Lesson:8 (interrupt)

- ✓ **Lesson 8: Interrupts**
  - ✓ 8.1 Introduction
  - ✓ 8.2 Commands
  - ✓ 8.3 Kconfig
  - ✓ 8.4 Exercise

What is an interrupt?

An interrupt service routine (ISR) is a function that executes asynchronously in response to a hardware or software interrupt. An ISR normally preempts the execution of the current thread, allowing the response to occur with very low overhead. Thread execution resumes only once all ISR work has been completed.

An ISR has the following key properties:

- An interrupt request (IRQ) signal that triggers the ISR
- A priority level associated with the IRQ
- An interrupt handler function that is invoked to handle the interrupt
- An argument value that is passed to that function

An interrupt descriptor table (IDT) or a vector table is used to associate a given interrupt source with a given ISR. Only a single ISR can be associated with a specific IRQ at any given time.

Multiple ISRs can utilize the same function to process interrupts, allowing a single function to service a device that generates multiple types of interrupts or to service multiple devices (usually of the same type). The argument value passed to an ISR's function allows the function to determine which interrupt has been signaled.

The kernel provides a default ISR for all unused IDT entries. This ISR generates a fatal system error if an unexpected interrupt is signaled.

The kernel supports interrupt nesting. This allows an ISR to be preempted in mid-execution if a higher priority interrupt is signaled. The lower priority ISR resumes execution once the higher priority ISR has completed its processing.

An ISR's interrupt handler function executes in the kernel's interrupt context. This context has its own dedicated stack area. The size of the interrupt context stack must be capable of handling the execution of multiple concurrent ISRs if interrupt nesting support is enabled.

Important: Many kernel APIs can be used only by threads, and not by ISRs. In cases where a routine may be invoked by both threads and ISRs the kernel provides the k_is_in_isr() function to allow the routine to alter its behavior depending on whether it is executing as part of a thread or as part of an ISR.

How to define a regular interrupt?

An ISR is defined at run-time by calling IRQ_CONNECT. It must then be enabled by calling irq_enable().

Important: IRQ_CONNECT() is not a C function and does some inline assembly magic behind the scenes. All its arguments must be known at build time. Drivers that have multiple instances may need to define per-instance config functions to configure each instance of the interrupt.

The following code defines and enables an ISR.

```
#define MY_DEV_IRQ  24       /* device uses IRQ 24 */
#define MY_DEV_PRIO  2       /* device uses interrupt priority 2 */
/* argument passed to my_isr(), in this case a pointer to the device */
#define MY_ISR_ARG  DEVICE_GET(my_device)
#define MY_IRQ_FLAGS 0       /* IRQ flags. Unused on non-x86 */

void my_isr(void *arg)
{
   ... /* ISR code */
}

void my_isr_installer(void)
{
   ...
   IRQ_CONNECT(MY_DEV_IRQ, MY_DEV_PRIO, my_isr, MY_ISR_ARG,
MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
   ...
}
```

How to define a 'direct' ISR?

Regular Zephyr interrupts introduce some overhead which may be unacceptable for some low-latency use-cases. Specifically:

- The argument to the ISR is retrieved and passed to the ISR
- If power management is enabled and the system was idle, all the hardware will be resumed from low-power state before the ISR is executed, which can be very time-consuming
- Although some architectures will do this in hardware, other architectures need to switch to the interrupt stack in code
- After the interrupt is serviced, the OS then performs some logic to potentially make a scheduling decision

Zephyr supports so-called 'direct' interrupts, which are installed via IRQ_DIRECT_CONNECT. These direct interrupts have some special implementation requirements and a reduced feature set; see the definition of IRQ_DIRECT_CONNECT for details.

The following code demonstrates a direct ISR:

```
#define MY_DEV_IRQ  24       /* device uses IRQ 24 */
#define MY_DEV_PRIO  2       /* device uses interrupt priority 2 */
/* argument passed to my_isr(), in this case a pointer to the device */
#define MY_IRQ_FLAGS 0       /* IRQ flags. Unused on non-x86 */

ISR_DIRECT_DECLARE(my_isr)
{
   do_stuff();
   ISR_DIRECT_PM(); /* PM done after servicing interrupt for best latency */
   return 1; /* We should check if scheduling decision should be made */
}

void my_isr_installer(void)
{
   ...
   IRQ_DIRECT_CONNECT(MY_DEV_IRQ, MY_DEV_PRIO, my_isr,
MY_IRQ_FLAGS);
   irq_enable(MY_DEV_IRQ);
   ...
}
```

How to disable interrupts?

In certain situations it may be necessary for the current thread to prevent ISRs from executing while it is performing time-sensitive or critical section operations.

A thread may temporarily prevent all IRQ handling in the system using an IRQ lock. This lock can be applied even when it is already in effect, so routines can use it without having to know if it is already in effect. The thread must unlock its IRQ lock the same number of times it was locked before interrupts can be once again processed by the kernel while the thread is running.

Important: The IRQ lock is thread-specific. if thread A locks out interrupts then performs an operation that allows thread B to run (e.g. giving a semaphore or sleeping for N milliseconds), the thread's IRQ lock no longer applies once thread A is swapped out. This means that interrupts can be processed while thread B is running unless thread B has also locked out interrupts using its own IRQ lock.

When thread A eventually becomes the current thread once again, the kernel re-establishes thread A's IRQ lock. This ensures thread A won't be interrupted until it has explicitly unlocked its IRQ lock.

Alternatively, a thread may temporarily disable a specific IRQ. The IRQ must be subsequently enabled to permit the ISR to execute.

Important: Disabling an IRQ prevent all threads in the system from being preempted by the associated ISR, not just the thread that disabled the IRQ.

When to use an interrupt?

Use a regular or direct ISR to perform interrupt processing that requires a very rapid response, and can be done quickly without blocking.

Note: Interrupt processing that is time consuming, or involves blocking, should be handed off to a thread. See Offloading ISR Work for a description of various techniques that can be used in an application.

The following interrupt-related APIs are provided by irq.h:

Command	**Description**
IRQ_CONNECT	Initialize an interrupt handler. This routine initializes an interrupt handler for an IRQ. The IRQ must be subsequently enabled before the interrupt handler begins servicing interrupts.
IRQ_DIRECT_CONNECT	Initialize a 'direct' interrupt handler. This routine initializes an interrupt handler for an IRQ. The IRQ must be subsequently enabled via irq_enable() before the interrupt handler begins servicing interrupts.
ISR_DIRECT_HEADER	Common tasks before executing the body of an ISR. This macro must be at the beginning of all direct interrupts and performs minimal architecture-specific tasks before the ISR itself can run. It takes no arguments and has no return value.
ISR_DIRECT_FOOTER	Common tasks before exiting the body of an ISR. This macro must be at the end of all direct interrupts and performs minimal architecture-specific tasks like EOI. It has no return value.
ISR_DIRECT_PM	Perform power management idle exit logic.
ISR_DIRECT_DECLARE	Helper macro to declare a direct interrupt service routine.
irq_lock()	Lock interrupts. This routine disables all interrupts on the CPU.

Command	Description
irq_unlock()	Unlock interrupts.
irq_enable()	Enable an IRQ.
irq_disable()	Disable an IRQ.
irq_is_enabled()	Get IRQ enable state.

The following interrupt-related APIs are provided by kernel.h:

Command	Description
k_is_in_isr()	Determine if code is running at interrupt level.
k_is_preempt_thread()	Determine if code is running in a preemptible thread.

Button-interrupt

GPIO-input-interrupt

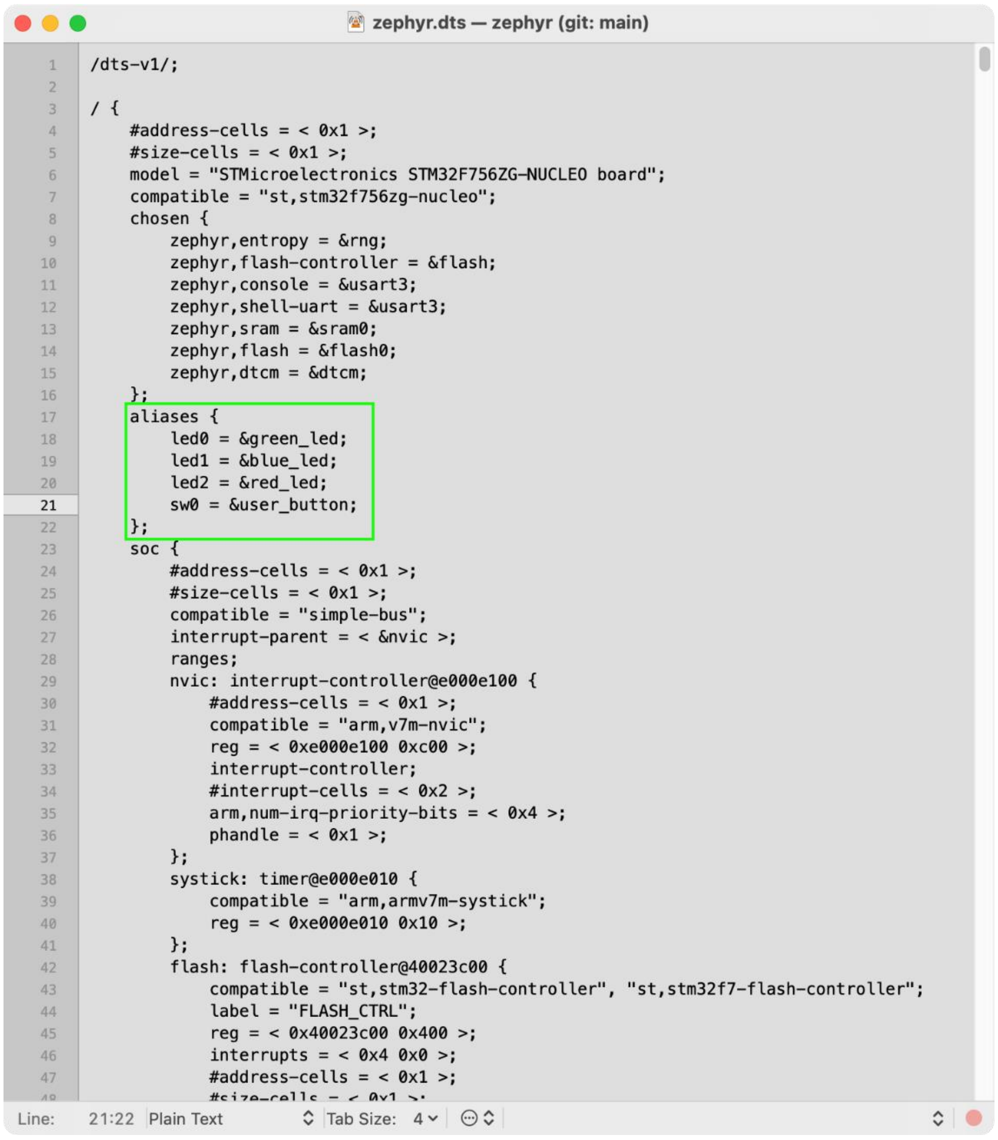
Configure an interrupt on a button press.

1 thread running continously putting out "loop message" every second

on button press: print out "isr message"

Zephyr docs Example 1 Example 2

User button and LED aliases can be found in the devicetree file `build/zephyr/zephyr.dts`.



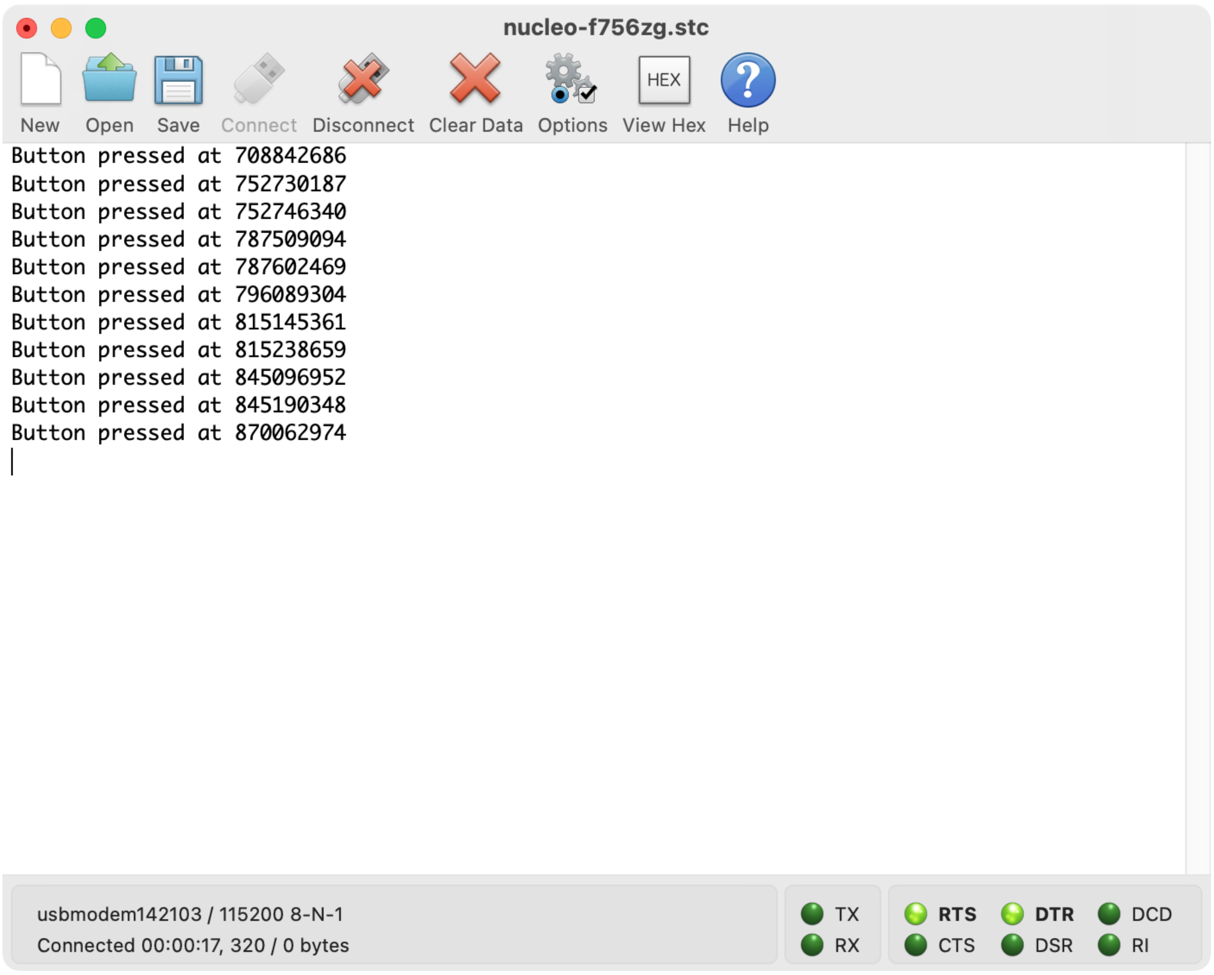
zephyr.dts — zephyr (git: main)

```
/dts-v1/;

/ {
    #address-cells = < 0x1 >;
    #size-cells = < 0x1 >;
    model = "STMicroelectronics STM32F756ZG-NUCLEO board";
    compatible = "st,stm32f756zg-nucleo";
    chosen {
        zephyr,entropy = &rng;
        zephyr,flash-controller = &flash;
        zephyr,console = &usart3;
        zephyr,shell-uart = &usart3;
        zephyr,sram = &sram0;
        zephyr,flash = &flash0;
        zephyr,dtcm = &dtcm;
    };
    aliases {
        led0 = &green_led;
        led1 = &blue_led;
        led2 = &red_led;
        sw0 = &user_button;
    };
    soc {
        #address-cells = < 0x1 >;
        #size-cells = < 0x1 >;
        compatible = "simple-bus";
        interrupt-parent = < &nvic >;
        ranges;
        nvic: interrupt-controller@e000e100 {
            #address-cells = < 0x1 >;
            compatible = "arm,v7m-nvic";
            reg = < 0xe000e100 0xc00 >;
            interrupt-controller;
            #interrupt-cells = < 0x2 >;
            arm,num-irq-priority-bits = < 0x4 >;
            phandle = < 0x1 >;
        };
        systick: timer@e000e010 {
            compatible = "arm,armv7m-systick";
            reg = < 0xe000e010 0x10 >;
        };
        flash: flash-controller@40023c00 {
            compatible = "st,stm32-flash-controller", "st,stm32f7-flash-controller";
            label = "FLASH_CTRL";
            reg = < 0x40023c00 0x400 >;
            interrupts = < 0x4 0x0 >;
            #address-cells = < 0x1 >;
            #size-cells = < 0x1 >;
```

Line: 21:22 Plain Text Tab Size: 4

Expected output:



```
Button pressed at 708842686
Button pressed at 752730187
Button pressed at 752746340
Button pressed at 787509094
Button pressed at 787602469
Button pressed at 796089304
Button pressed at 815145361
Button pressed at 815238659
Button pressed at 845096952
Button pressed at 845190348
Button pressed at 870062974
```

usbmodem142103 / 115200 8-N-1
Connected 00:00:17, 320 / 0 bytes