Fundamentals of Embedded and Real Time Systems

MODULE 09

TAMER AWAD

Grading and Attendance

This is a **pass/fail** course that is dependent on performance and participation.

~~Classroom students must attend at least 80% of sessions, in-person, to be eligible to pass the course.~~

Online students must **participate online**, ~~or in person~~, for **at least 60% of the sessions** to be eligible to pass the course.

All students must **complete a minimum of 80% of total assignments** to be eligible to pass the course. Individual assignments will have prescribed weights and due dates.

Review Module 08

Module 09

Exceptions & Interrupts
- Why Interrupts
- Exceptions & Interrupts
- Interrupt Sources
- Nested Vector Interrupt Controller
- Exception definitions
- Interrupt handlers as regular C-functions in Cortex-M

Interrupt Masking
- Masking interrupts and exceptions
- PRIMASK Register
- FAULTMASK Register

Interrupt Service Routines
- What are ISRs
- Characteristics of a good ISR
- Important considerations for ISRs in C
- Non-Nested Interrupts
- Nested Interrupts

Realtime Operating Systems
- Operating System Terminology
- What is a Realtime Operating System
- Desktop vs RTOS based applications
- Preemptive vs Non-Preemptive RTOS
- When to use an RTOS
- The Scheduler
- Tasks & Data
- RTOS Services

Pulse Width Modulation
- What is PWM
- Hardware Timers
- Timer Input Capture Mode
- Generating PWM with Timer Output
- Multi Channel Outputs

Bonus Assignment
- Assignment 08

Exceptions & Interrupts

- Why Interrupts
- Exceptions & Interrupts
- Sources
- Nested Vector Interrupt Controller
- Exception definitions
- Interrupt handlers as regular C-functions in Cortex-M

Why Interrupts

- System response is an important aspect of embedded systems
- "*How to make the system react rapidly to external events, even if it is in the middle of doing something else*"
- One approach to the response problem is to use ***interrupts***.
- Interrupts cause the microprocessor in the embedded system to suspend doing whatever it is doing and to execute some different code instead, code that will respond to whatever event caused the interrupt.
- *Source: An Embedded Software Primer (David E. Simon)*

Exceptions & Interrupts

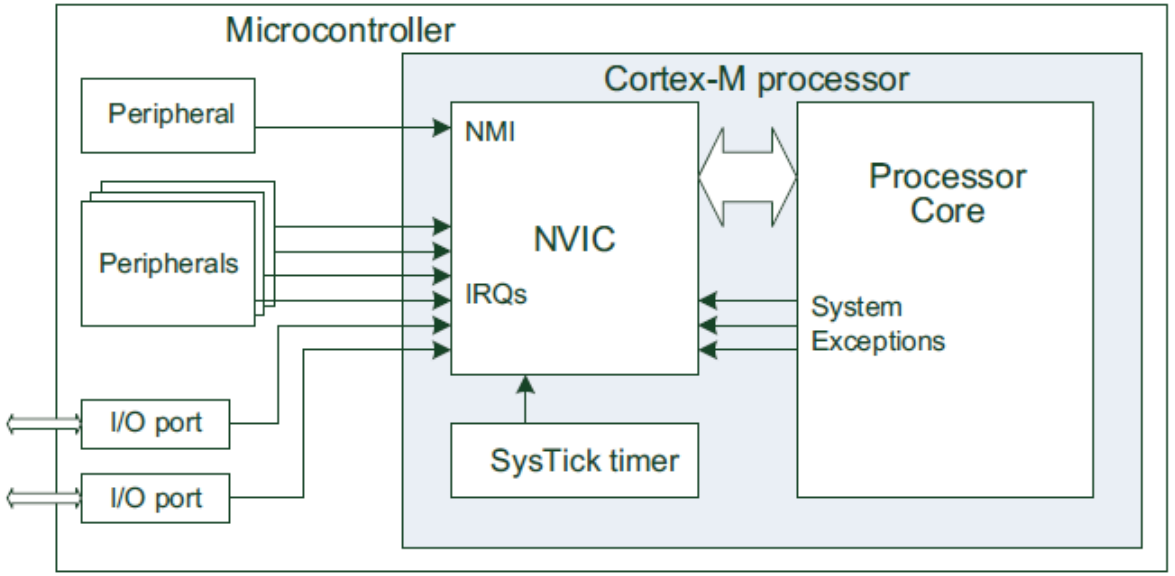
- Exceptions are events that cause changes to program flow.
- When an exception happens, the processor suspends the current executing task and executes a part of the program called the exception handler.
- After the execution of the exception handler is completed, the processor then resumes normal program execution.
- **In the ARM architecture, interrupts are one type of exception.**
- Interrupts are usually generated from peripheral or external inputs, and in some cases, they can be triggered by software.
- The exception handlers for interrupts are also referred to as Interrupt Service Routines (ISR).

Interrupt Sources

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

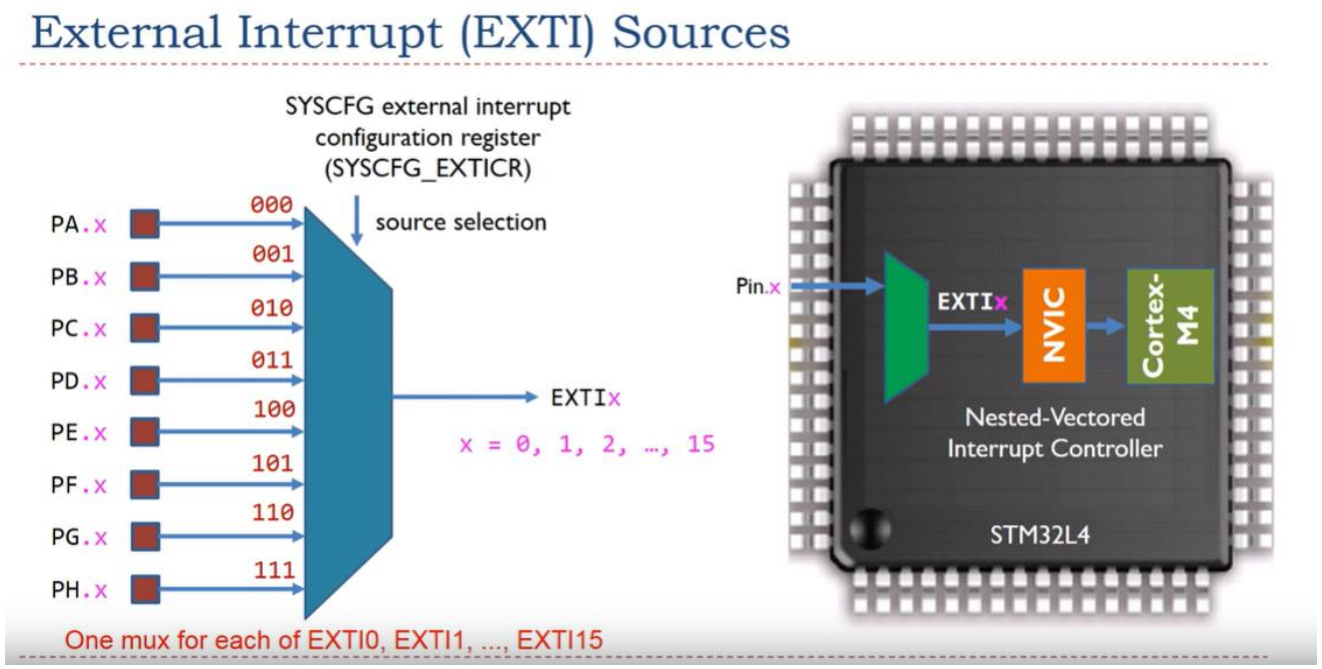
Interrupt Sources

- Each exception source has an exception number.
- Exception numbers 1 to 15 are classified as **system exceptions**
- Exceptions 16 and above are for interrupts.
- The design of the NVIC in the Cortex-M3 and Cortex-M4 processors can support up to 240 interrupt inputs.
- *Source: The Definitive Guide to ARM Cortex*



Interrupt Sources (EXTI)

- NVIC supports 16 External Interrupts (EXTI) associated with GPIO pins.
- EXTI0 to EXTI15 each one of these interrupt lines is associated with a GPIO.
- The MCU has more GPIO pins than external interrupts.
- There is one multiplexer for each of the 16 external interrupts.
- STM32 maps PinX to EXTIx
- The MUX specifies which pin is selected as an EXTI source.
- **No two or more pins can be used from the same group simultaneously.**

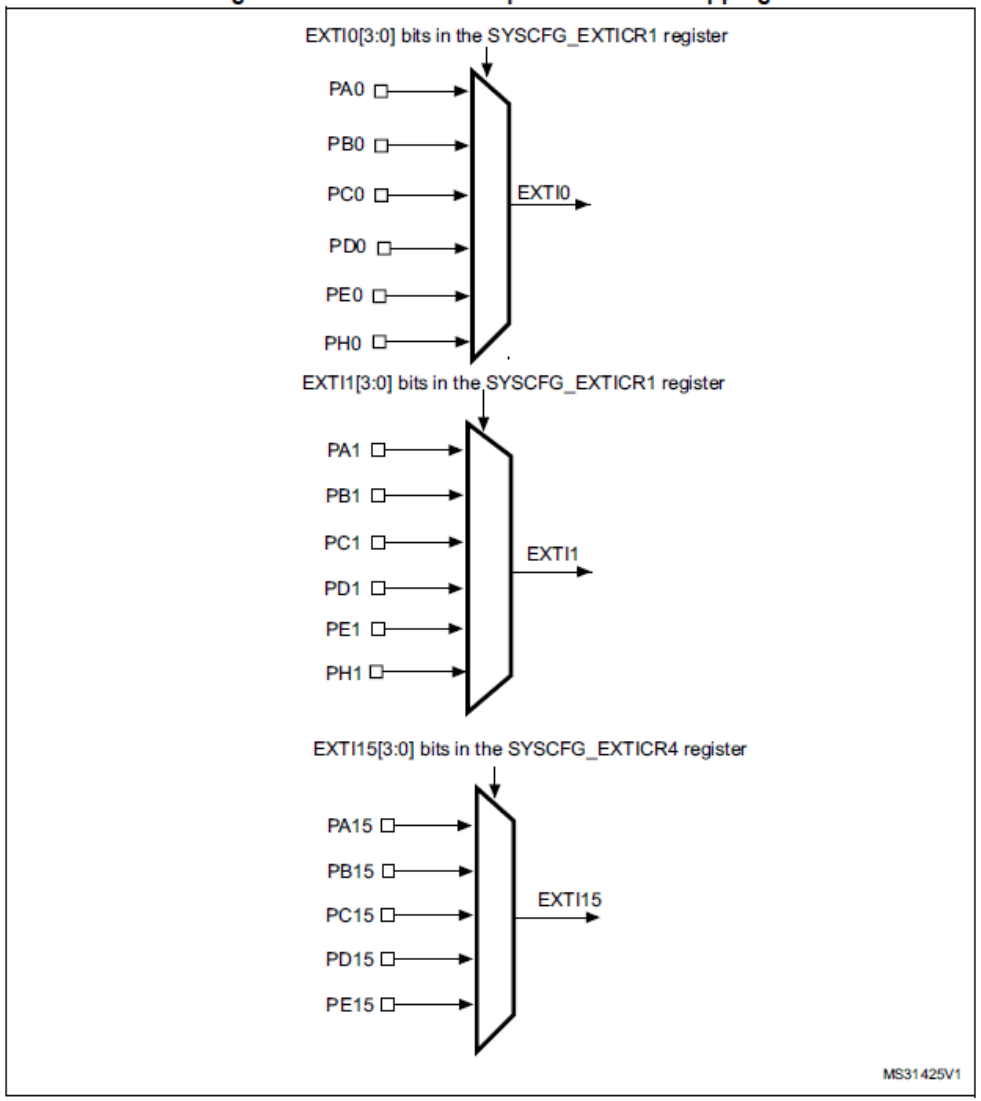


Source: EXTI tutorial by Dr. Yifeng Zhu

10.2.5 External interrupt/event line mapping

Up to 81 GPIOs (STM32F401xB/C and STM32F401xD/E) are connected to the 16 external interrupt/event lines in the following manner:

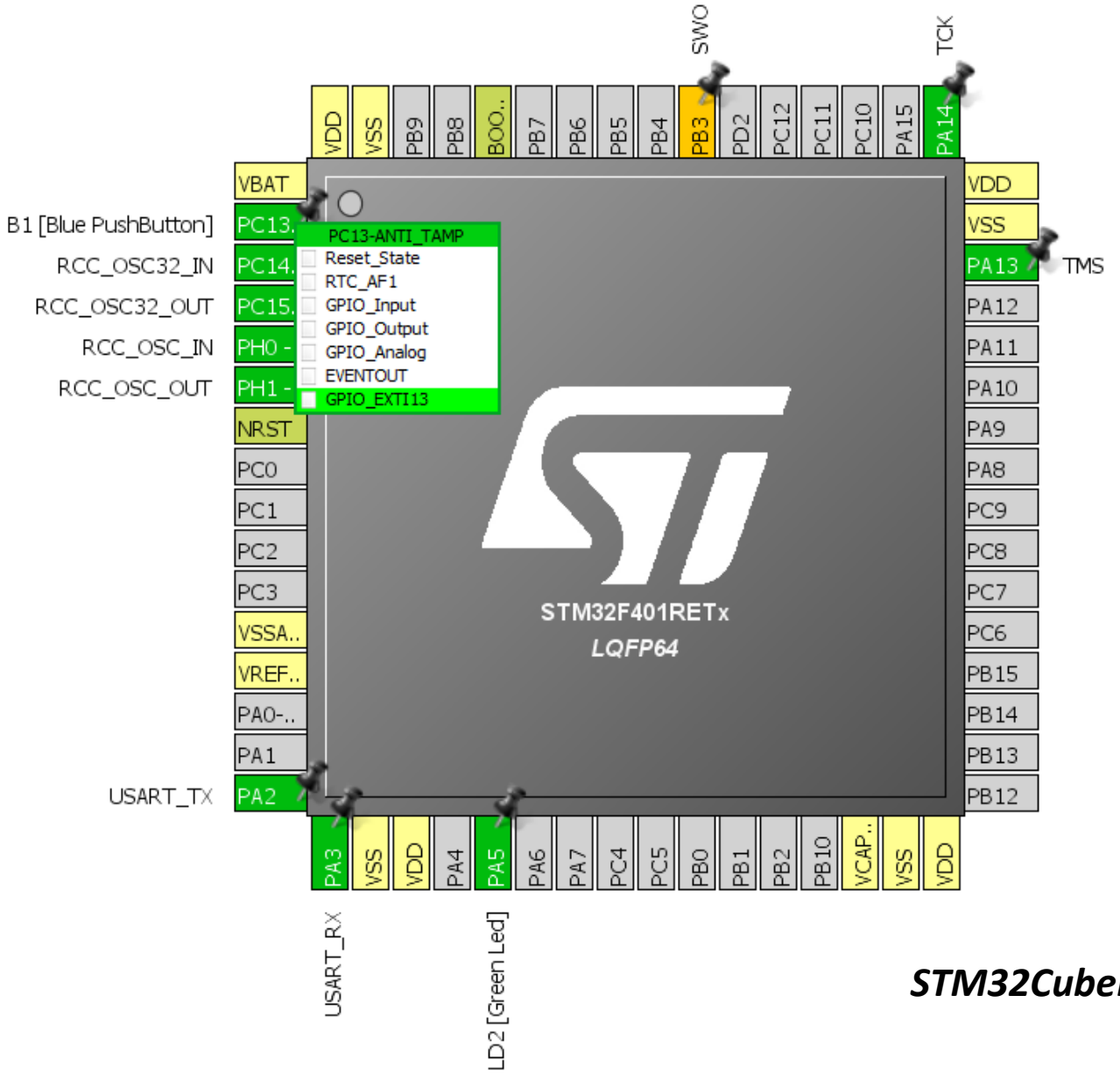
Figure 30. External interrupt/event GPIO mapping



External Interrupts

STM32F401 Reference Manual – RM0368

Push Button PC13 & EXTI13



STM32CubeMX

Nested Vector Interrupt Controller (NVIC)

- The NVIC is a part of the Cortex-M processor.
- It is programmable and its registers are memory mapped. Its registers are located in the System Control Space (SCS) of the memory map.
- Beside interrupts from peripherals and other external inputs, the NVIC also supports a number of system exceptions.
- The NVIC handles the exceptions and interrupt configurations, prioritization, and interrupt masking.
- The NVIC in the Cortex-M3 and M4 processors can support up to 240 interrupt inputs. However, in practice the number of interrupt inputs implemented in the design is far less, typically in the range of 16 to 100.
- The exception number is reflected in various registers, including the IPSR, and it is used to determine the exception vector addresses.
- Exception vectors are stored in a vector table, and the processor reads this table to determine the starting address of an exception handler during the exception entrance sequence.

NVIC Features

- Flexible exception and interrupt management
  - Each interrupt (apart from the NMI) can be enabled or disabled and can have its pending status set or cleared by software
- Nested exception/interrupt support
  - When an exception occurs, the NVIC will compare the priority level of the exception to the current level.
  - If the new exception has a higher priority, the current running task will be suspended.
  - This process is called “preemption.”
- Vectored exception/interrupt entry
  - The Cortex-M processors automatically locate the starting point of the exception handler from a vector table in the memory.
- Interrupt masking
  - Provides several interrupt masking registers (such as PRIMASK & BASEPRI)

Table 7.1 List of System Exceptions

Exception Number	Exception Type	Priority	Descriptions
1	Reset	−3 (Highest)	Reset
2	NMI	−2	Non-Maskable Interrupt (NMI), can be generated from on chip peripherals or from external sources.
3	Hard Fault	−1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage Fault	Programmable	Memory management fault; MPU violation or program execution from address locations with XN (eXecute Never) memory attribute.
5	Bus Fault	Programmable	Bus error; usually occurs when AHB interface receives an error response from a bus slave (also called *prefetch abort* if it is an instruction fetch or *data abort* if it is a data access). Can also be caused by other illegal accesses.
6	Usage Fault	Programmable	Exceptions due to program error or trying to access co-processor (the Cortex-M3 and Cortex-M4 processor do not support co-processors).
7–10	Reserved	NA	–
11	SVC	Programmable	SuperVisor Call; usually used in OS environment to allow application tasks to access system services.
12	Debug Monitor	Programmable	Debug monitor; exception for debug events like breakpoints, watchpoints when software based debug solution is used.
13	Reserved	NA	–
14	PendSV	Programmable	Pendable service call; An exception usually used by an OS in processes like context switching.
15	SYSTICK	Programmable	System Tick Timer; Exception generates by a timer peripheral which is included in the processor. This can be

List of System Exceptions

CMSIS-Core Exception Definitions

Table 7.3 CMSIS-Core Exception Definitions

Exception Number	Exception Type	CMSIS-Core Enumeration (IRQn)	CMSIS-Core Enumeration Value	Exception Handler Name
1	Reset	-	-	Reset_Handler
2	NMI	NonMaskableInt_IRQn	−14	NMI_Handler
3	Hard Fault	HardFault_IRQn	−13	HardFault_Handler
4	MemManage Fault	MemoryManagement_IRQn	−12	MemManage_Handler
5	Bus Fault	BusFault_IRQn	−11	BusFault_Handler
6	Usage Fault	UsageFault_IRQn	−10	UsageFault_Handler
11	SVC	SVCall_IRQn	−5	SVC_Handler
12	Debug Monitor	DebugMonitor_IRQn	−4	DebugMon_Handler
14	PendSV	PendSV_IRQn	−2	PendSV_Handler
15	SYSTICK	SysTick_IRQn	−1	SysTick_Handler
16	Interrupt #0	(device-specific)	0	(device-specific)
17	Interrupt #1 - #239	(device-specific)	1 to 239	(device-specific)

CMSIS Functions for Interrupt Control

Table 7.4 Commonly Used CMSIS-Core Functions for Basic Interrupt Control

Functions	Usage
void NVIC_EnableIRQ (IRQn_Type IRQn)	Enable an external interrupt
void NVIC_DisableIRQ (IRQn_Type IRQn)	Disable an external interrupt
void NVIC_SetPriority (IRQn_Type IRQn, uint32_t priority)	Set the priority of an interrupt
void __enable_irq(void)	Clear PRIMASK to enable interrupts
void __disable_irq(void)	Set PRIMASK to disable all interrupts
void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)	Set priority grouping configuration

How can interrupt handlers be regular C functions in Cortex-M?

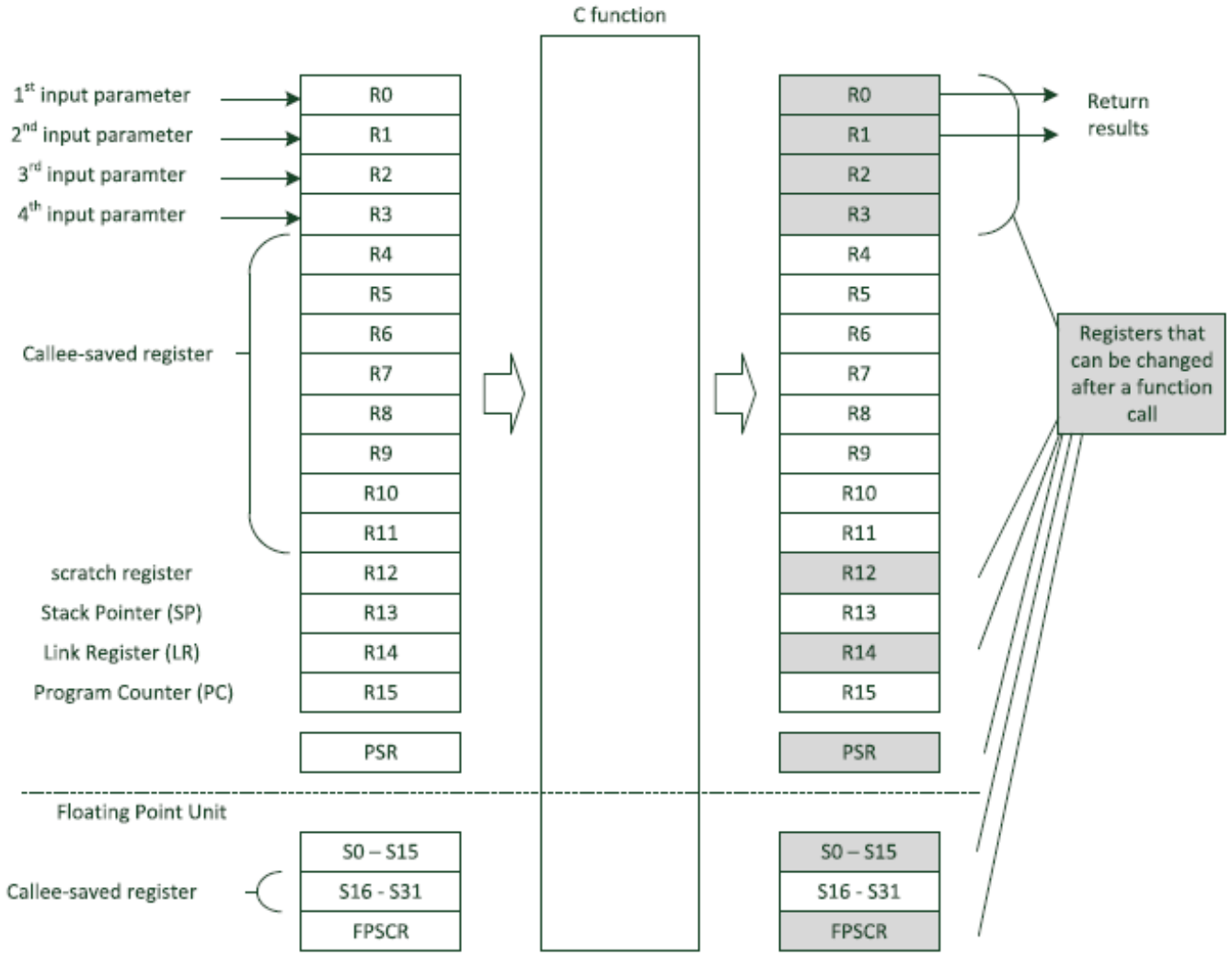


FIGURE 8.1

Register usage in a function call in AAPCS

Register usage in a function call

Interrupt handlers as C-Functions

- The ability for the processor to invoke the interrupt handlers as regular C functions is a unique feature of the ARM Cortex-M processor.
- In most other processors, the interrupt handlers require a special entry code and they return through a special return-from-interrupt instruction, so they can't be regular C functions.
- Also, interrupt handlers typically must save and restore more CPU registers than regular functions.
- *How the Cortex-M chip solved the problem with saving all the right CPU registers and returning from interrupt functions.*

Exceptions handlers as C-Functions

- In order to allow a C function to be used as an exception handler, the exception mechanism needs to save **R0** ***to*** **R3**, **R12**, **LR**, and **PSR** at exception entrance automatically, and restore them at exception exit under the control of the processor's hardware.
- In this way when returned to the interrupted program, all the registers would have the same value as when the interrupt entry sequence started.
- In addition, since the value of the return address (**PC**) is not stored in LR as in normal C function calls (the exception mechanism puts an EXC_RETURN code in LR at exception entry, which is used in exception return), the value of the return address also needs to be saved by the exception sequence.
- So in total eight registers need to be saved during the exception handling sequence on the Cortex-M3 or Cortex-M4 processors
- The block of data that are pushed to the stack memory at exception entrance is called a **stack frame**.

Exception Stack Frame for Cortex-M4

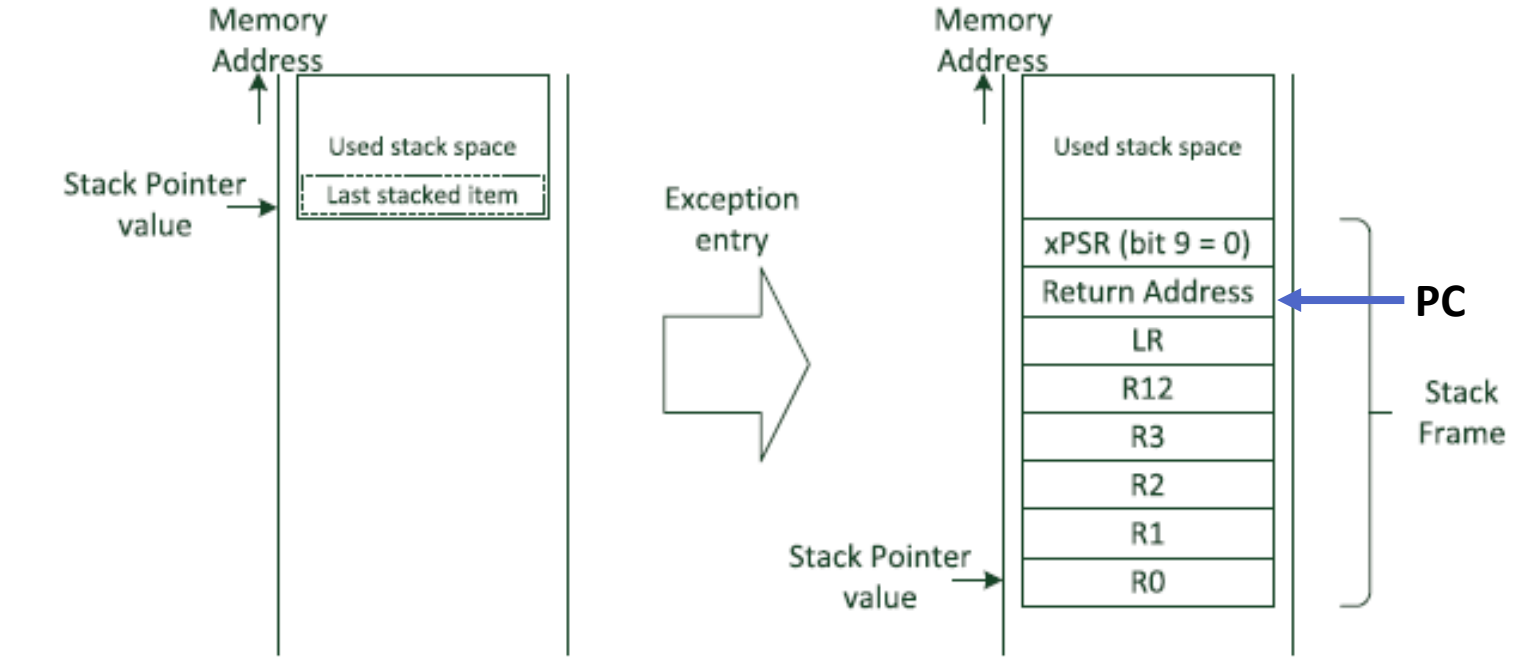


FIGURE 8.2

EXC_RETURN

As the processor enters the exception handler or Interrupt Service Routine (ISR), the value of the Link Register (LR) is updated to a code called EXC_RETURN.

The value of this code is used to trigger the exception return mechanism when it is loaded into the Program Counter (PC) using BX, POP, or memory load instructions (LDR or LDM).

Some bits of the EXC_RETURN code are used to provide additional information about the exception sequence.

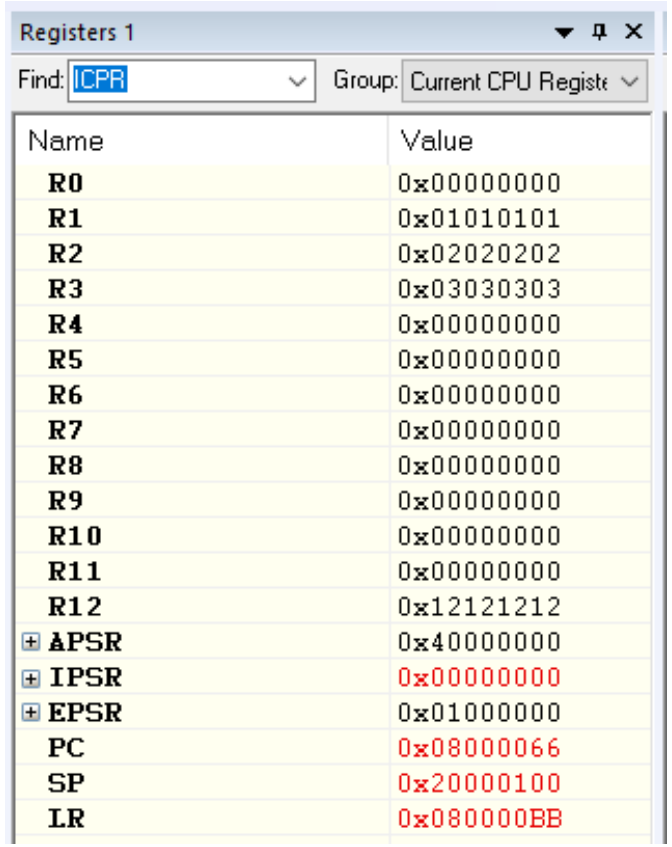
Table 8.2 Valid Values for EXC_RETURN

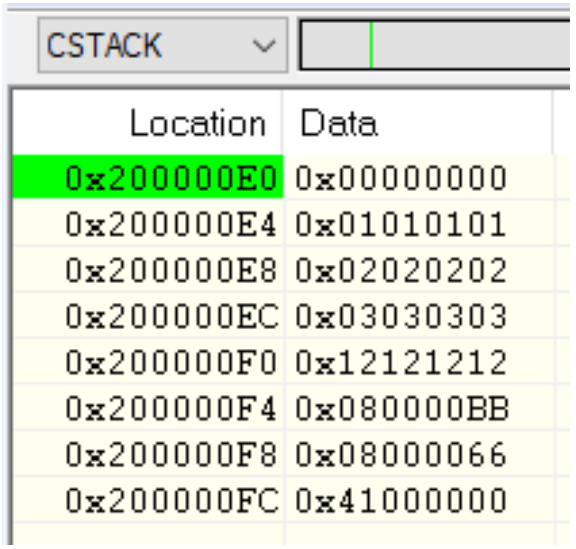
	Floating Point Unit was used before Interrupt (FPCA = 1)	Floating Point Unit was not used before Interrupt (FPCA = 0)
Return to Handler mode (always use Main Stack)	0xFFFFFFE1	0xFFFFFFF1
Return to Thread mode and use the Main Stack for return	0xFFFFFFE9	0xFFFFFFF9
Return to Thread mode and use the Process Stack for return	0xFFFFFFED	0xFFFFFFFD

Demo: Stack Frame

- Using SysTick_Handler demo
- Step into the Handler
- Show the IPSR value of the SysTick interrupt (0xF)
- Show stack frame unwinding
- Map values on stack to the CPU registers

Interrupt Stack Frame





FRAME
R0
R1
R2
R3
R12
LR
PC
PSR

Interrupt Masking

- Masking interrupts and exceptions
- PRIMASK Register
- FAULTMASK Register

Masking interrupts and exceptions

- Might need to temporarily disable all interrupts to carry out some time critical tasks. Can use the PRIMASK register for this purpose.
- The PRIMASK register is a 1-bit wide interrupt mask register.
- When set, it blocks all exceptions (including interrupts) except NMI and Hard-Fault exception.
- The FAULTMASK is very similar to PRIMASK. It will block the Hard-Fault handler.
- Only the NMI exception handler can be executed when FAULTMASK is set.
- The FAULTMASK register is very similar to PRIMASK, but it also blocks the HardFault exception.
- FAULTMASK can be used by fault handling code to suppress the triggering of further faults during fault handling.
- Unlike PRIMASK, FAULTMASK is cleared automatically at exception return.

PRIMASK - *(Section 7.10.1 – The Definitive Guide…)*

- In C programming, you can use the functions provided in CMSIS-Core to set and clear PRIMASK:

```
void __enable_irq();                    // Clear PRIMASK
void __disable_irq();                   // Set PRIMASK
void __set_PRIMASK(uint32_t priMask);   // Set PRIMASK to value
uint32_t __get_PRIMASK(void);           // Read the PRIMASK value
```

- In assembly language programming, you can change the value of PRIMARK register using CPS (Change Processor State) instructions:

```
CPSIE I         ; Clear PRIMASK (Enable interrupts)
CPSID I         ; Set PRIMASK (Disable interrupts)
```

- The PRIMASK register can also be accessed using the MRS and MSR instructions.

```
MOVS R0, #1
MSR PRIMASK, R0         ; Write 1 to PRIMASK to disable all interrupts

MOVS R0, #0
MSR PRIMASK, R0         ; Write 0 to PRIMASK to allow interrupts
```

FAULTMASK - *(Section 7.10.2 – The Definitive Guide…)*

- In C programming, can use the functions provided in CMSIS-Core to set and clear FAULTMASK:

```
void __enable_fault_irq();                  // Clear FAULTMASK
void __disable_fault_irq();                 // Set FAULTMASK
void __set_FAULTMASK(uint32_t faultMask);   // Set FAULTMASK to value
uint32_t __get_FAULTMASK(void);             // Read the FAULTMASK value
```

- In assembly programming, can change the status of the FAULTMASK using CPS instructions:

```
CPSIE F         ; Clear FAULTMASK
CPSID F         ; Set FAULTMASK
```

- The FAULTMASK register can also be accessed using the MRS and MSR instructions.

```
MOVS R0, #1
MSR FAULTMASK, R0           ; Write 1 to FAULTMASK to disable all interrupts

MOVS R0, #0
MSR FAULTMASK, R0           ; Write 0 to FAULTMASK to allow interrupts
```

Demo: Interrupt Mask

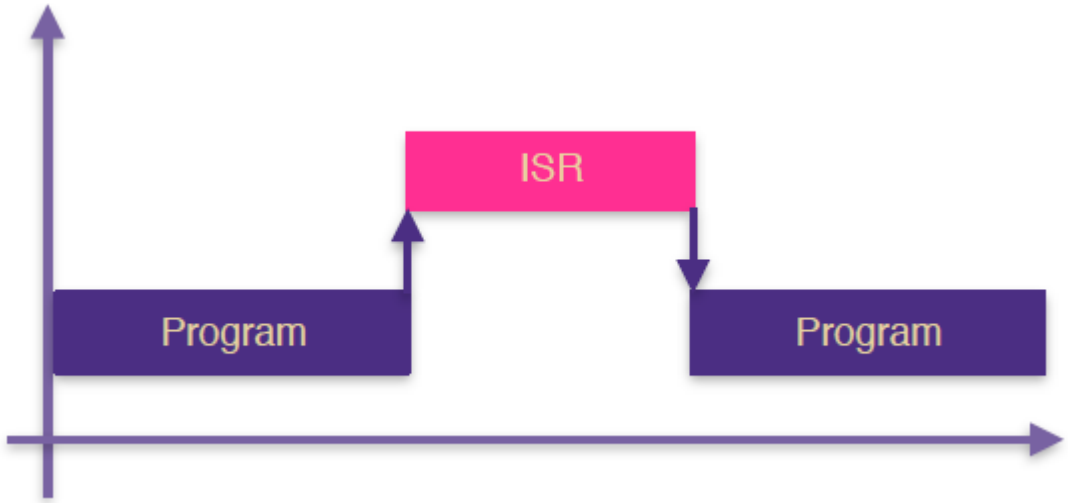
- Using SysTick_Handler demo
- Break into the Handler
- Set PRIMASK and run program
- Observe no more SysTick interrupts triggering handler
- Pause and clear PRIMASK
- Pause inside SysTick_Handler
- Set FAULTMASK
- Step out of SysTick_Handler
- Observe “clearing” of FAULTMASK
- Step out of SysTick_Handler, then set FAULTMASK
- Observe no more SysTick interrupts triggering handler
- Show usage of __disable_irq() & __enable_irq inside SysTick_Handler

Interrupt Service Routines

- Interrupt Service Routine (ISR)
- Characteristics of a good ISR
- Important considerations for ISRs in C
- Non-Nested Interrupts
- Nested Interrupts

Interrupt Service Routine (ISR)

- An ISR is a callback whose execution is triggered by the reception of an interrupt.
- The processor automatically suspends the execution of the regular program and starts the execution of a special predefined function called interrupt handler or interrupt service routine.
- After the interrupt handler completes, the processor goes back to the regular program



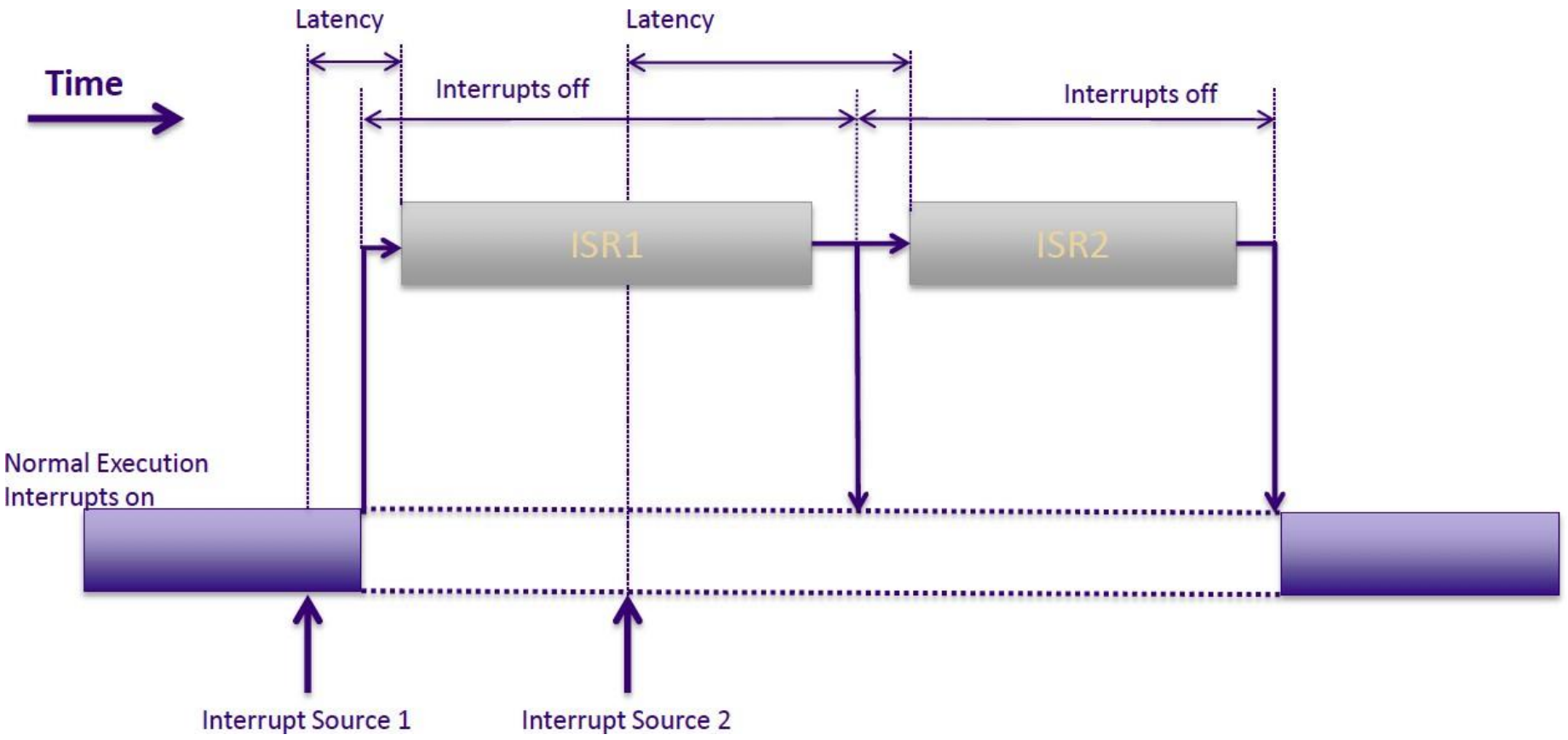
Characteristics of a Good ISR

- **Short**
  - Handles the time critical section only
  - Passes any other time-consuming non-time critical work to a non-interrupt context thread
- **Fast**
  - Preferably no function calls
  - Function call overhead
  - Is the callee well behaved? Must follow same characteristics of a good ISR.
- **Deterministic runtime**
  - Avoid nondeterministic loops or function calls
  - Beware of calling code that you don't know what's inside.
  - i.e. Must be able to calculate how long it will take to execute (to characterize interrupt latency)

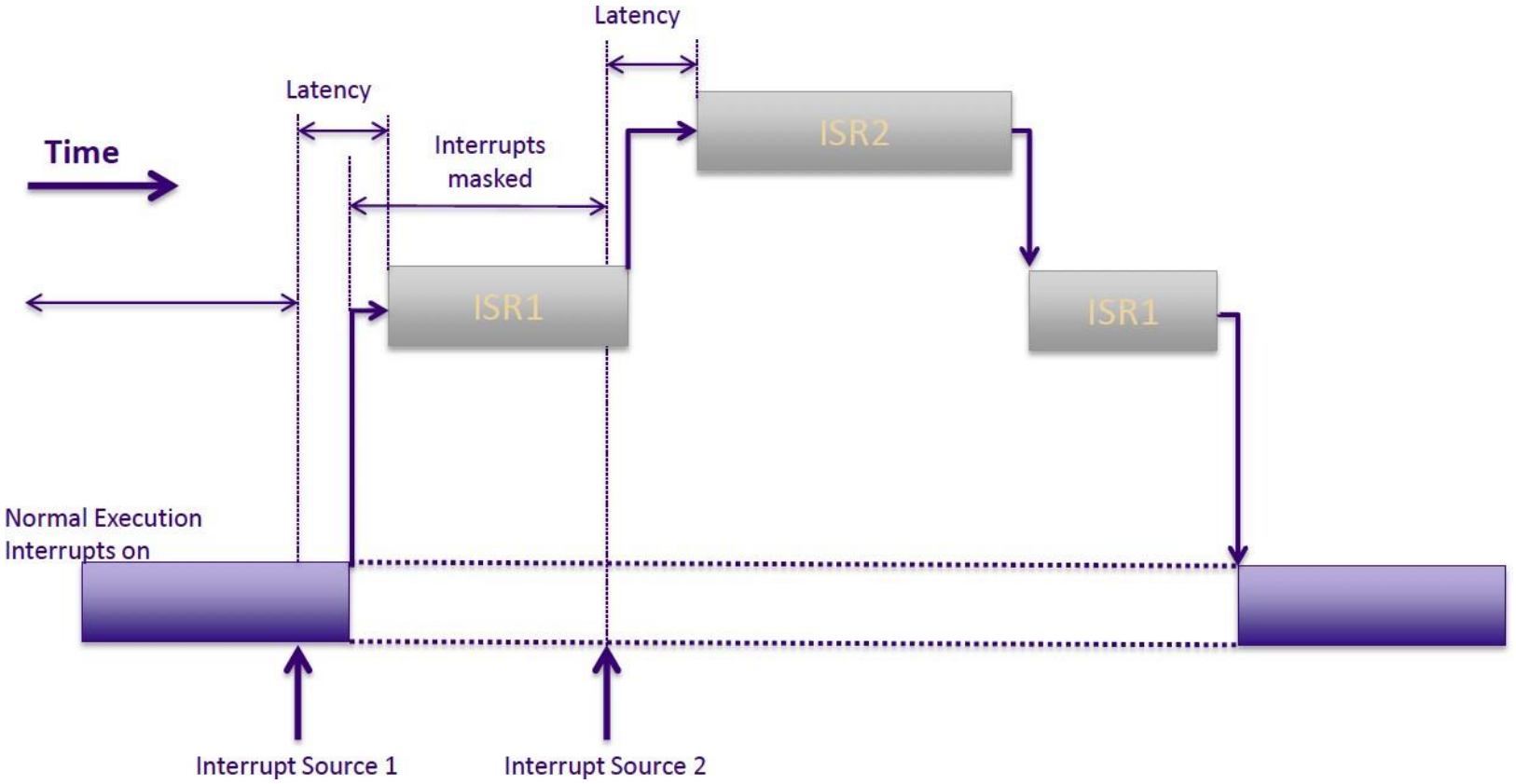
Important Considerations for ISR's in C

- ISRs take no parameter inputs, return no values
- Passing of parameters is done through other "thread safe" methods, IPCs (Inter-Process Communication: queues, mailboxes, etc.) or maybe protected global variables.
- Cannot call an ISR like a normal function
- CPU handles exceptions differently (CPU context saved/restored by hardware)

Non-Nested Interrupts



Nested Interrupts



BREAK 1

Realtime Operating System

- Operating System Terminology
- What is a Realtime Operating System
- Desktop vs RTOS based applications
- Preemptive vs Non-Preemptive RTOS
- When to use an RTOS
- The Scheduler
- Tasks & Data
- RTOS Services

Operating System Terminology

- A **system** is a regularly interacting or interdependent group of units forming an integrated whole.
- A **task** is a group of instructions that perform a function of a system. It is the main building block for software written for an RTOS environment.
- **Multi-tasking** is managing multiple tasks concurrently
- **Scheduler** attending to each of the tasks in a predetermined scheme called schedule:
  - Lines up the tasks in accordance with their priority
  - In any given time decide which task should perform and which task should pause
- **Priority** - importance of a task
- **Preemption** - allows higher priority task to pause a lower priority task in order to maintain system integrity

Realtime Operating Systems

- A **Realtime Operating System** (RTOS) is an operating system (OS) intended to serve **real-time** applications that process data as it comes in, typically without buffered delays.

Realtime Operating Systems

- Key factors in a Realtime Operating System are **minimal interrupt latency** and **minimal thread switching latency**.

- An RTOS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.

Realtime Operating System

- An RTOS typically has a small memory footprint and provides very fast context switching (the time required to switch from one task to another).

- Unlike OSs for personal computers or mobile computing devices (e.g., tablets), most embedded OS do not have any user interface, although user interface components (e.g., a GUI) can be added as application tasks running on the system.

Desktop vs RTOS based applications

- On a desktop computer the operating system takes control of the machine as soon as it is turned on and then lets you start your applications.
- You compile and link your applications separately from the operating system.

Desktop vs RTOS based applications

- In an embedded system, you usually link your application and the RTOS together.
- At boot-up time, your application usually gets control first, and then starts the RTOS.
- Thus, the application and the RTOS are much more tightly tied to one another than an application and its desktop operating system.

Desktop vs RTOS based applications

- Many RTOSes do not protect themselves as carefully from your application as do desktop operating systems.

- For example, whereas most desktop operating systems check that any pointer you pass into a system function is valid, many RTOSs skip this step in the interest of better performance.

- Of course, if the application is doing something like passing a bad pointer into the RTOS, the application is probably about to crash anyway; for many embedded systems, it may not matter if the application takes the RTOS down with it: the whole system will have to be rebooted any way.

Desktop vs RTOS based applications

- To save memory RTOSes typically include just the services that you need for your embedded system and no more.

- Most RTOSes allow you to configure them extensively before you link them to the application, letting you leave out any functions you don't plan to use.

- Unless you need them, you can configure away such common operating system functions as file managers, drivers, utilities, and perhaps even memory management.

Preemptive vs Non-Preemptive RTOS

- A **preemptive** RTOS will stop a lower-priority task as soon as the higher-priority task unblocks.

- A **non-preemptive** RTOS (aka **cooperative**) will only take the microprocessor away from the lower-priority task when that task blocks

- *References:*
  - Cooperative Multitasking
  - Cooperative Multitasking vs Preemptive Multitasking

When to use an RTOS

- An Embedded OS (or an RTOS) divides the available CPU processing time into a number of time slots and carries out different tasks in different time slots.
- Because the switching between time slots may happen hundreds of times per second, or more, it appears to the user that the processor executes several tasks in parallel.
- Many applications do not require an embedded OS at all.
- The key benefit of using an embedded OS is to provide a scalable way of enabling several concurrent tasks to run in parallel.
- If the tasks are all fairly short and don't overlap each other most of the time, you can simply use an interrupt-driven arrangement to support multiple tasks.

When to use an RTOS

- There are a number of factors to consider when deciding whether to use an embedded OS or not:
  - An embedded OS requires extra memory overhead.
  - An embedded OS requires execution time overhead. Processing time is required for context switching/task scheduling. Usually very small.
  - Some require license fees and/or royalty fees. Many others are free.
  - Some might only work with certain microcontroller devices or can be toolchain-specific. If portability is important then select an embedded OS which is supported on multiple platforms.
- In general, as software code gets more complex, use of an embedded OS can make handling of multiple tasks much easier.

Tasks

- A “**task**” is the basic building block of software written on top of an RTOS.
- A task is simply a subroutine.
- Each task has its own stack.
- The RTOS provides a function that starts a task, telling it which subroutine is the starting point for each task and some other parameters; such as, the task’s priority, and where the RTOS should find memory for the task's stack
- Most RTOSes allow you to have as many tasks as you could reasonably want.

Tasks

- Each task in an RTOS is always in one of three states:

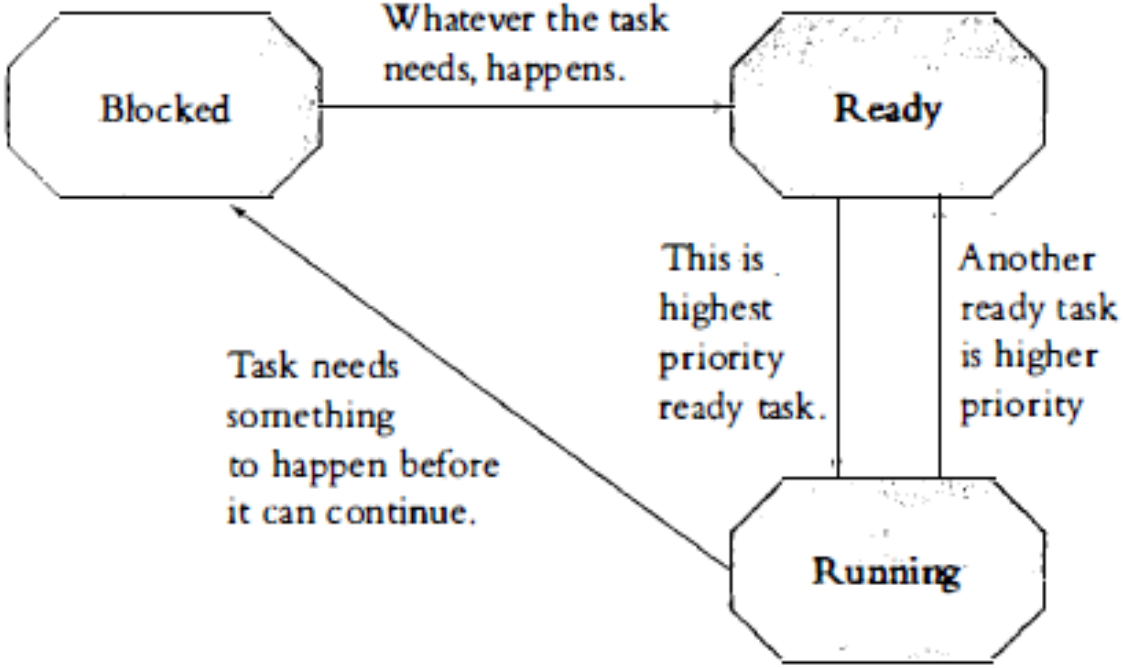
1. **Running (executing on CPU)**: which means that the microprocessor is executing the instructions that make up this task. Unless we are on a multiprocessor system, there is only one microprocessor; hence only one task that is in the running state at any given time.

2. **Ready (ready to be run)**: which means that the task is ready to run but some other task is already in the running state. Any number of tasks can be in this state.

3. **Blocked (waiting for an event)**: which means that the task does not have anything to do right now, even if the microprocessor becomes available. Tasks get into this state because they are waiting for some external event. For example, a task that handles data coming in from a network will have nothing to do when there is no data. A task that responds to the user when he presses a button has nothing to do until the user presses the button. Any number of tasks can be in this state as well.

Tasks and States

Figure 6.1 Task States



The Scheduler

- A part of the RTOS called the scheduler keeps track of the state of each task and decides which one task should go into the running state.
- Unlike the scheduler in Unix or Windows, the schedulers in most RTOSes are entirely simpleminded about which task should get the processor.
- They look at priorities you assign to the tasks, and among the tasks that are not in the blocked state, the one with the highest priority runs, and the rest of them wait in the ready state.

Tasks and Data

- Each task has its own private context.
- This includes the register values, a program counter, and a stack.
- However, all other data-global, static, initialized, uninitialized, and everything else-is shared among all the tasks in the system.

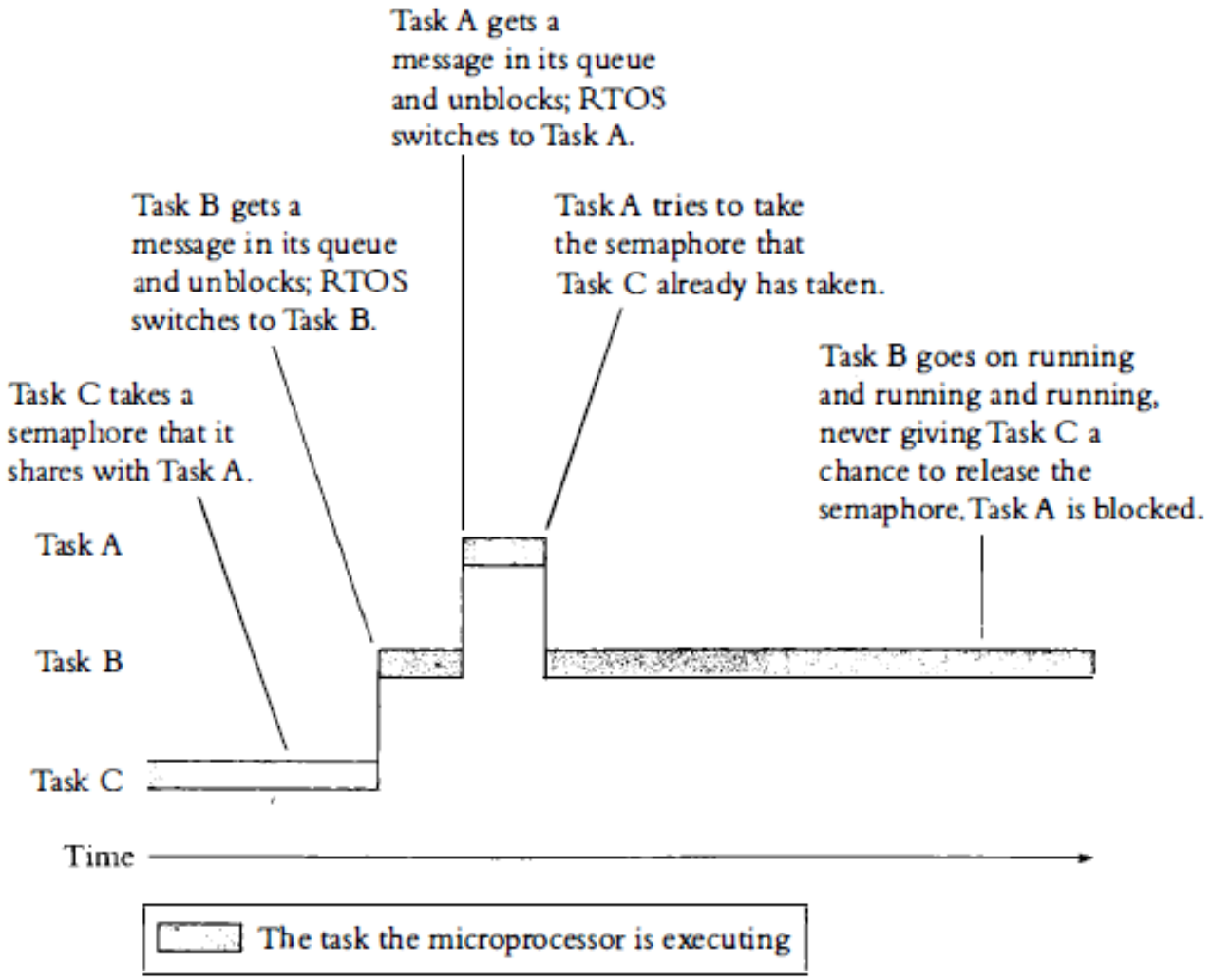
Shared data problem revisited

- With switching between multiple tasks and with tasks sharing the same data, we hit the shared data problem again (as we saw with interrupts).
- **Reentrancy**: A reentrant function is a function that can be called by more than one task and that will always work correctly.
- Three rules to decide if a function is re-entrant:
  - A reentrant function may not use variables in a non-atomic way unless they are stored on the stack of the task that called the function or are otherwise the private variables of that task.
  - A reentrant function may not call any other function that is not reentrant.
  - A reentrant function may not use the hardware in a non-atomic way; as we saw with the GPIO registers (doing a read-modify-write) and their aliased bit-banding registers (during a direct write).

Shared data problem

- Possible methods to protect against the shared data problem.
- Temporarily disabling interrupts.
- Using Semaphores/Mutexes (constructs in the OS to signal readiness between tasks or lock shared resources).
  - Priority inversion is a potential problem.
  - Forgetting to take or release a semaphore/mutex or using the wrong one are other common ways to cause yourself problems.
- Use message passing constructs in the OS between tasks that need to share data (MailBoxes, Queues, Pipes…etc.)

Figure 6.17 Priority Inversion



Priority inversion

Priority inheritance

- Priority inheritance is one way to avoid the priority inversion problem.
- If a higher-priority-task attempts to acquire a semaphore held by a lower-priority-task
- The priority of the lower-priority-task is temporarily raised to the value of the higher priority task until the resource is released.

RTOS Services

- Tasks must be able to communicate with one another to coordinate activities and to share data.

- Most RTOSs offer some combination of services; such as message queues, mailboxes, and pipes, for this purpose.

- RTOSs offer events which are one-bit flags with which tasks signal one another. Events can be formed into groups, and a task can wait for a combination of events within a group.

- The specific features of these services are RTOS-dependent.

RTOS Timing services

- Most RTOSs maintain a heartbeat timer that interrupts periodically and that is used for all the RTOS timing services.
- The interval between heartbeat timer interrupts is called the system tick.
- The most common RTOS timing services are these:
  - A task can block itself for a specified number of system ticks (ex: delay within a task)
  - A task can limit how many system ticks it will wait for a semaphore, a queue, etc (ex: timeout used when awaiting on data/signals from other tasks)
  - Your code can tell the RTOS to call a specified function after a specified number of system ticks.

Memory allocation

- Memory allocation is more critical in a real-time operating system.
- There cannot be memory leaks.
- Even though many RTOSs offer the standard malloc and free functions, embedded engineers often avoid them because they are relatively unpredictable.
- It is more common to use memory allocation based on a pool of fixed-size buffers.
- Or whenever possible, all required memory allocation is specified statically at compile time.
  - To avoid memory fragmentation
    To avoid unpredictable response times

OS Services Summary

- Task Management: Create/Delete tasks
- Time Management: Delay/GetTime/SetTime
- Task Synchronization/Communication/Event Handling:
  - Semaphores (to protect shared data)
  - Mutexes (to protect shared data)
  - Message Mailboxes (to communicate single message)
  - Message Queues (to communicate multiple messages)
  - Event Flags (to synchronize between multiple tasks)
- Memory Management (within a specified memory pool)

The second course of the UW Embedded Certificate will cover all these OS services in detail using the uC-OSII Realtime Operating System.

BREAK 2

Pulse Width Modulation

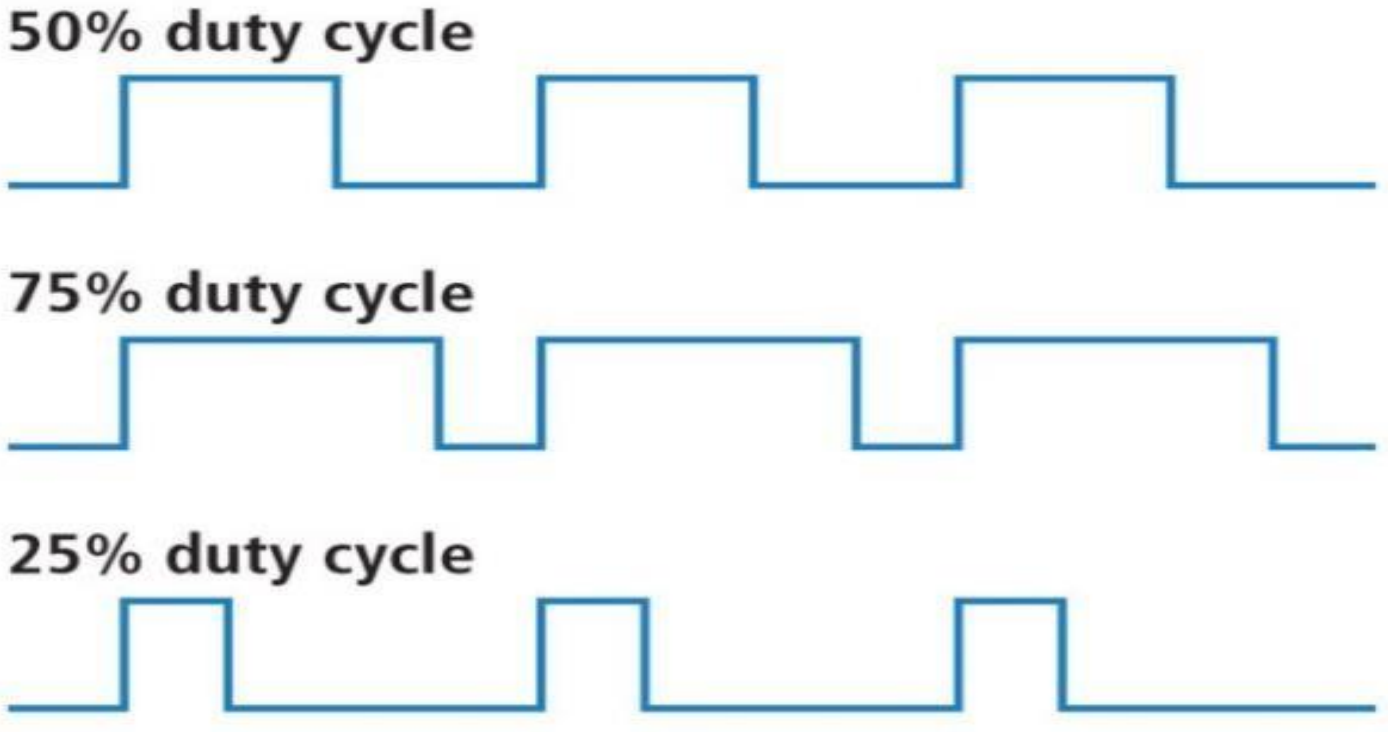
- What is PWM
- Hardware Timers
- Timer Input Capture Mode
- Generating PWM with Timer Output
- Multi Channel Outputs

Pulse Width Modulation

- Pulse width modulation is a method of controlling the average power delivered by an electrical signal.
- It does so by switching the on-and-off phases of a digital signal quickly and varying the width of the "on" phase or **duty cycle**.
- The longer the switch is on compared to the off periods, the higher the total power supplied to the load.
- **The duty cycle** is expressed as the percentage of being fully on (which is 100% duty cycle).

Pulse Width Modulation

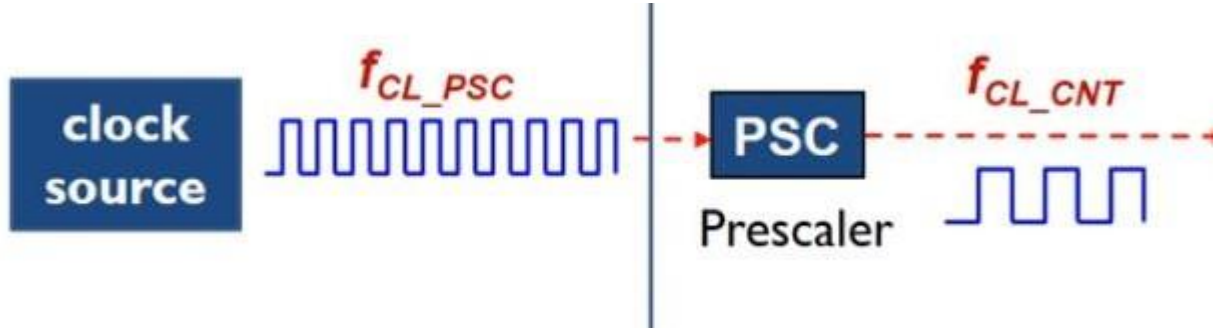
- Can be used to fine control different electronic devices. For example the **brightness of an LED** Or the speed of a stepper motor.



50%, 75%, and 25% Duty Cycle Examples

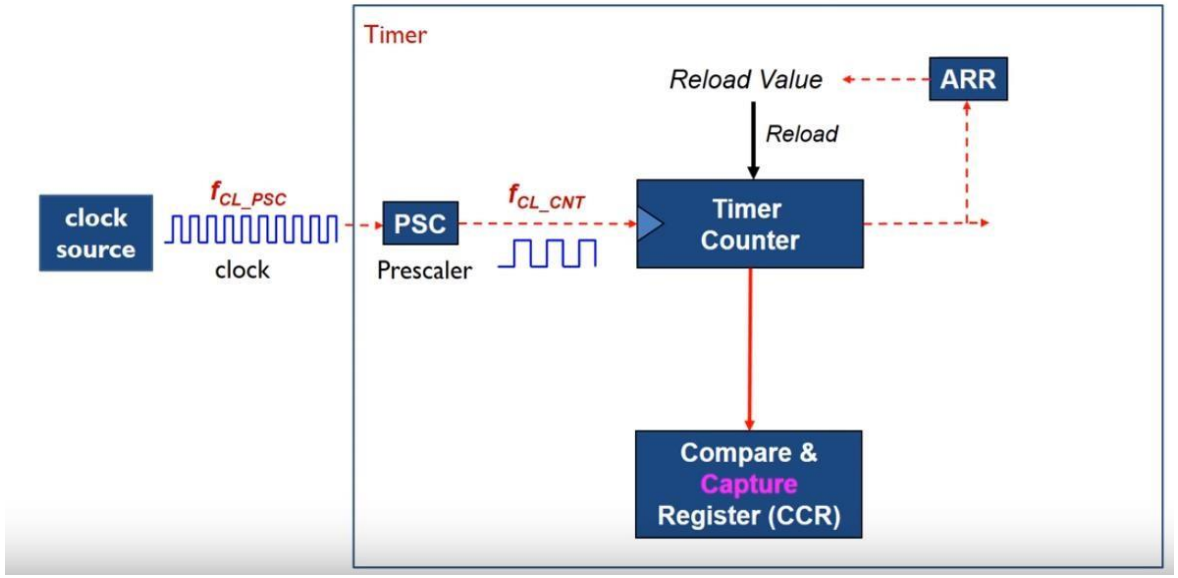
Hardware Timers

- A hardware timer is a component built within the MCU chip.
- Each timer has a special register which is called the timer counter. The counter runs freely if enabled. It automatically counts upwards or downwards driven by a clock source.
- So it is hardware not software which keeps repeatedly incrementing or decrementing the counter value for each rising edge of the clock signal.
- By modifying a PSC(prescaler) register we can feed the counter with clock pulses at the rate we desire.



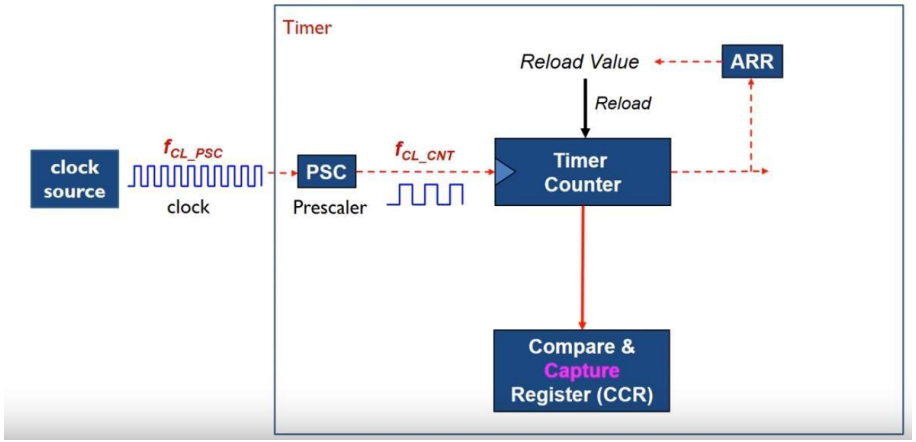
Timer Input in Compare & Capture Mode

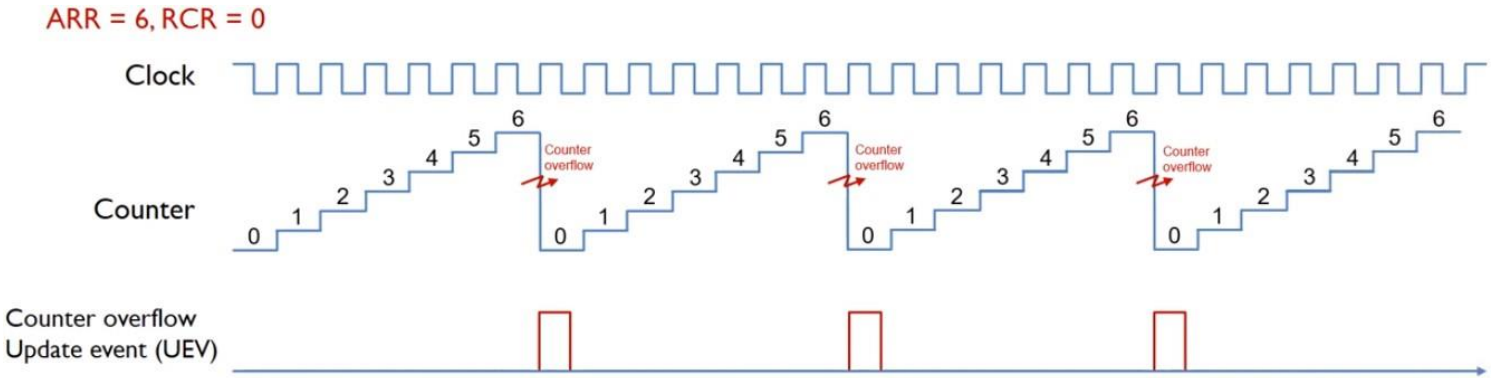
- A timer can be used to capture when a specific external event occurs such as a rising edge of an external signal.
- Hardware automatically copies the current value of the timer counter to the compare and capture register CCR.
- We can use such ability to measure the timing information of a signal event such as rising edge or falling edge.
- To measure the external signal interval two consecutive captures are needed so we can calculate the period or the pulse width by subtracting these two CCR values.



Generating PWM with Timer Output

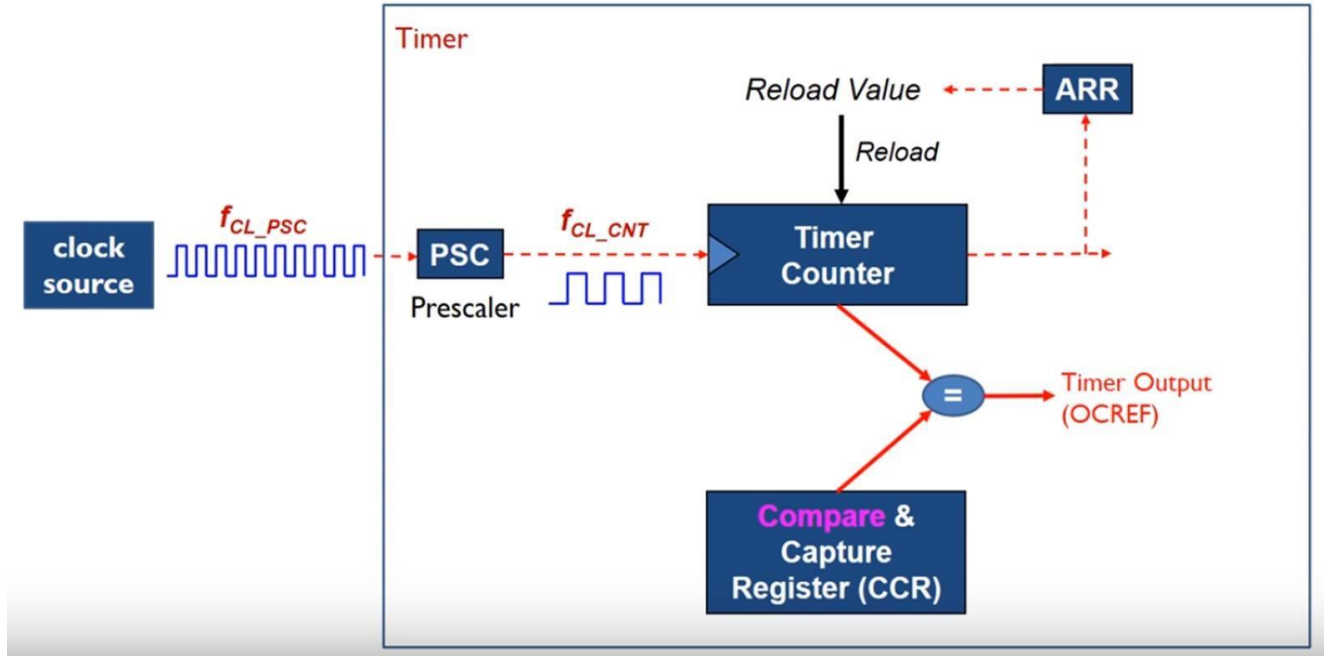
- The auto reload register ARR holds the maximum counter value
- For example, when the counter reaches six it rolls over and is reset. When the counter resets it triggers a counter overflow and an update event (UEV). After that a new period starts. The counting period is determined by the auto reload value as well as the clock period.





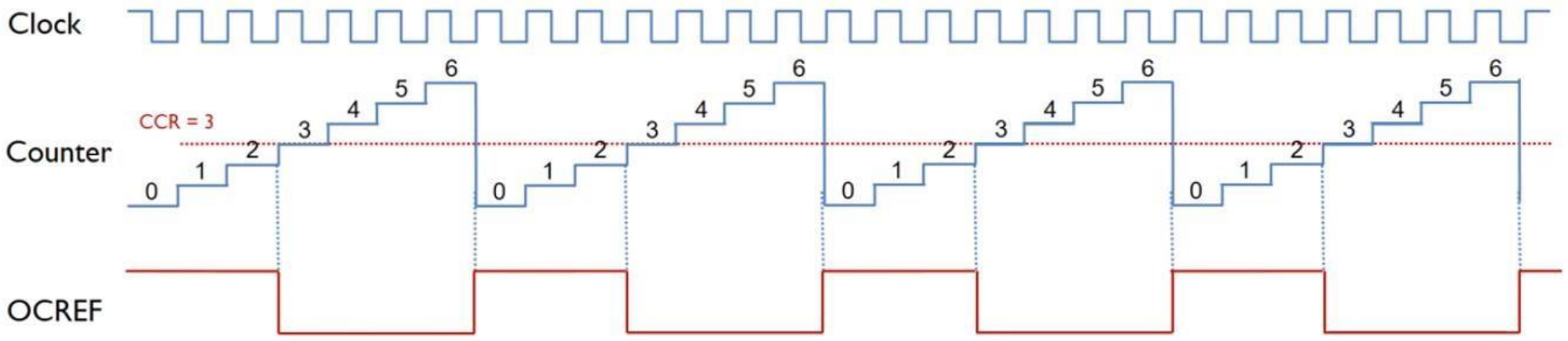
Generating PWM with Timer Output

- When a timer is configured to generate an output signal it constantly compares the free running counter with a value stored in a special register called the compare and capture register (CCR).
- In the PWM mode the timer controls the output of 1 or more output channels.
- When the counter value reaches 0, maximum or a **compare** value defined for each channel, the output value of the channel can be changed.



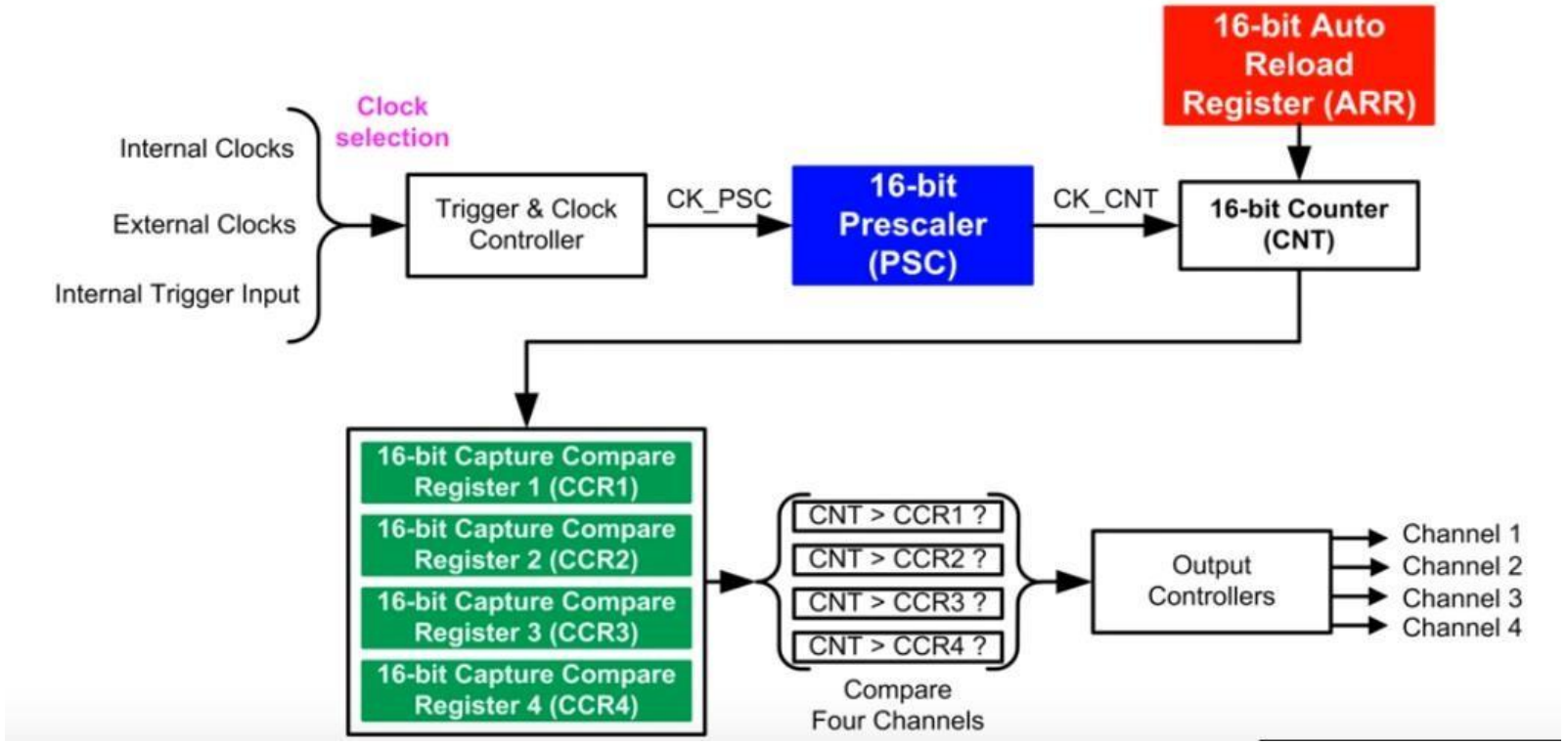
Generating PWM with Timer Output

Software can generate a PWM signal with a frequency determined by the value of the **TIMx_ARR** register and a duty cycle determined by the value of the **TIMx_CCRx** register.



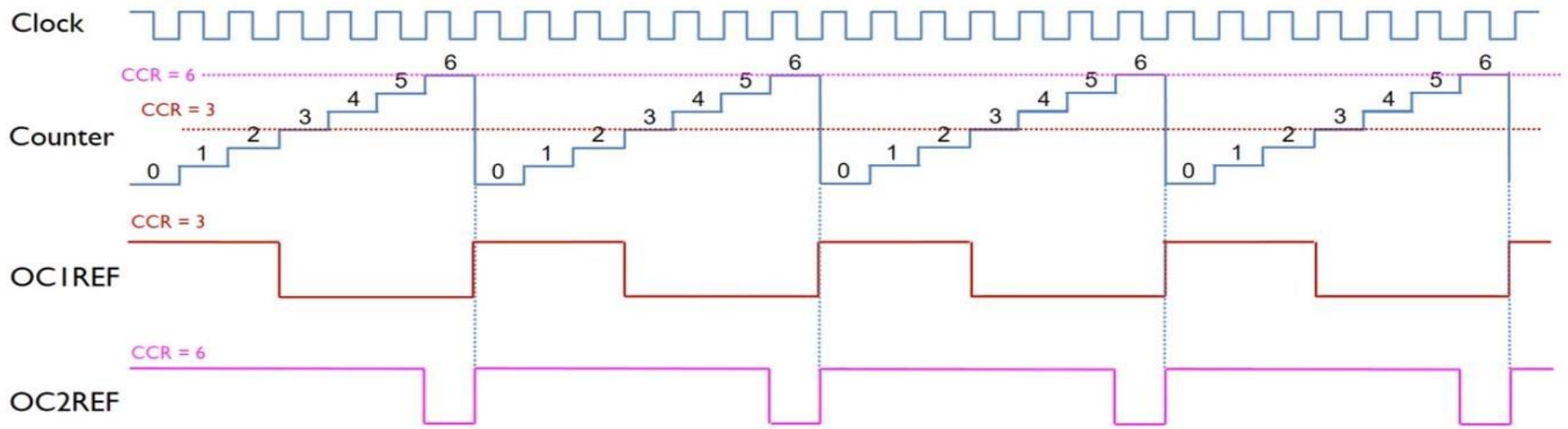
Multi Channel Outputs

Typically a single timer can generate up to four PWM signals with independent duty cycles and identical frequency.



Multi Channel Outputs

Typically each timer has four channels. Each channel has its own compare and capture register. These four channels share the timer counter and the auto reload register ARR. Therefore these PWM outputs have exactly the same period. However their duty cycle can be different because the value of these CCR registers can differ from each other.



Examples & References

- Examples:
  - **https://www.waveshare.com/wiki/STM32CubeMX_Tutorial_Series:_PWM**
  - https://visualgdb.com/tutorials/arm/stm32/pwm/

- References:
  - https://www.analogictips.com/pulse-width-modulation-pwm/
  - https://en.wikipedia.org/wiki/Stepper_motor
  - https://www.motioncontrolonline.org/blog-article.cfm/What-Kinds-of-Applications-are-Best-for-Stepper-Motors/76

Demo Video

Demo the use of a PWM channel output to modify the brightness of the user LED on our NUCLEO-F401RE

Assignment 08

Suggested Reading

- ***"The Definitive Guide to ARM Cortex M3 & M4" by Joseph Yiu (Third Edition)***
  - Cha 4.5: Exceptions and interrupts
  - Cha 7: Exceptions and Interrutps
  - Cha 19.1: Introduction to Embedded OSes
- ***"An Embedded Software Primer" by David E. Simon***
  - Cha 6: Introduction to Real-Time Operating Systems
  - Cha 7: More Operating System Services
- ***"STM32F401 Reference Manual – RM0368"***
  - Cha 10.1: Nested Vector Interrupt Controller
  - Cha 10.2: External interrupt/event controller
  - Cha 10.3: EXTI Registers
  - Cha 13:General-purpose timers

Interrupt Tutorials by Dr. Yifeng Zhu

- ARM Cortex-M Interrupts
- Interrupt Enable & Priority
- External Interrupts (EXTI)