Base addresses of the ADC modules are:

ADC0: 0x4003.8000

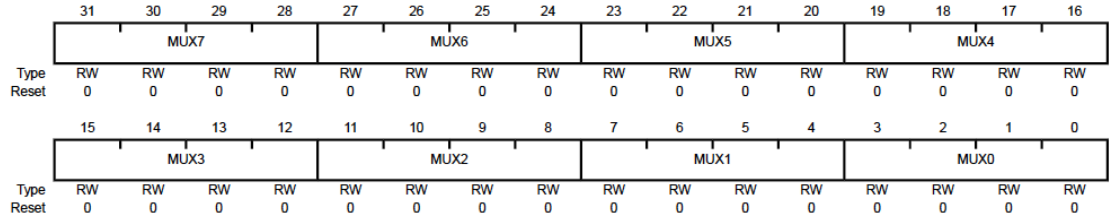
ADC1: 0x4003.9000

Analog-to-Digital Converter Run Mode Clock Gating Control: RCGCADC, 0x400FE638



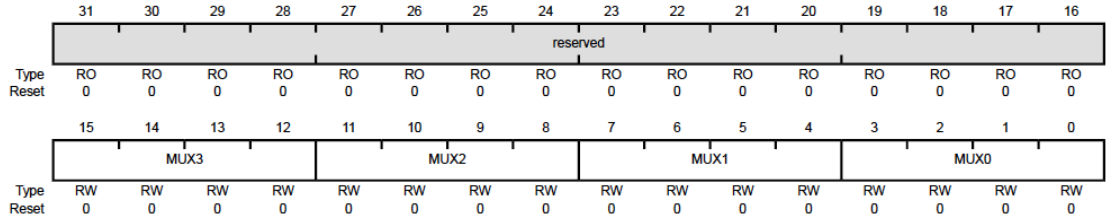
Rn=0 ADC module n is disabled.
Rn=1 Enable and provide a clock to ADC module n in Run mode.

ADC Sample Sequence Input Multiplexer Select 0: ADCSSMUX0, offset 0x040

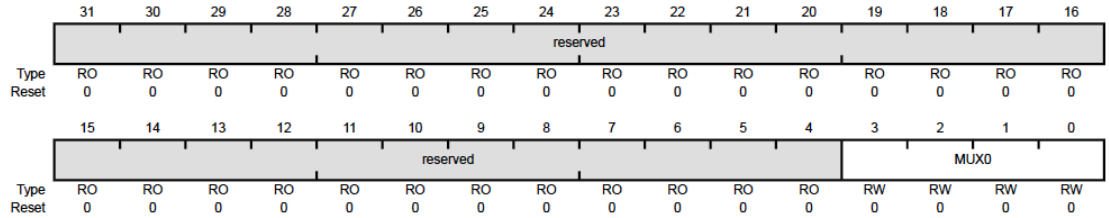


ADC Sample Sequence Input Multiplexer Select 1 (ADCSSMUX1), offset 0x060

ADC Sample Sequence Input Multiplexer Select 2 (ADCSSMUX2), offset 0x080



ADC Sample Sequence Input Multiplexer Select 3 (ADCSSMUX3), offset 0x0A0



Bit/Field	Name	Type	Reset	Description
31:28	MUX7	RW	0x0	8th Sample Input Select The MUX7 field is used during the eighth sample of a sequence executed with the sample sequencer. It specifies which of the analog inputs is sampled for the analog-to-digital conversion. The value set here indicates the corresponding pin, for example, a value of 0x1 indicates the input is AIN1.
27:24	MUX6	RW	0x0	7th Sample Input Select The MUX6 field is used during the seventh sample of a sequence executed with the sample sequencer. It specifies which of the analog inputs is sampled for the analog-to-digital conversion.
23:20	MUX5	RW	0x0	6th Sample Input Select The MUX5 field is used during the sixth sample of a sequence executed with the sample sequencer. It specifies which of the analog inputs is sampled for the analog-to-digital conversion.
19:16	MUX4	RW	0x0	5th Sample Input Select The MUX4 field is used during the fifth sample of a sequence executed with the sample sequencer. It specifies which of the analog inputs is sampled for the analog-to-digital conversion.
15:12	MUX3	RW	0x0	4th Sample Input Select The MUX3 field is used during the fourth sample of a sequence executed with the sample sequencer. It specifies which of the analog inputs is sampled for the analog-to-digital conversion.
11:8	MUX2	RW	0x0	3rd Sample Input Select The MUX2 field is used during the third sample of a sequence executed with the sample sequencer. It specifies which of the analog inputs is sampled for the analog-to-digital conversion.
7:4	MUX1	RW	0x0	2nd Sample Input Select The MUX1 field is used during the second sample of a sequence executed with the sample sequencer. It specifies which of the analog inputs is sampled for the analog-to-digital conversion.
3:0	MUX0	RW	0x0	1st Sample Input Select The MUX0 field is used during the first sample of a sequence executed with the sample sequencer. It specifies which of the analog inputs is sampled for the analog-to-digital conversion.

ADC Sample Sequence Control 0 (ADCSSCTL0), offset 0x044

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	TS7	IE7	END7	D7	TS6	IE6	END6	D6	TS5	IE5	END5	D5	TS4	IE4	END4	D4
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	TS3	IE3	END3	D3	TS2	IE2	END2	D2	TS1	IE1	END1	D1	TS0	IE0	END0	D0
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

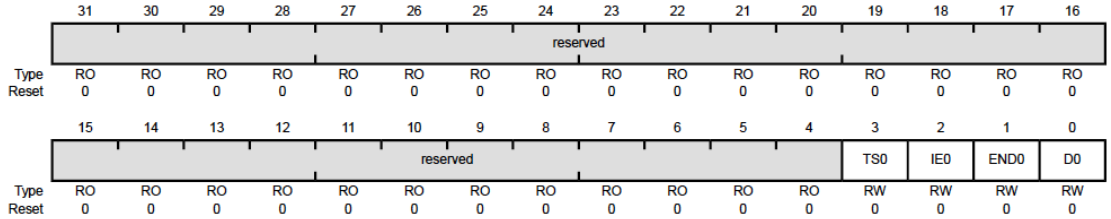
ADC Sample Sequence Control 1 (ADCSSCTL1), offset 0x064

ADC Sample Sequence Control 2 (ADCSSCTL2), offset 0x084.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

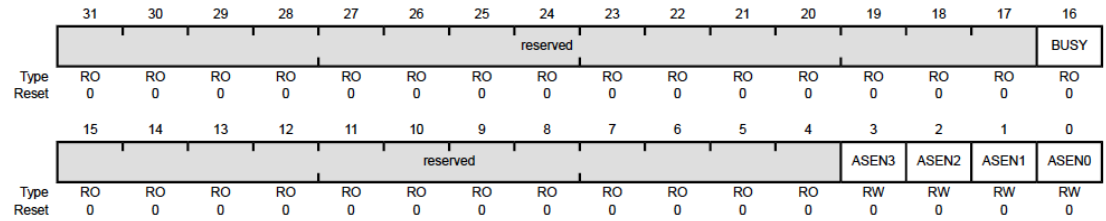
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	TS3	IE3	END3	D3	TS2	IE2	END2	D2	TS1	IE1	END1	D1	TS0	IE0	END0	D0
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

ADC Sample Sequence Control 3 (ADCSSCTL3), offset 0x0A4



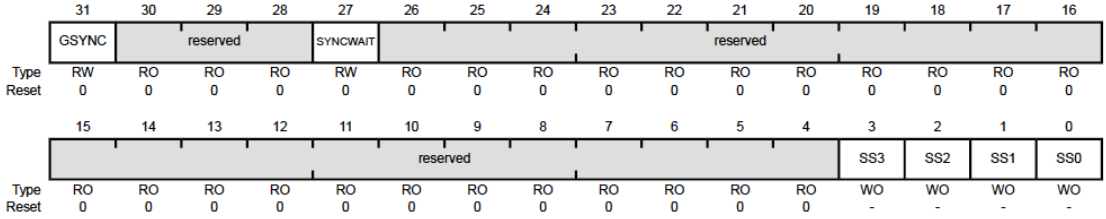
3	TS0	RW	0	1st Sample Temp Sensor Select Value Description 0 The input pin specified by the **ADCSSMUXn** register is read during the first sample of the sample sequence. 1 The temperature sensor is read during the first sample of the sample sequence.
2	IE0	RW	0	1st Sample Interrupt Enable Value Description 0 The raw interrupt is not asserted to the interrupt controller. 1 The raw interrupt signal (INR0 bit) is asserted at the end of the first sample's conversion. If the MASK0 bit in the **ADCIM** register is set, the interrupt is promoted to the interrupt controller. It is legal to have multiple samples within a sequence generate interrupts.
1	END0	RW	0	1st Sample is End of Sequence Value Description 0 Another sample in the sequence is the final sample. 1 The first sample is the last sample of the sequence. It is possible to end the sequence on any sample position. Software must set an ENDn bit somewhere within the sequence. Samples defined after the sample containing a set ENDn bit are not requested for conversion even though the fields may be non-zero.
0	D0	RW	0	1st Sample Differential Input Select Value Description 0 The analog inputs are not differentially sampled. 1 The analog input is differentially sampled. The corresponding **ADCSSMUXn** nibble must be set to the pair number "i", where the paired inputs are "2i and 2i+1". Because the temperature sensor does not have a differential option, this bit must not be set when the TS0 bit is set.

ADC Active Sample Sequencer (ADCACTSS), offset 0x000



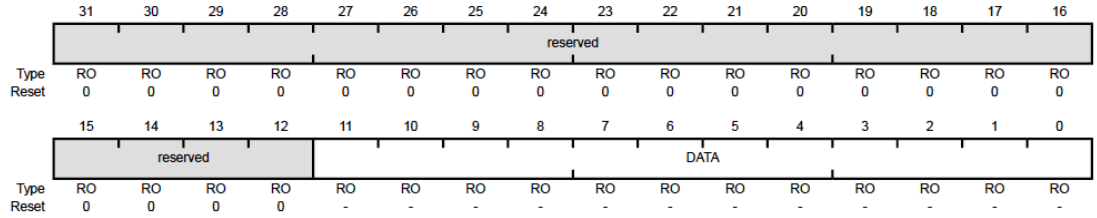
ASENn=0, sample sequencer n is disabled,
ASENn=1, sample sequencer n is enabled.
BUSY=0, ADC is idle,
BUSY=1, ADC is busy.

ADC Processor Sample Sequence Initiate (ADCPSSI), offset 0x028

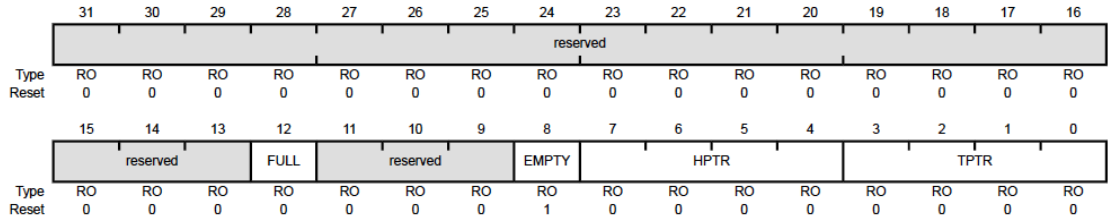


GSYNC bit is cleared once sampling has been initiated.
GSYNC=1 This bit initiates sampling in multiple ADC modules at the same time. Any ADC module that has been initialized by setting an SSn bit and the SYNCWAIT bit starts sampling once this bit is written.
SYNCWAIT=0 Sampling begins when a sample sequence has been initiated.
SYNCWAIT=1 This bit allows the sample sequences to be initiated, but delays sampling until the GSYNC bit is set.
SSn=0 No effect.
SSn=1 Begin sampling on Sample Sequencer n, if the sequencer is enabled in the ADCACTSS register.

ADC Sample Sequence Result FIFO 0 (ADCSSFIFO0), offset 0x048
ADC Sample Sequence Result FIFO 1 (ADCSSFIFO1), offset 0x068
ADC Sample Sequence Result FIFO 2 (ADCSSFIFO2), offset 0x088
ADC Sample Sequence Result FIFO 3 (ADCSSFIFO3), offset 0x0A8



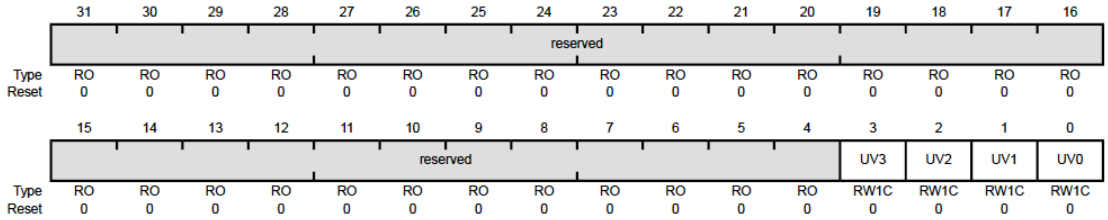
ADC Sample Sequence FIFO 0 Status (ADCSSFSTAT0), offset 0x04C
ADC Sample Sequence FIFO 1 Status (ADCSSFSTAT1), offset 0x06C
ADC Sample Sequence FIFO 2 Status (ADCSSFSTAT2), offset 0x08C
ADC Sample Sequence FIFO 3 Status (ADCSSFSTAT3), offset 0x0AC



FULL=0 The FIFO is not currently full.
FULL=1 The FIFO is currently full.
EMPTY=0 The FIFO is not currently empty.
EMPTY=1 The FIFO is currently empty.
HPTR / TPTR: the current "head" / "tail" pointer index for the FIFO, that is, the next entry to

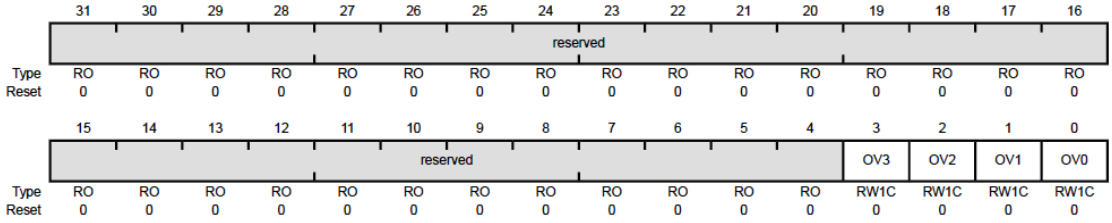
be written.

ADC Underflow Status (ADCUSTAT), offset 0x018



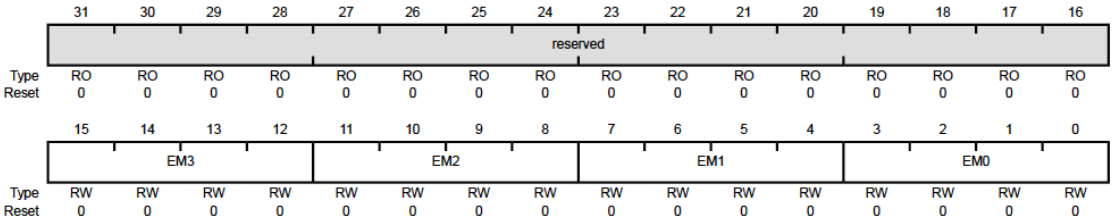
UVn=0 The FIFO has not underflowed.
UVn=1 The FIFO for the Sample Sequencer has hit an underflow condition, meaning that the FIFO is empty and a read was requested. The problematic read does not move the FIFO pointers, and 0s are returned.
This bit is cleared by writing a 1.

ADC Overflow Status (ADCOSTAT), offset 0x010



OVn=0 The FIFO has not overflowed.
OVn=1 The FIFO for Sample Sequencer n has hit an overflow condition, meaning that the FIFO is full and a write was requested. When an overflow is detected, the most recent write is dropped.
This bit is cleared by writing a 1.

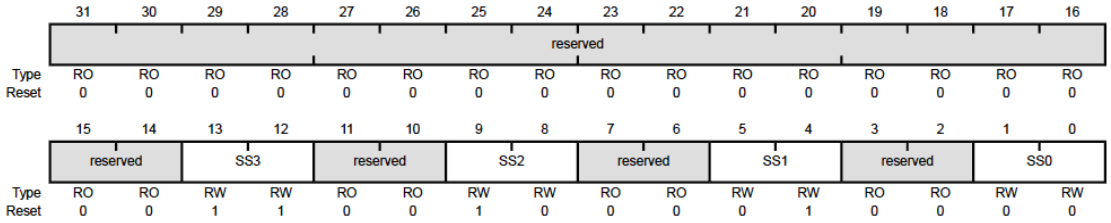
ADC Event Multiplexer Select (ADCEMUX), offset 0x014



Name	Type	Reset	Description
EMn	RW	0x0	SSn Trigger Select This field selects the trigger source for Sample Sequencern . The valid configurations for this field are:

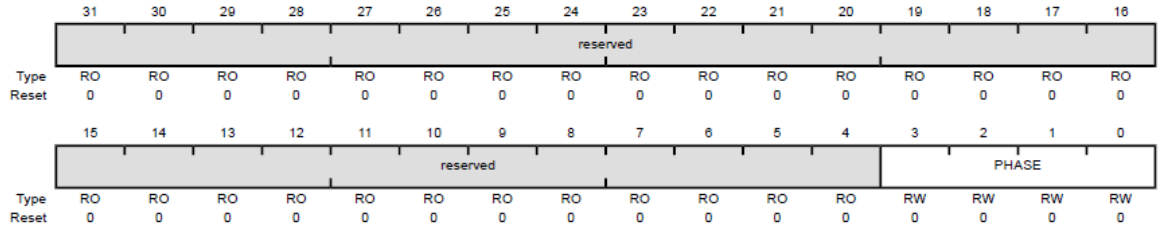
Value	Event
0x0	Processor (default) The trigger is initiated by setting the SSn bit in the **ADCPSSI** register.
0x1	Analog Comparator 0 This trigger is configured by the **Analog Comparator Control 0 (ACCTL0)** register
0x2	Analog Comparator 1 This trigger is configured by the **Analog Comparator Control 1 (ACCTL1)** register
0x3	reserved
0x4	External (GPIO Pins) This trigger is connected to the GPIO interrupt for the corresponding GPIO (see "ADC Trigger Source" on page 655). **Note:** GPIOs that have AINx signals as alternate functions can be used to trigger the ADC. However, the pin cannot be used as both a GPIO and an analog input.
0x5	Timer In addition, the trigger must be enabled with the TnOTE bit in the **GPTMCTL** register
0x6	PWM generator 0 The PWM generator 0 trigger can be configured with the **PWM0 Interrupt and Trigger Enable (PWM0INTEN)** register
0x7	PWM generator 1 The PWM generator 1 trigger can be configured with the **PWM1INTEN** register
0x8	PWM generator 2 The PWM generator 2 trigger can be configured with the **PWM2INTEN** register
0x9	PWM generator 3 The PWM generator 3 trigger can be configured with the **PWM3INTEN** register
0xA-0xE	reserved
0xF	Always (continuously sample)

ADC Sample Sequencer Priority (ADCSSPRI), offset 0x020



SSn: This field contains a binary-encoded value that specifies the priority encoding of Sample Sequencer n. A priority encoding of 0x0 is highest and 0x3 is lowest.

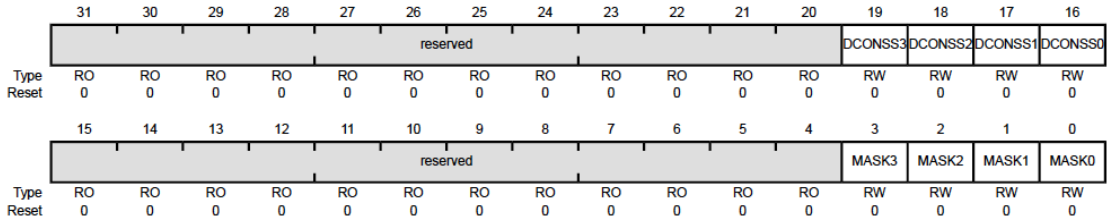
ADC Sample Phase Control: ADCSPC



Name	Type	Reset	Description
PHASE	RW	0x0	Phase Difference This field selects the sample phase difference from the standard sample time.

Value	Description
0x0	ADC sample lags by 0.0°
0x1	ADC sample lags by 22.5°
0x2	ADC sample lags by 45.0°
0x3	ADC sample lags by 67.5°
0x4	ADC sample lags by 90.0°
0x5	ADC sample lags by 112.5°
0x6	ADC sample lags by 135.0°
0x7	ADC sample lags by 157.5°
0x8	ADC sample lags by 180.0°
0x9	ADC sample lags by 202.5°
0xA	ADC sample lags by 225.0°
0xB	ADC sample lags by 247.5°
0xC	ADC sample lags by 270.0°
0xD	ADC sample lags by 292.5°
0xE	ADC sample lags by 315.0°
0xF	ADC sample lags by 337.5°

ADC Interrupt Mask (ADCIM), offset 0x008



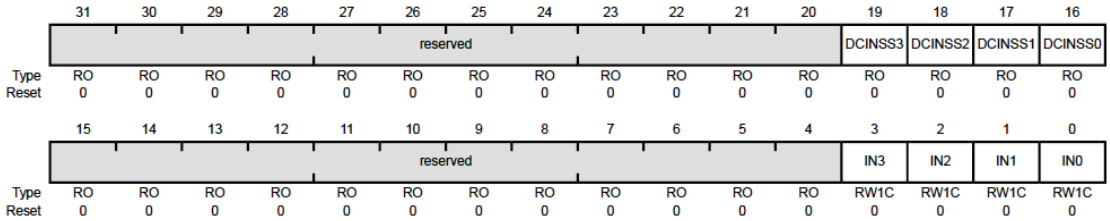
DCONSSn=0 The status of the digital comparators does not affect the SSn interrupt status.

DCONSSn=1 The raw interrupt signal from the digital comparators (INRDC bit in the ADCRIS register) is sent to the interrupt controller on the SSn interrupt line.

MASKn=0 The status of Sample Sequencer n does not affect the SSn interrupt status.

MASKn=1 The raw interrupt signal from Sample Sequencer 3 (ADCRIS register INRn bit) is sent to the interrupt controller.

ADC Interrupt Status and Clear (ADCISC), offset 0x00C

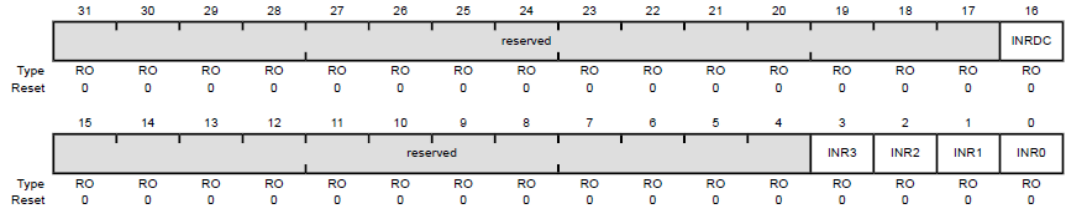


DCINSSn=0 No interrupt has occurred or the interrupt is masked. Both the INRDC bit in the ADCRIS register and the DCONSSn bit in the ADCIM register are set, providing a level-based interrupt to the interrupt controller.

DCINSSn=1 This bit is cleared by writing a 1 to it. Clearing this bit also clears the INRDC bit in the ADCRIS register.

INn=0 No interrupt has occurred or the interrupt is masked.
INn=1 Both the INRn bit in the ADCRIS register and the MASKn bit in the ADCIM register are set, providing a level-based interrupt to the interrupt controller.
This bit is cleared by writing a 1. Clearing this bit also clears the INRn bit in the ADCRIS register.

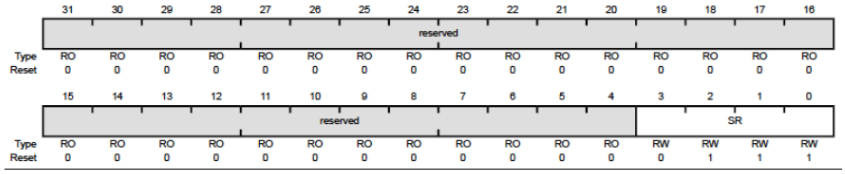
ADC Raw Interrupt Status (ADCRIS), offset 0x004



INRn = 0 An interrupt has not occurred.
INRn = 1 A sample has completed conversion and if the respective ADCSSCTLn IEx bit is set, enabling a raw interrupt.

This bit is cleared by writing a 1 to the INn bit in the ADCISC register.

ADC Peripheral Configuration (ADCPC), offset 0xFC4



ADC Sample Rate

This field specifies the number of ADC conversions per second and is used in Run, Sleep, and Deep-Sleep modes. The field encoding is based on the legacy RCGC0 register encoding. The programmed sample rate cannot exceed the maximum sample rate specified by the MSR field in the ADCPP register. The SR field is encoded as follows:

Value	Description
0x0	Reserved
0x1	125 ksps
0x2	Reserved
0x3	250 ksps
0x4	Reserved
0x5	500 ksps
0x6	Reserved
0x7	1 Msps
0x8 - 0xF	Reserved

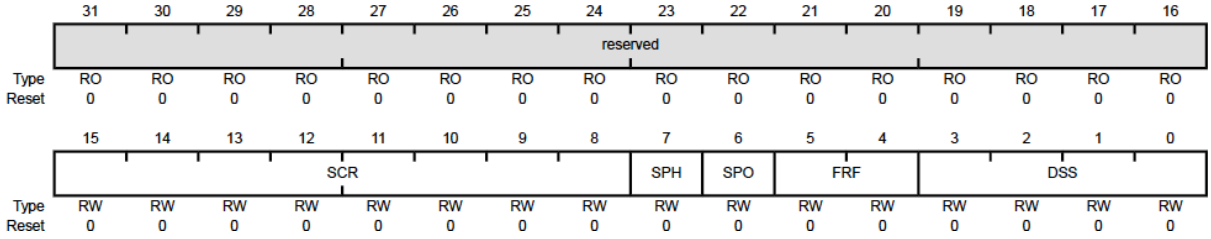
Serial

Base addresses of the modules:

SSI0: 0x4000.8000
SSI1: 0x4000.9000
SSI2: 0x4000.A000
SSI3: 0x4000.B000

Offset	Name	Type	Reset	Description
0x000	SSICR0	RW	0x0000.0000	SSI Control 0
0x004	SSICR1	RW	0x0000.0000	SSI Control 1
0x008	SSIDR	RW	0x0000.0000	SSI Data
0x00C	SSISR	RO	0x0000.0003	SSI Status
0x010	SSICPSR	RW	0x0000.0000	SSI Clock Prescale
0x014	SSIIM	RW	0x0000.0000	SSI Interrupt Mask
0x018	SSIRIS	RO	0x0000.0008	SSI Raw Interrupt Status
0x01C	SSIMIS	RO	0x0000.0000	SSI Masked Interrupt Status
0x020	SSIICR	W1C	0x0000.0000	SSI Interrupt Clear
0x024	SSIDMACTL	RW	0x0000.0000	SSI DMA Control
0xFC8	SSICC	RW	0x0000.0000	SSI Clock Configuration
0xFD0	SSIPeriphID4	RO	0x0000.0000	SSI Peripheral Identification 4
0xFD4	SSIPeriphID5	RO	0x0000.0000	SSI Peripheral Identification 5
0xFD8	SSIPeriphID6	RO	0x0000.0000	SSI Peripheral Identification 6
0xFDC	SSIPeriphID7	RO	0x0000.0000	SSI Peripheral Identification 7
0xFE0	SSIPeriphID0	RO	0x0000.0022	SSI Peripheral Identification 0
0xFE4	SSIPeriphID1	RO	0x0000.0000	SSI Peripheral Identification 1
0xFE8	SSIPeriphID2	RO	0x0000.0018	SSI Peripheral Identification 2
0xFEC	SSIPeriphID3	RO	0x0000.0001	SSI Peripheral Identification 3
0xFF0	SSIPCellID0	RO	0x0000.000D	SSI PrimeCell Identification 0
0xFF4	SSIPCellID1	RO	0x0000.00F0	SSI PrimeCell Identification 1
0xFF8	SSIPCellID2	RO	0x0000.0005	SSI PrimeCell Identification 2
0xFFC	SSIPCellID3	RO	0x0000.00B1	SSI PrimeCell Identification 3

SSI Control 0 (SSICR0), offset 0x000



15:8 SCR RW 0x00 SSI Serial Clock Rate

This bit field is used to generate the transmit and receive bit rate of the SSI. The bit rate is:

`BR=SysClk/(CPSDVSR * (1 + SCR))`

where `CPSDVSR` is an even value from 2-254 programmed in the **SSICPSR** register, and `SCR` is a value from 0-255.

7 SPH RW 0 SSI Serial Clock Phase

This bit is only applicable to the Freescale SPI Format.

The `SPH` control bit selects the clock edge that captures data and allows it to change state. This bit has the most impact on the first bit transmitted by either allowing or not allowing a clock transition before the first data capture edge.

Value	Description
0	Data is captured on the first clock edge transition.
1	Data is captured on the second clock edge transition.

6 SPO RW 0 SSI Serial Clock Polarity

Value	Description
0	A steady state Low value is placed on the `SSInClk` pin.
1	A steady state High value is placed on the `SSInClk` pin when data is not being transferred.

Note: If this bit is set, then software must also configure the GPIO port pin corresponding to the `SSInClk` signal as a pull-up in the **GPIO Pull-Up Select (GPIOPUR)** register.

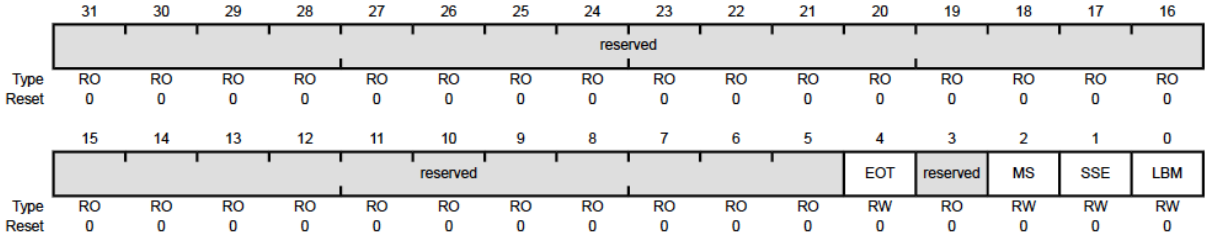
5:4 FRF RW 0x0 SSI Frame Format Select

Value	Frame Format
0x0	Freescale SPI Frame Format
0x1	Texas Instruments Synchronous Serial Frame Format
0x2	MICROWIRE Frame Format
0x3	Reserved

3:0	DSS	RW	0x0	SSI Data Size Select

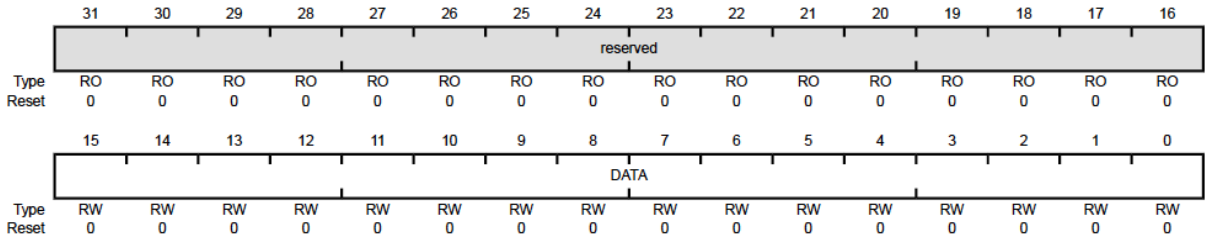
Value	Data Size
0x0-0x2	Reserved
0x3	4-bit data
0x4	5-bit data
0x5	6-bit data
0x6	7-bit data
0x7	8-bit data
0x8	9-bit data
0x9	10-bit data
0xA	11-bit data
0xB	12-bit data
0xC	13-bit data
0xD	14-bit data
0xE	15-bit data
0xF	16-bit data

SSI Control 0 (SSICR0), offset 0x000

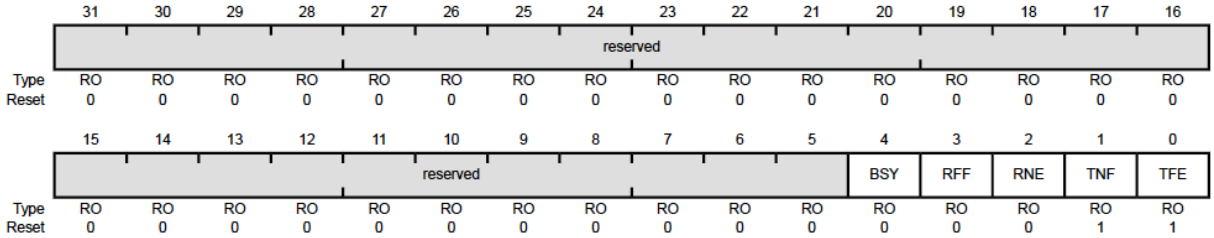


4	EOT	RW	0	End of Transmission This bit is only valid for Master mode devices and operations (MS = 0x0). Value Description 0 The TXRIS interrupt indicates that the transmit FIFO is half full or less. 1 The End of Transmit interrupt mode for the TXRIS interrupt is enabled.
2	MS	RW	0	SSI Master/Slave Select This bit selects Master or Slave mode and can be modified only when the SSI is disabled (SSE=0). Value Description 0 The SSI is configured as a master. 1 The SSI is configured as a slave.
1	SSE	RW	0	SSI Synchronous Serial Port Enable Value Description 0 SSI operation is disabled. 1 SSI operation is enabled. **Note:** This bit must be cleared before any control registers are reprogrammed.
0	LBM	RW	0	SSI Loopback Mode Value Description 0 Normal serial port operation enabled. 1 Output of the transmit serial shift register is connected internally to the input of the receive serial shift register.

SSI Data (SSIDR), offset 0x008



SSI Status (SSISR), offset 0x00C



Bit/Field	Name	Type	Reset	Description
4	BSY	RO	0	SSI Busy Bit

Value	Description
0	The SSI is idle.
1	The SSI is currently transmitting and/or receiving a frame, or the transmit FIFO is not empty.

Bit/Field	Name	Type	Reset	Description
3	RFF	RO	0	SSI Receive FIFO Full

Value	Description
0	The receive FIFO is not full.
1	The receive FIFO is full.

Bit/Field	Name	Type	Reset	Description
2	RNE	RO	0	SSI Receive FIFO Not Empty

Value	Description
0	The receive FIFO is empty.
1	The receive FIFO is not empty.

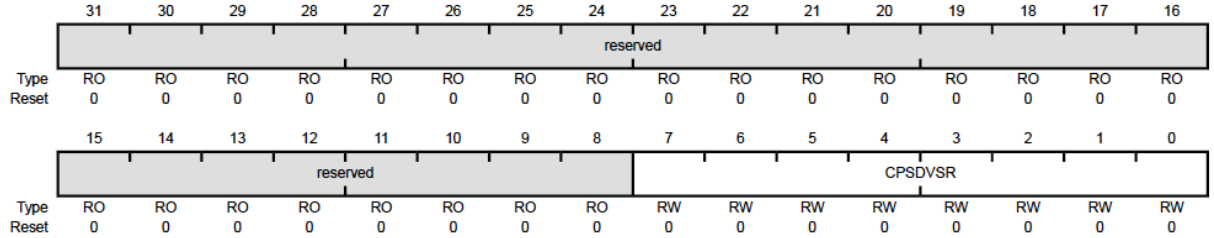
Bit/Field	Name	Type	Reset	Description
1	TNF	RO	1	SSI Transmit FIFO Not Full

Value	Description
0	The transmit FIFO is full.
1	The transmit FIFO is not full.

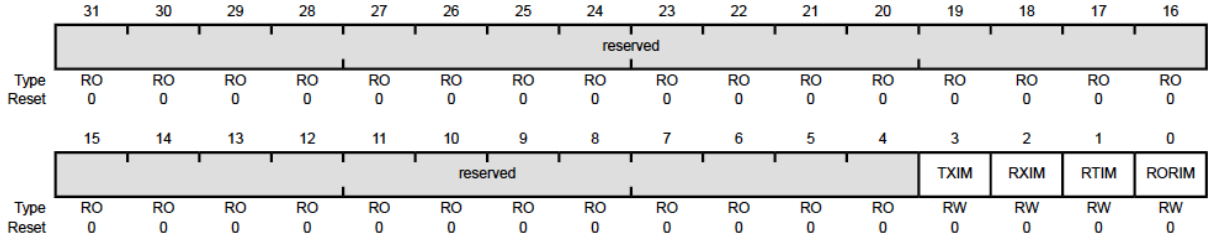
Bit/Field	Name	Type	Reset	Description
0	TFE	RO	1	SSI Transmit FIFO Empty

Value	Description
0	The transmit FIFO is not empty.
1	The transmit FIFO is empty.

SSI Clock Prescale (SSICPSR), offset 0x010



SSI Interrupt Mask (SSIIM), offset 0x014



3	TXIM	RW	0	SSI Transmit FIFO Interrupt Mask

Value	Description
0	The transmit FIFO interrupt is masked.
1	The transmit FIFO interrupt is not masked.

2	RXIM	RW	0	SSI Receive FIFO Interrupt Mask

Value	Description
0	The receive FIFO interrupt is masked.
1	The receive FIFO interrupt is not masked.

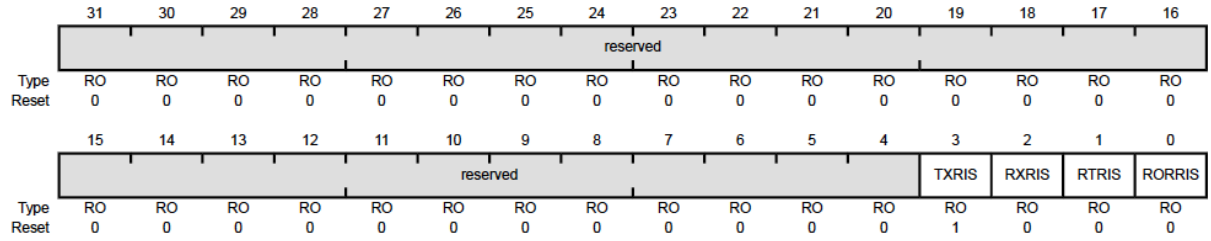
1	RTIM	RW	0	SSI Receive Time-Out Interrupt Mask

Value	Description
0	The receive FIFO time-out interrupt is masked.
1	The receive FIFO time-out interrupt is not masked.

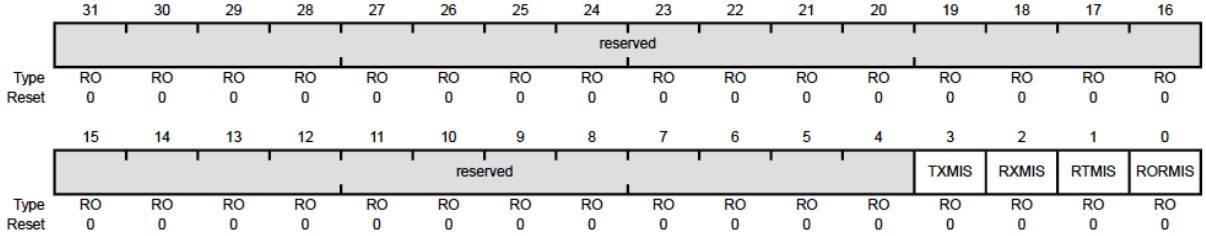
0	RORIM	RW	0	SSI Receive Overrun Interrupt Mask

Value	Description
0	The receive FIFO overrun interrupt is masked.
1	The receive FIFO overrun interrupt is not masked.

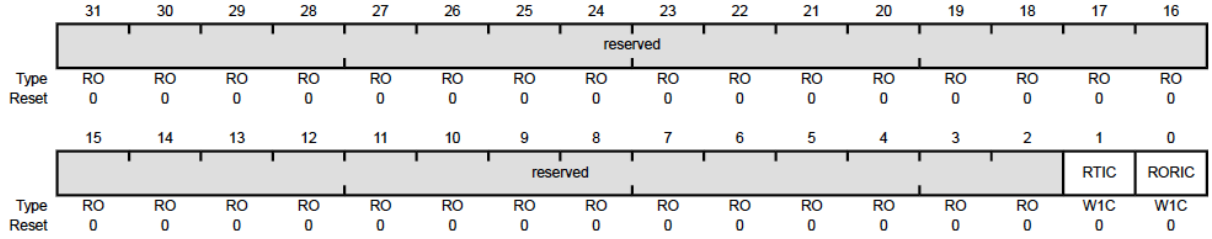
SSI Raw Interrupt Status (SSIRIS), offset 0x018



SSI Masked Interrupt Status (SSIMIS), offset 0x01C

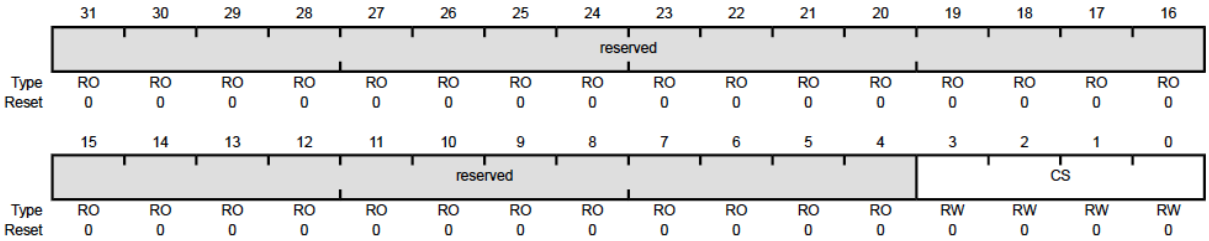


SSI Interrupt Clear (SSIICR), offset 0x020



1	RTIC	W1C	0	SSI Receive Time-Out Interrupt Clear Writing a 1 to this bit clears the `RTRIS` bit in the **SSIRIS** register and the `RTMIS` bit in the **SSIMIS** register.
0	RORIC	W1C	0	SSI Receive Overrun Interrupt Clear Writing a 1 to this bit clears the `RORRIS` bit in the **SSIRIS** register and the `RORMIS` bit in the **SSIMIS** register.

SSI Clock Configuration (SSICC), offset 0xFC8



3:0	CS	RW	0	SSI Baud Clock Source The following table specifies the source that generates for the SSI baud clock:

Value	Description
0x0	System clock (based on clock source and divisor factor)
0x1-0x4	reserved
0x5	PIOSC
0x6 - 0xF	Reserved