Hands on session 07: Interrupts

Session Introduction: This session will introduce the concept of interrupts.

Session Objectives:

- **To understand GPIO interrupts**
- **To understand external interrupts**

The ESP32 offers up to 32 interrupt slots for each core. Each interrupt has a certain priority level and can be categorized into two types.

Hardware Interrupts – These occur in response to an external event. For example, GPIO Interrupt(when a key is pressed down) or a Touch Interrupt(when touch is detected)

Software Interrupts – These occur in response to a software instruction. For example, a Simple Timer Interrupt or Watchdog Timer Interrupt(when timer times out)

ESP32 GPIO Interrupt

In ESP32, we can define an interrupt service routine function that will be called when a GPIO pin changes its signal value. With an ESP32 board, all the GPIO pins can be configured to function as interrupt request inputs.

Attaching Interrupt to a GPIO Pin

In Arduino IDE, we use a function called **attachInterrupt()** to set an interrupt on a pin by pin basis. The recommended syntax looks like below.

```
attachInterrupt(GPIO Pin, ISR, Mode);
```

This function takes three parameters:

GPIO Pin – Sets the GPIO pin as an interrupt pin, which tells the ESP32 which pin to monitor.
ISR – Is the name of the function that will be called every time the interrupt is triggered.
Mode – Defines when the interrupt should be triggered. Five constants are predefined as valid values:

- **LOW:** Triggers interrupt whenever the pin is low
- **HIGH:** Triggers interrupt whenever the pin is high
- **RISING:** Triggers interrupt whenever the pin is changing from low to high
- **FALLING:** Triggers interrupt whenever the pin is changing from high to low
- **CHANGE:** Triggers interrupt whenever the pin is changing state

Interrupt Service Routine

Interrupt Service Routine is invoked when an interrupt occurs on any GPIO pin. Its syntax looks like below.

```
void IRAM_ATTR ISR() {
   Statements;
}
```

Why IRAM_ATTR?

- By flagging a piece of code with the IRAM_ATTR attribute we are declaring that the compiled code will be placed in the Internal RAM (IRAM) of the ESP32.

- Otherwise, the code is placed in the Flash. And flash on the ESP32 is much slower than internal RAM.
- Some timing critical code may be placed into IRAM to reduce the penalty associated with loading the code from flash. ESP32 reads code and data from flash via a 32 kB cache. In some cases, placing a function into IRAM may reduce delays caused by a cache miss.

For reliable interrupt operation:

Variables should be declared **volatile** if their values can be changed during an interrupt service routine.

```
volatile int state = LOW;
```

places variables in RAM so that a processor storage register is not used and could get corrupted during an interrupt.

When writing your interrupt service routine, you should write it in a following way:

```
void IRAM_ATTR ISR()
{
  portENTER_CRITICAL(&synch);

statements…

  portEXIT_CRITICAL(&synch);
}
```

IRAM_ATTR

All interrupts should be placed in instruction RAM(IRAM) using IRAM_ATTR directive

Why use IRAM_ATTR?

the code you want to run is NOT in RAM, then where else could it be? The answer is "flash" ... if it is in flash, then when a request to execute that code is received, the code has to be executed from there. Flash on the ESP32 is much slower than RAM access ... so there is a memory cache which can be used to resolve some of that ... however we can't be assured that when we branch to a piece of code that it will be present in cache and hence may need a slow load from flash.

If the code we want to run is an interrupt service routine (ISR), we invariably want to get in and out of it as quickly as possible. If we had to "wait" within an ISR for a load from flash, things would go horribly wrong. By flagging a function as existing in RAM we are effectively sacrificing valuable RAM for the knowledge that its access will be optimal and of constant time.

ESP32 needs a special handler to synchronise variable modification between ISR and main routines.

```
portMUX_TYPE synch = portMUX_INITIALIZER_UNLOCKED;
void IRAM_ATTR ISR()
{
  portENTER_CRITICAL(&synch);

statements…

  portEXIT_CRITICAL(&synch);
}
```

We should declare a variable of type portMUX_TYPE, which we will need to take care of the synchronization between the main code and the interrupt.

Enter ISR with: portENTER_CRITICAL(&synch);
Exit ISR with: portENTER_CRITICAL(&synch);