A white background with black dots

Description automatically generated

**LABORATORY MANUAL**

**Interrupts**

**ay23/24**

**OBJECTIVES**

* 1. Learn the difference between polling and interrupts
  2. Configure and implement hardware interrupts
  3. Setup and configure the IR-based sensor as a wheel encoder

**EQUIPMENT**

1. Computer or laptop that has the Pico Build tools installed
2. Raspberry Pico W
3. HC-020K Photoelectric encoders
4. Breadboard
5. LED with resistor
6. Micro-USB Cable

**NOTE:**

1. Only students wearing fully covered shoes are allowed in the lab due to safety.
2. Bring your laptop with Code Composer Studio installed.
3. For your understanding and a better quiz preparation, take notes about the given tasks, especially questions or unexpected code behaviour.

**Introduction**

Embedded systems can interact with their surrounding environment in many ways. Whether sensing or actuating, there are two primary mechanisms initiating actions: **event-triggered** and **time-triggered**. In this lab, we learn how to use interrupts which are the building blocks for **event-triggered** functionality. Lastly, we will develop code that incorporates interrupts to use the IR sensor as a wheel encoder for measuring the wheel rotation speed.

***Task 1: Introduction to Interrupts***

A hardware interrupt is an electronic signal that alerts the microprocessor of an event. An interrupt can be triggered by either an internal peripheral (e.g. timer) or an external device (e.g. button).

In the previous lab session, polling was used to detect when a high signal is received. When polling is used, the microprocessor repeatedly checks whether the event has occurred. In the case of the previous lab, the value of the GPIO pin is read to determine if it is high or low. The microprocessor detects it quickly since it is always active and does nothing but check this single condition, as shown in Figure 1 (bottom).

While polling is a simple way to check for state changes, there's a cost. If the checking interval is too long, there can be a long lag between occurrence and Detection, and you may fail to see the change entirely if the state changes back before you check. A shorter interval will get faster and more reliable Detection and consume much more processing time and power since many more checks will come back negative.

An alternative is to configure an interrupt on the button's GPIO pin such that when a pre-configured trigger condition is met, an interrupt is generated. With this approach, the microprocessor can enter a low-power sleep state and be woken up with the interrupt. The Pico can detect signal changes (e.g. rising or falling edges) to generate an interrupt, as shown in Figure 1 (top). If a GPIO pin is configured to be pulled up, a falling edge will occur when the button is pressed, pulling the pin to the ground. Interrupts are thus better suited to handle asynchronous events.



***Figure 1****: Detecting an event with polling and with hardware interrupts [1].*

Typical microprocessors have multiple states, such as active and sleep, that are important for power management. Depending on the application, the transition between these states can be time-triggered or event-triggered. In both cases, an interrupt initiates the change from sleep to an active state. An event-triggered can come from a GPIO pin, while a time-triggered can come from a timer or a real-time clock (RTC). Figure 2 shows a typical state diagram for these two systems.

You can enter the WFE (Wait for Event) power mode in the Pico by calling the sleep\_ms() or sleep\_us() provided by the “pico\_time” library. Depending on the lower power mode level, the microprocessor will turn off a set of peripherals and internal functions to save energy. Then it will halt (sleep) until an interrupt happens. When the interrupt is received, it will switch to active mode and execute the corresponding interrupt service routine (ISR). Afterwards, it will continue with the program in the main() function.

An event such as a button press will lead to an interrupt. The interrupt flag (IFG) register has a bit assigned to each configured interrupt peripheral. On every occasion the interrupt happens, the corresponding IFG bit is set. Three conditions must be met before the microprocessor is alerted about the event:

1. The pin must be configured to trigger an interrupt.

2. The corresponding interrupt must be enabled.

3. If these two conditions hold, the global interrupt enable (GIE) register is tested. If global interrupts are enabled, the microprocessor then is preempted, and the according to interrupt service routine (ISR) be executed.

This global switch is useful to temporarily turn off all three interrupts to prevent modifying a global variable or avoid preemption between instructions that logically belong together. The interrupt vector table maps the interrupts to the correct ISR.



***Figure 2****: Polling: The system typically remains in active mode to detect an event.* ***(left)***

*Interrupt: The system can be in sleep mode until an event occurs.* ***(right)***

A dedicated or grouped interrupt is triggered, depending on the source of the interrupt. For peripherals like GPIO ports, multiple pins could produce the same interrupt. In these cases, it is necessary to query the pin's interrupt vector register to identify the interrupt's exact source. Typically, this is done inside the ISR. Once an ISR identifies the source of the interrupt, it can react in accordance. Typically, ISRs execute in a privileged mode that can mask other interrupts. Hence, ISR should be **as short as possible** and only set application-specific flags to indicate to the microprocessor's main thread to execute the corresponding task in response to an interrupt.

Each interrupt type has a priority such that it is always clear which interrupt is handled first in case multiple interrupts occur at the same time. Furthermore, it is possible to allow nested interrupts, i.e., one interrupt interrupts another interrupt's handling.

**Configuring an event-triggered interrupt**

On reset, all interrupts are disabled. The following steps are necessary to configure a GPIO event-triggered interrupt on the MSP432.

1. Configure the GPIO pin as an input. gpio\_set\_input\_enabled()
2. Clear the pin's interrupt flag. This makes sure that no previous interrupts are handled. (This step is not mandatory, but it is good practice to do so.) irq\_clear()
3. Set the interrupt enable (IE) bit of the specific GPIO pin and select the edge (clock signal) that triggers the interrupt (enabling the interrupt in the peripheral). gpio\_set\_irq\_enabled()
4. Enable interrupts globally (set global interrupt enable (GIE) bit). irq\_set\_enabled()
5. Details of the corresponding API call can be found in SDK manual. After these steps, any new event (positive-edge or negative-edge) on the pin will generate an interrupt, preempt the microprocessor, and start executing the corresponding ISR.

The SDK also provides a convenience function that performs the above functions in a single function call. gpio\_set\_irq\_enabled\_with\_callback(). For example usage, please refer to [here](https://github.com/raspberrypi/pico-examples/blob/master/gpio/hello_gpio_irq/hello_gpio_irq.c).

**Defining an Interrupt Service Routine (ISR)**

The Interrupt Service Routine (ISR) is executed if the processor detects the corresponding interrupt. ISR has few restrictions; since it cannot receive arguments or return values, it can only modify global variables. In general, a programmer has the flexibility to choose any arbitrary function to be an ISR.

The Pico uses a user defined callback function as an ISR. The function is called when ever an interrupt occurs.

**Task 1.1: Event driven interrupt**

Configure Lab 2 solution to use an event-driven interrupt. The LED should toggle on each rising edge.

**Task 2: IR-based Wheel Encoder**



***Figure 4*** *HC-020K Photoelectric encoders*

The working principle of the encoder shown in Figure 4 is presented in Figure 5 (a). It uses a slotted wheel with a single LED and photodetector pair that generate pulses as the wheel turns, and the speed of an object can be calculated by measuring the pulse duration Δti (i.e. elapsed time or period of a pulse) between successive pulses [2]. It comprises three connections, GND, VCC and OUT. GND and VCC are meant to supply power to the module (in our case via the Pico GND and 3.3V pins), while OUT generates the square-pulse signal.



***Figure 5 (a)*** *Encoder principle and* ***(b)*** *operations of measuring elapsed time****.*** *[3]*

**Task 2.1: Testing the Wheel Encoder**

This section illustrates the working of a wheel encoder without a single line of source code. This is achieved by using a separate LED on a breadboard. The VCC pin will be connected to a 3.3V pin and GND connected to GND, as shown in Figure 6. Connect the Pico to the micro-USB to boot it up.

***Figure 6*** *Hard-wiring the encoder to Pico*

The encoder's LED will be turned on perpetually, while the photosensor will detect the light from the LED if unblocked and vice-versa. The specifications for the IR sensor is as follows:

* • Chip: 74HC14D
* • VCC: +3.3V ~ 5 V
* • GND: Ground or negative
* • OUT: The signal output, link MCU I/O port
* • HIGH = Detect obstacles
* • LOW = No obstacle

Follow the video guide for "Wheel" followed by the "Wheel Encoder". We are not going to use the motors in this lab session. However, the wheels help illustrate the use of the encoder. Turning the wheel "attached" to the wheel encoder will cause the LED to toggle each time it transitions from blocked to unblocked and vice-versa.

**Task 2.2: Wheel Encoder using Interrupts**

This sub-task seeks to set up an interrupt (in this case, an ISR) such that each time the photosensor in the encoder detects a transition from block to unblock (or vice-versa), it will increment the counter. A full rotation comprising 20 transitions (corresponding to the 20 notches on the encoder) will toggle LED via the GP2. The following is the program flow for both the **main function** and **ISR**.

**Main Function**

1. Declare a **global** counter variable
2. Initialise the **global** variable
3. Configure pin GP2 as **input** (with pull-up resistor)
4. Configure pin GP25 as **output**
5. **Clear** the interrupt flag for pin GP2
6. **Enable** interrupt for pin GP2
7. **Assign** callback function to interrupt.
8. **Enable** interrupt for IRQ\_BANK0
9. **Enable** master interrupt
10. Forever loop

**Interrupt Service Routine (ISR)**

1. Declare an callback function to increment global variable
2. In the same function, check if a full rotation is achieved
   1. If yes, toggle LED and reset global variable
   2. Else, do nothing

***Submission***

Submit the completed code.

***References***

[1] Renesas. "Essentials of Microcontroller Use Learning about Peripherals: Interrupts." Accessed on 19 Sept 2021. (https://www.renesas.com/us/en/support/engineer-school/mcu-programming-peripherals-04-interrupts)

[2] Nurmi, Jarmo, et al. "Micro-electromechanical system sensors in unscented Kalman filter-based condition monitoring of hydraulic systems." 2013 IEEE/ASME International Conference on Advanced Intelligent Mechatronics. IEEE, 2013.

[3] Mones, Zainab & Feng, Guojin & Muo, Ugo & Wang, Tie & Gu, Fengshou & Ball, Andrew. (2016). Performance evaluation of wireless MEMS accelerometer for reciprocating compressor condition monitoring: Proceedings of the International Conference on Power Transmissions 2016 (ICPT 2016), Chongqing, P.R. China, 27–30 October 2016.