Microprocessors

Fall 2020

6. Exceptions and Interrupts

Tentative Weekly Schedule

- Week x1 - Introduction to Course
- Week x2 - Architecture
- Week x3 - Assembly Language Introduction
- Week x4 - Assembly Language Usage, Memory and Faults
- Week x5 - Embedded C and Toolchain
- **Week x6 - Exceptions and Interrupts**
- Week x7 - Timers
- Week x8 - Modulation
- Week x9 - Serial Communications I
- Week xA - Serial Communications II
- Week xB - Analog Interfacing
- Week xC - DMA
- Week xD - RTOS
- Week xE - Wireless Communications

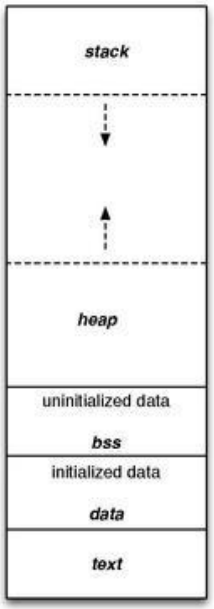
Review: Basic C program structure

```c
#include "stm32g0xx.h"
#define LEDDELAY    1600000

int main(void);
void delay(volatile uint32_t);

int main(void) {
    RCC->IOPENR |= (1U << 2);
    GPIOC->MODER &= ~(3U << 2*6);
    GPIOC->MODER |= (1U << 2*6);
    GPIOC->ODR |= (1U << 6);
    while(1) {
        delay(LEDDELAY);
        GPIOC->ODR ^= (1U << 6);
    }
    return 0;
}

void delay(volatile uint32_t s) {
    for(; s>0; s--);
}
```

- Register addresses for specific microcontroller
- Preprocessor directives
- Function declarations
- Bitwise operations
- Infinitive loop
- Function definitions

Review: Memory Layout for Program

- **text** segment holds the code and usually found in ROM. It is read-only memory (when program is executing) and code can be executed from here
- **data** segment holds the initialized code and usually found in RAM. It is read-write memory
- **bss** segment holds the uninitialized and zero-initialized code (all initialized to zero)
- **heap** segment holds the dynamically allocated memory segments (**malloc**). Grows towards larger addresses
- **stack** segment holds the local variables and registers for functions as temporary storage area. Grows towards smaller addresses



Review: Where the variables will be placed

- **text** segment
  - `e - 4` bytes
- **data** segment:
  - `a, c, g - 12` bytes
- **bss** segment
  - `b, d, h - 16` bytes
- **stack** segment:
  - `j` (within fun function)
  - `f, k, s` (within main function)

```c
int a = 21;
char b[8] = {0};
static int c = 14;
char d[4];
const unsigned int e = 1337;

void fun() {
    static int g = 3;
    static int h = 0;
    int j = 0;
    ++g + j + ++h;
}

int main(void) {
    int f;
    int k = 4;
    char s[10];
    fun();
    for(;;);
    return 0;
}
```


Review: GNU Toolchain

- GNU GCC Toolchain is a popular toolchain produced by GNU Project.
- ARM is maintaining a GNU toolchain targeted at embedded ARM processors.
- ARM GNU Embedded toolchain includes
  - **gcc** - GNU Compiler Collection
  - **binutils** - a suite of tools including linker, assembler and other tools
  - **gdb** - GNU debugger, a code debugging tool
  - **newlib** - C library for embedded systems (such as stdlib.h, math.h, stdio.h, time.h, ...)

Review: Example - blinky project - 3 files

`blinky.c`, `system_stm32g0xx.c`, `startup_stm32g031k8tx.s`

```
arm-none-eabi-gcc -DSTM32G031xx -mcpu=cortex-m0plus -mthumb -O0 -std=gnu11 -g -gdwarf-2 -MMD
-MP -MF"Debug/main.d" -MT"Debug/main.d" -fno-common -Wall -Wextra -pedantic
-Wmissing-include-dirs -Wsign-compare -Wcast-align -Wconversion -fsingle-precision-constant
-fomit-frame-pointer -ffunction-sections -fdata-sections --specs=nano.specs -I../include -c main.c
-o Debug/main.o
```

```
arm-none-eabi-gcc -DSTM32G031xx -mcpu=cortex-m0plus -mthumb -O0 -std=gnu11 -g -gdwarf-2 -MMD
-MP -MF"Debug/main.d" -MT"Debug/main.d" -fno-common -Wall -Wextra -pedantic
-Wmissing-include-dirs -Wsign-compare -Wcast-align -Wconversion -fsingle-precision-constant
-fomit-frame-pointer -ffunction-sections -fdata-sections --specs=nano.specs -I../include -c main.c
-o Debug/main.o
```

```
arm-none-eabi-gcc -DSTM32G031xx -mcpu=cortex-m0plus -mthumb -O0 -std=gnu11 -g -gdwarf-2 -MMD
-MP -MF"Debug/system_stm32g0xx.d" -MT"Debug/system_stm32g0xx.d" -fno-common -Wall -Wextra
-pedantic -Wmissing-include-dirs -Wsign-compare -Wcast-align -Wconversion
-fsingle-precision-constant -fomit-frame-pointer -ffunction-sections -fdata-sections
--specs=nano.specs -I../include -c ../include/system_stm32g0xx.c -o Debug/system_stm32g0xx.o
```

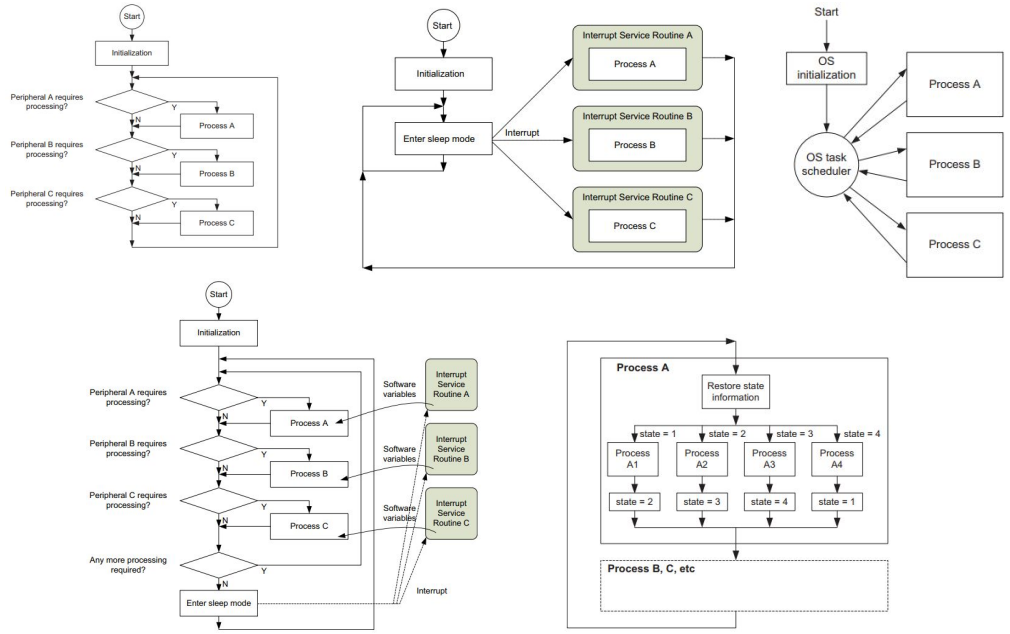
Linking stage

```
arm-none-eabi-gcc Debug/main.o Debug/system_stm32g0xx.o Debug/startup_stm32g031k8tx.o
-mcpu=cortex-m0plus -mthumb --specs=nano.specs -Wl,--gc-sections -Wl,-Map=Debug/blinky.map
-Wl,--cref -T../linker/STM32G031K8Tx_FLASH.ld -lc -o Debug/blinky.elf
```


Review: Automating tasks: make utility

- Well, memorizing all these is hard, and typing them is cumbersome. Thankfully there is a way to automate all these tasks with a utility: **make**
- **make** utility automatically determines which pieces of a large program need to be recompiled, and issues commands to re-compile them.
- All the steps can be included and organized in a file called **`makefile`** by defining a set of rules and giving them order.
- Usually IDEs auto generate these makefiles based on your configuration in the background and execute them when you hit compile button.

Review: Software Program Flows



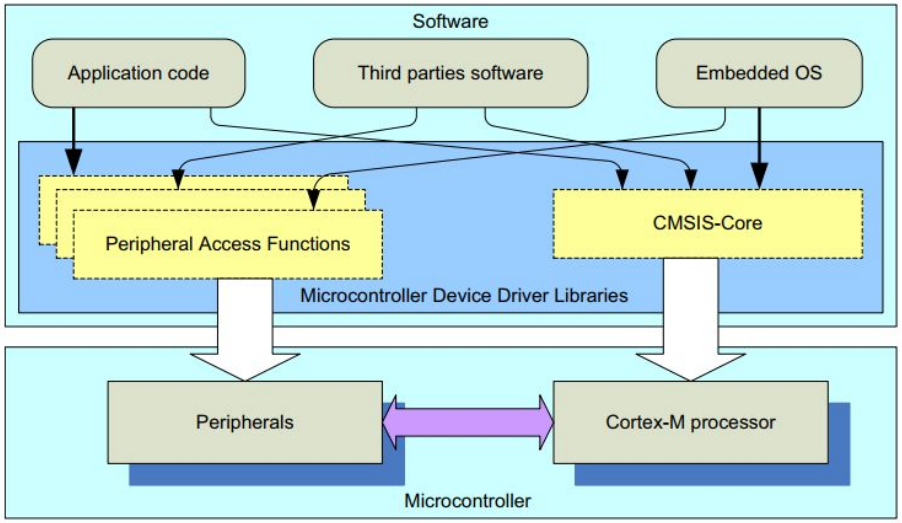
CMSIS

- **Cortex Microcontroller Software Interface Standard (CMSIS)** is a **software framework** to provide a **standardized** software interface to the processor features like interrupt and system control functions.
- Provided by ARM and targets most Cortex-M processors
- Can be used with different toolchains and IDEs
- Consist of multiple packages

CMSIS Projects

CMSIS-CORE - Standardized API for the Cortex-M processor core and peripherals.
CMSIS-DSP - DSP library collection
CMSIS-RTOS Common API for RTOS along with a reference implementation based on RTX.
CMSIS-NN Collection of efficient neural network kernels
CMSIS-Driver Generic peripheral driver interfaces for middleware.

CMSIS-CORE



How can we detect when a button is pressed?

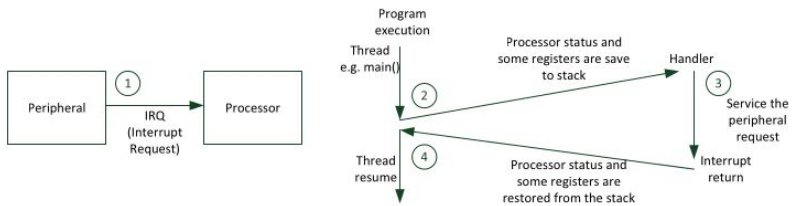
- One option is to check regularly in your main loop, called **polling**
  - If the main loop is long, you might miss it
  - Not scalable
  - All the reading of the port makes it both slow and wasteful of CPU time
- A better option is to use **a special hardware** in the MCU to detect the press and **alert us**
  - A designated piece of software can run afterwards, becomes efficient
  - Scalable since our code size does not matter

Exceptions and Interrupts

- Exceptions (and Interrupts) are **events** that cause a request to **change in program flow** outside of the normal programmed sequence to **execute a piece of software** to service the request.
- The process involves suspending the current executing task, or waking up from sleep mode, and executing the piece of software called **Exception Handler** to service the request
- **Interrupts** are **a type of exception**

Handling Interrupts (and Exceptions)

1. peripheral **generates** an **i**nterrupt **req**uest (IRQ) to the processor
2. processor **suspends** current task saving processor state on the stack
3. processor **executes i**nterrupt handler **s**ervice **r**outine (ISR) after locating the starting address from vector table to service the peripheral (handle interrupt)
4. processor **resumes** the suspended task after completing servicing the IRQ and restoring the processor state.



Interrupts

- **Hardware-triggered asynchronous software routine**
  - **Triggered** by hardware signal from peripheral or external device
  - **Asynchronous** - can happen anywhere in the program
  - **Software routine** runs in response to interrupt
- Fundamental mechanism of microcontrollers
  - Provides **efficient event-based processing** rather than polling
  - Provides **quick response to events** regardless of program state, complexity and location
  - Allows embedded systems to be **responsive** without an operating system

Example C program

```c
static int count = 0;

void buttonISR(void) {
    // Toggle LED
    GPIOC->ODR ^= (1 << 6);
    count++;
}

int main(void) {
    // Initialize LED and Buttons
    // Initialize interrupt

    while(1) {
        // Do nothing
    }
   return 0;
}
```

- **No calls** to the function
- Hardware **automatically jumps** there to **service** the interrupt
- If variables need to be **shared**, they should be declared **globally**
  - Also this requires careful considerations both in terms of optimization and race conditions

Exception 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** - the exception is being serviced by the processor, but not yet completed
  - An interrupt service handler can interrupt the execution of another interrupt service handler. In this case **both exceptions are in active state**
- **Active and Pending** - the exception is being serviced by the processor and there is a **pending exception from the same source**

Exception Types in Cortex M0+

Non-Maskable Interrupt (NMI) - Cannot be disabled and has the highest priority.
HardFault - used for handling fault conditions during program execution
SVCall - Exception that happens when SVC instruction is executed usually when allowing applications to access system services in OSes
Pendable Service Call - Commonly used to schedule system operations to be carried out when high priority tasks are completed by OS
System Tick Timer - A general timer for mostly OS uses such as context switching
Interrupts - usually tied to peripherals. 32 total.

Exception Types in Cortex M0+

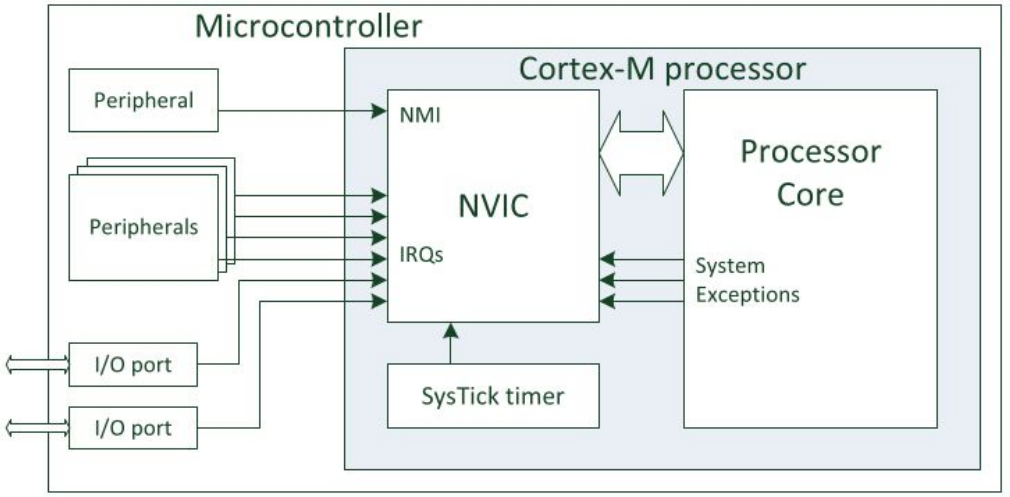
Exception number	Exception type	Priority	Descriptions
1	Reset	-3 (Highest)	Reset
2	NMI	-2	Non-Maskable Interrupt
3	HardFault	-1	Fault handling exception
4–10	Reserved	NA	–
11	SVCall	Programmable	Supervisor call via SVC instruction
12–13	Reserved	NA	–
14	PendSV	Programmable	Pendable request for system service
15	SysTick	Programmable	System Tick Timer
16	Interrupt #0	Programmable	External Interrupt #0
17	Interrupt #1	Programmable	External Interrupt #1
…	…	…	…
47	Interrupt #31	Programmable	External Interrupt #31

- Microcontroller vendors connect peripheral IRQs to specific numbers. (i.e vector table locations)
- Programmable meaning - priorities can be arranged. (i.e which can interrupt which)

Nested Vectored Interrupt Controller

- NVIC is a programmable unit that allows software to manage interrupts and exceptions. It has a number of memory mapped registers for the following
  - **Enabling or disabling** of each of the interrupts
  - Defining the **priority levels** of each interrupts and some of the system exceptions
  - Enabling the software to access the **pending status of each interrupt**, including the capability to trigger interrupts by setting pending status in software.
- NVIC registers can only be accessed in privileged state and must be aligned to 32-bit transfers

Nested Vectored Interrupt Controller



NVIC Registers - en / dis and pending

`0xE000E100 SETENA` - Set enable for interrupt 0 to 31
`0xE000E180 CLRENA` - Clear enable for interrupt 0 to 31
`0xE000E200 SETPEND` - Set pending for interrupt 0 to 31
`0xE000E280 CLRPEND` - Clear pending for interrupt 0 to 31

- Each bit corresponds to an interrupt with the same number
- Having separate registers help prevent **race conditions**
- CMSIS calls are

```
void NVIC_EnableIRQ(IRQn_Type IRQn);
void NVIC_DisableIRQ(IRQn_Type IRQn);
void NVIC_SetPendingIRQ(IRQn_Type IRQn);
void NVIC_ClearPendingIRQ(IRQn_Type IRQn);
```

where IRQn of 0 refer to Interrupt #0

Vector Table

- Exception handling is **vectored**, which means the processor's hardware automatically determines which exception to service
- Starting addresses are fetched from **Vector Table** which is simply a lookup table in memory
- NVIC generates the vector for processor to look up the starting address

Memory Address		Exception Number
0x0000004C	Interrupt#3 vector	19
0x00000048	Interrupt#2 vector	18
0x00000044	Interrupt#1 vector	17
0x00000040	Interrupt#0 vector	16
0x0000003C	SysTick vector	15
0x00000038	PendSV vector	14
0x00000034	Not used	13
0x00000030	Not used	12
0x0000002C	SVC vector	11
0x00000028	Not used	10
0x00000024	Not used	9
0x00000020	Not used	8
0x0000001C	Not used	7
0x00000018	Not used	6
0x00000014	Not used	5
0x00000010	Not used	4
0x0000000C	HardFault vector	3
0x00000008	NMI vector	2
0x00000004	Reset vector	1
0x00000000	MSP initial value	0

Vector Table - relocation

- LSB needs to be set to one to indicate Thumb state.
- Cortex-M0+ processor has a vector table relocation feature to define it at a different part of memory space using a hardware register called **VTOR** (Vector Table Offset Register)
- VTOR must be a multiple of 0x100 (256 bytes)

Memory Address		Exception Number
0x0000004C	Interrupt#3 vector	19
0x00000048	Interrupt#2 vector	18
0x00000044	Interrupt#1 vector	17
0x00000040	Interrupt#0 vector	16
0x0000003C	SysTick vector	15
0x00000038	PendSV vector	14
0x00000034	Not used	13
0x00000030	Not used	12
0x0000002C	SVC vector	11
0x00000028	Not used	10
0x00000024	Not used	9
0x00000020	Not used	8
0x0000001C	Not used	7
0x00000018	Not used	6
0x00000014	Not used	5
0x00000010	Not used	4
0x0000000C	HardFault vector	3
0x00000008	NMI vector	2
0x00000004	Reset vector	1
0x00000000	MSP initial value	0

Exception Priority Levels

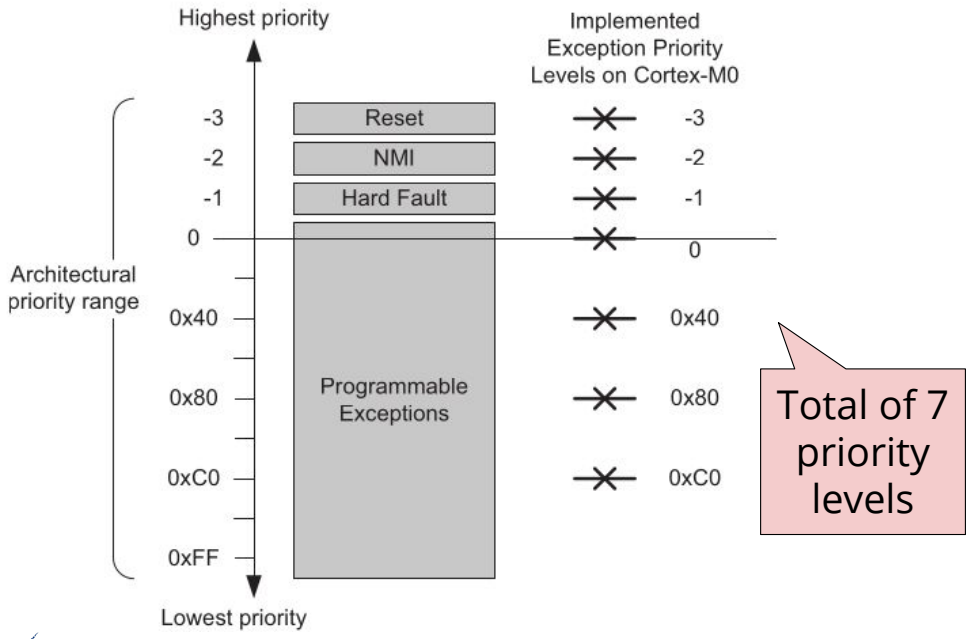
- The priority level is used to decide whether the exception **will happen or delayed**.
- Cortex-M0+ processors support **three** **fixed highest priority** levels for three system exceptions (**Reset, NMI, and HardFault**) and **four** **programmable levels** for all other exceptions including interrupts.
- The priority level configuration registers are 8-bit wide, but **only the two MSBs** are implemented.

Exception Priority Levels

- Since bits 5-0 are not implemented, they are always zero, and writes are ignored
- Possible levels are **0x00, 0x40, 0x80, 0xC0**
  - 0x00 the highest priority
  - 0xC0 the lowest priority
- Other Cortex-M processors implement more bits for priority

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Implemented		Not implemented, read as zero					

Available Priority levels of Cortex-M0+



NVIC Registers - priority level

- Each interrupt has an associated 1-byte priority level register.
- NVIC registers can only be accessed using word-size transfers - so four priority level registers at the same time.
- Unimplemented bits are read as zero.

Bit	31 30	24	23 22	16	15 14	8	7 6	0
0xE000E41C	31		30		29		28	
0xE000E418	27		26		25		24	
0xE000E414	23		22		21		20	
0xE000E410	19		18		17		16	
0xE000E40C	15		14		13		12	
0xE000E408	11		10		9		8	
0xE000E404	7		6		5		4	
0xE000E400	IRQ 3		IRQ 2		IRQ 1		IRQ 0	

NVIC Registers - priority level

```
NVIC->IP[IRQn >> 2] = (NVIC->IP[IRQn >> 2] & ~(0xFF
<< ((IRQn & 0x3) * 8)) | (((priority << 6) & 0xFF)
<< ((IRQn & 0x3) * 8)) );
```

- CMSIS calls are

```
void NVIC_SetPriority(IRQn_Type IRQn,
uint32_t priority);
uint32_t NVIC_GetPriority(IRQn_Type IRQn);
```

  where IRQn of 0 is Interrupt #0, and prio is between 0 - 3
- Priority levels should be set before the interrupt is enabled.
- No dynamic priority level change is allowed in ARMv6-M
  - Disable the interrupt, change priority level, enable the interrupt.

Exception Handling wih priority levels

When an exception happens:

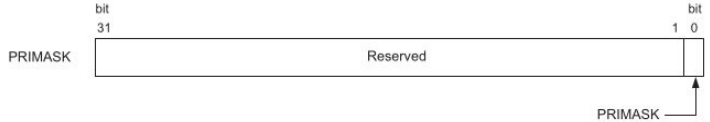
- If no exception running, it will be serviced normally
  - The process of switching from a current running task to an exception handler is called **preemption**
- If already an exception is serviced, and the new exception has higher priority, then preemption will happen, so new exception will be serviced
  - This is also called **Nested Interrupt / Exception**
- If already an exception is serviced, and the new exception has lower or equal priority, then it will wait for completion of the current serviced exception

If two exception happens at the same time with same priority levels, lower **exception type number** will go first

Acceptance of Exception Request

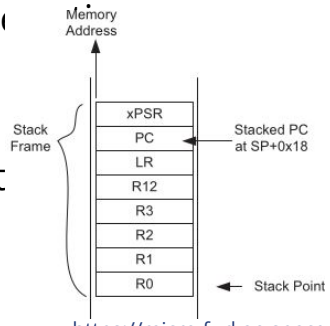
The processor accepts an exception if the following conditions are satisfied:

- The processor is **not halted** for debugging
- The exception **has to be enabled**
- The processor is not running an exception handler of **same or higher priority**
- The exception is **not blocked** by the **PRIMASK** interrupt masking register

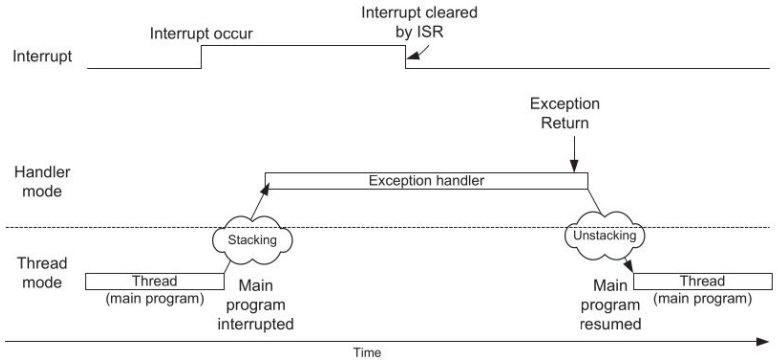


Stacking and Unstacking

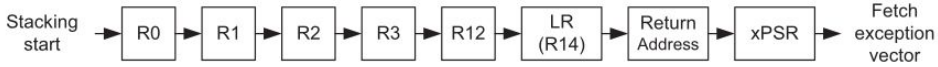
- In order to resume from servicing an interrupt, processor state must be saved. This is also called **saving the context**
- **Stacking** is the operation of saving some of the processor state in the stack.
  - For Cortex-M0+, this is 8 words. **R0-R3, R12, LR, PC, and xPSR** are pushed to th stack memory automatically.
- Link register is then updated to a special value used during except return called **EXC_RETURN**
- **Unstacking** is the reverse op.



Stacking and Unstacking

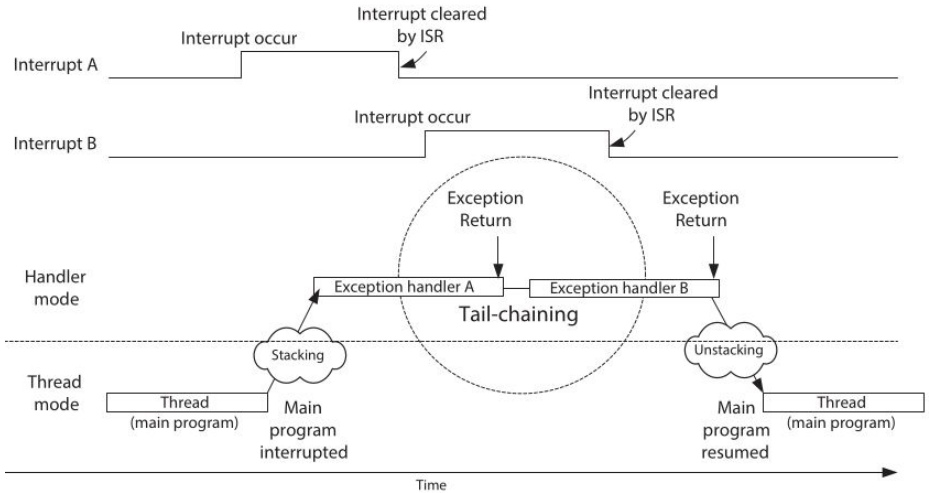


- Other registers (R4-R11) needs to be saved by software if needed.



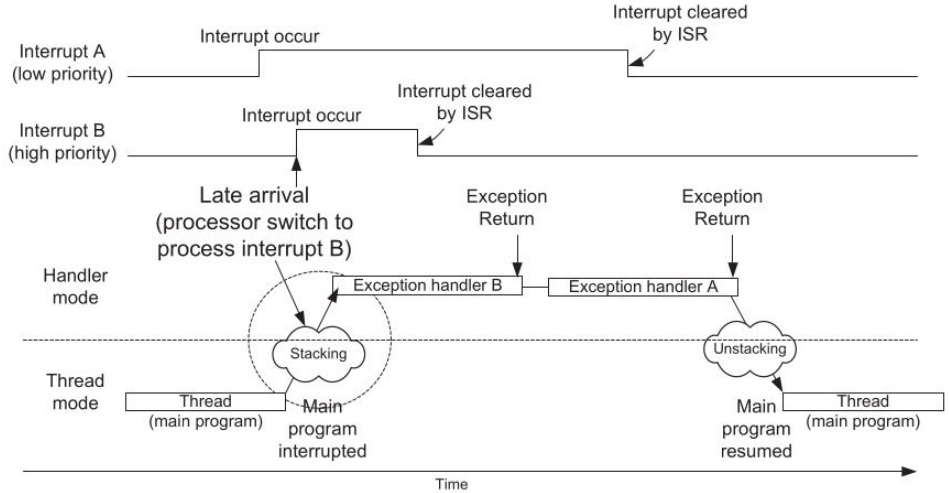
Tail chaining

If an exception is in pending state when another exception handler is completed, instead of returning to the interrupted program, the processor will go to the pending interrupt handler.



Late Arrival

If a higher priority exception occurs during stacking process of a lower-priority exception, the processor switches to handle the higher priority exception first



EXC_RETURN

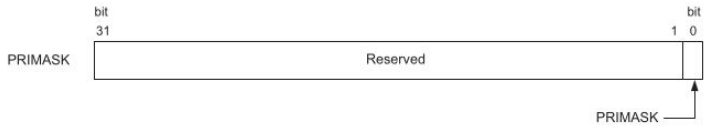
- Special architecturally defined value for triggering and helping exception return mechanism.
- This value is generated automatically when an exception is accepted and is stored into the LR after stacking.

Bits	31:28	27:4	3	2	1	0
Descriptions	EXC_RETURN indicator	Reserved	Return mode	Return stack	Reserved	Processor state
Value	0xF	0xFFFFFF	1 (thread) or 0 (handler)	0 (main stack) or 1 (process stack)	0	1 (reserved)

EXC_RETURN	Condition
0xFFFFFFF1	Return to handler mode (nested exception case)
0xFFFFFFF9	Return to Thread mode and use the main stack for return
0xFFFFFFFD	Return to Thread mode and use the process stack for return

Exception Masking Register (PRIMASK)

- Sometimes it is necessary to disable all interrupts for a short period of time (i.e atomic).
- Instead of disabling interrupts one at a time, PRIMASK register can be used to disable all interrupts, and enable them back.
- Note that HardFault and NMI are exempt from this.



PRIMASK - control

- To set PRIMASK (disable interrupts)

```
MOVS R0, #1
MSR  PRIMASK, R0
```

- Alternatively

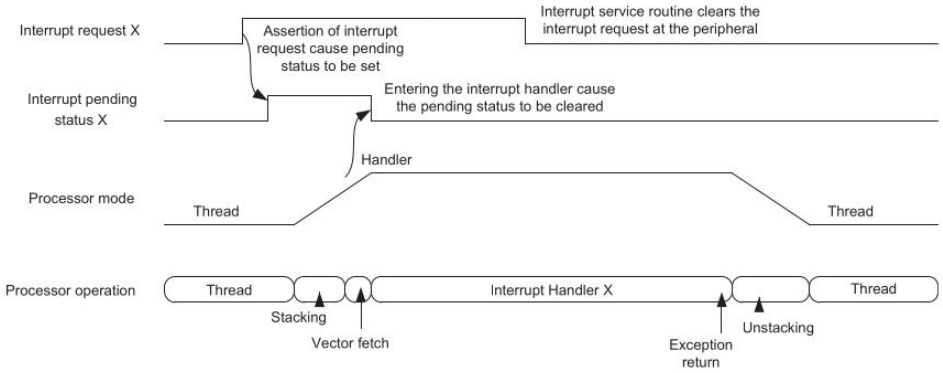
```
CPSIE i
CPSID i
```

- CMSIS versions

```
void __enable_irq(void);
void __disable_irq(void);
```

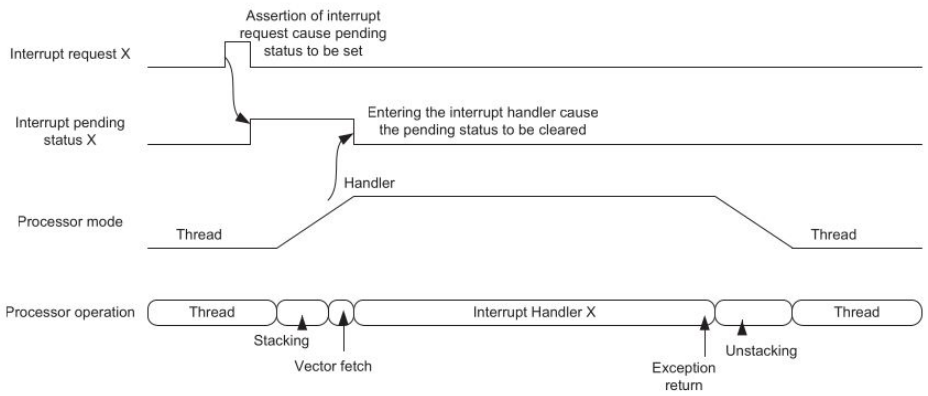

Interrupt Inputs and Pending Behavior

The peripherals will generate an interrupt request, and **if not pulsed**, **will be expecting to be cleared** in the ISR.
The pending status is cleared **automatically**.



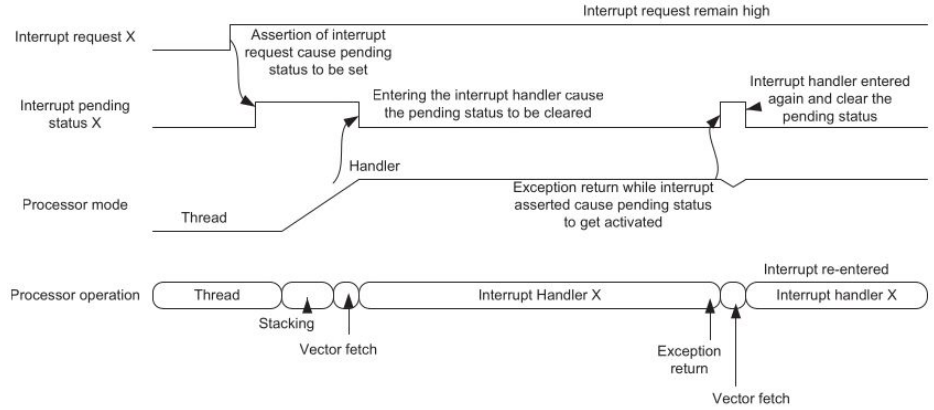
Interrupt Inputs and Pending Behavior

Some interrupt sources might generate IRQs in the form of a **pulse**. **ISR does not need to clear it**. The pending status is cleared **automatically**.



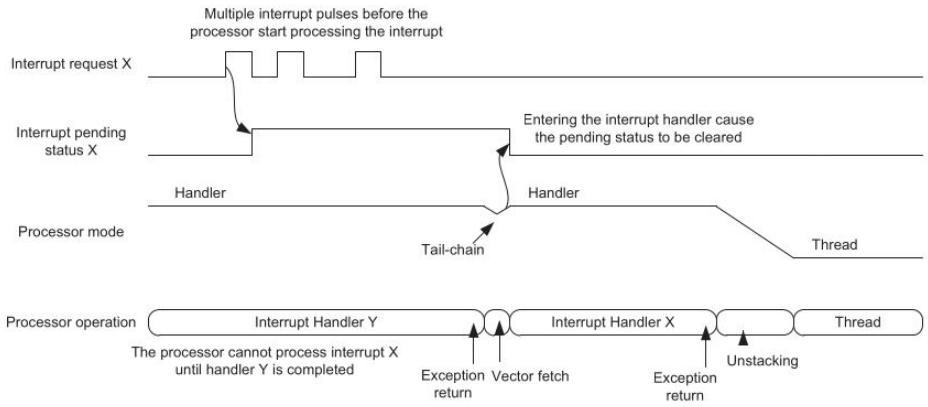
Interrupt Inputs and Pending Behavior

Forgetting to clear the request from peripheral might result in the e**xecution of an infinite loop of the ISR**



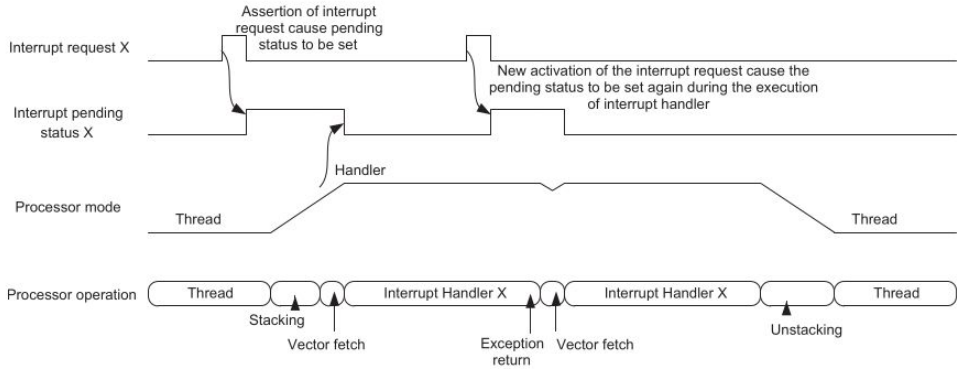
Interrupt Inputs and Pending Behavior

Multiple IRQ pulses are treated **as one interrupt** before servicing.



Interrupt Inputs and Pending Behavior

Another **IRQ from the same source** while handling the interrupt will be a served **as the second one**



Interrupt Latency

- There is a **latency** associated with servicing interrupts
- It can be defined as the **processor clock cycles** between the i**nterrupt is asserted** and the **start of the execution** of the interrupt handler
- For Cortex-M0+ this is *typically* **15 clock cycles**

Program design with interrupts

- How much work to do in ISR?
  - Spending a long time in ISR will delay other tasks (or even other ISRs)
  - In systems with multiple ISRs with short deadlines, **perform critical work in ISR** and **buffer partial results** for later processing
- How to communicate between ISR and other threads?
  - Data buffering
  - Data integrity and race conditions
    - **Volatile** data - can be updated outside of the program's immediate control
    - **Non-atomic shared data** - can be interrupted partway through read or write, therefore it is vulnerable to race conditions.

Volatile data

- Compilers assume that variables in memory don't change spontaneously and optimize based on that belief
  - Don't reload a variable from memory if current function hasn't changed it
  - Read variable from memory into register (faster access)
  - Write back to memory at the end of the procedure or before a procedure call, or when compiler runs out of free registers
- Optimization will fail
  - if variables are used inside different routines or procedures
  - Memory-mapped peripheral registers

Non-atomic shared data

When we want to change or modify a block of variables (or even single variable) there is always a chance that it might get interrupted halfway through it

```
GPIOD->ODR |= 0x01;
```

will translate to

```
LDR R0, =GPIOD->ODR
LDR R1, [R0]
MOVS R2, 0x01
ORRS R1, R1, R2
STR R1, [R0]
```

What will happen if this is interrupted somewhere here, and the ISR changes `GPIOD->ODR` register (or it changes automatically)

Another example - Checking time

```
void getTime(datetime_t* d) {
    d->day = current_time.day;
    d->hour = current_time.hour;
    d->minute = current_time.minute;
    d->second = current_time.second;
}

void timeISR(void) {
current_time.second++;
if (current_time.second > 59) {
    current_time.second = 0;
    current_time.minute++;
    if (current_time.minute > 59) {
        current_time.minute = 0;
        current_time.hour++;
        if (current_time.hour > 23) {
            current_time.day++;
            ...
}}}}
```

What happens when `timeISR()` is serviced in the middle of getTime routine?

`timeISR()` is tied to a timer and gets updated every second

Non-atomic shared data

- Fundamental problem is **race condition**
  Race Condition: Anomalous behavior due to unexpected critical dependence on the relative of timing of events. Result of example code depends on the relative timing of the read and write operations.
- Preemption enables ISR to interrupt other code and possibly **overwrite data**
- Must ensure **atomic (indivisible)** access to the object
- Native atomic object size depends on the **processor's instruction set** and word size
- Another way is to disable / enable interrupts within **critical code sections**
  Critical code section: A section of code which creates a possible race condition. Therefore a synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use

Remarks

- Protect any data object which both
  - **Requires multiple instructions to read or write** (non-atomic access)
  - **Is potentially written by an ISR**
- How many tasks/ISRs can write to the data object?
  - If one, then we have one-way communication
    - must ensure the data isn't overwritten partway through being read - writer and reader don't interrupt each other
  - If more than one, we
    - must ensure the data isn't overwritten partway through being read - writer and reader don't interrupt each other
    - must ensure the data isn't overwritten partway through being written - writers don't interrupt each other

Solution - Briefly disable preemption

```
void getTime(datetime_t* d) {
    __disable_irq();
    d->day = current_time.day;
    d->hour = current_time.hour;
    d->minute = current_time.minute;
    d->second = current_time.second;
    __enable_irq();
}

void timeISR(void) {
current_time.second++;
if (current_time.second > 59) {
    current_time.second = 0;
    current_time.minute++;
    if (current_time.minute > 59) {
        current_time.minute = 0;
        current_time.hour++;
        if (current_time.hour > 23) {
            current_time.day++;
            ..
}}}}
```

disable / enable IRQs will make sure it does not get preempted.

`timeISR()` is tied to a timer and gets updated every second

This week

- Read Chapters 3.5 and 8 from Yiu
- Project 1 due on Monday night
- HW3 is postponed to next week
- Lab3 is incoming on Tuesday

Links

- On volatile keyword https://barrgroup.com/embedded-systems/how-to/c-volatile-keyword
- CMSIS v5 https://github.com/ARM-software/CMSIS_5