# Lyra API Documentation

Most procedures return either of the following enumerations :

- `OK` - enumerated as 0x10, returned when operation is successful.
- `FAIL` - enumerated as 0x11, returned when operation fails.

When `FAIL` is returned, you can debug by printing `errno` which is set one of the following :

- `EINVAL` - Invalid argument
- `EFAULT` - Bad address, typically encountered when handle or handle instance points to `NULL`.
- `ENXIO` - No such device or address, typically encountered when controller handle points to invalid controller.

---

## GPIO

GPIO is controlled with the following API calls:

### `GPIO_Init`

```
int GPIO_Init( uint16_t GPIOx,
               uint16_t pin,
               uint16_t dir)
```

- **Description** : Initializes a GPIO port pin to general purpose Output or Input.
- **Parameters** :
  - `GPIOx` - GPIO instance to be initialized. Acceptable values - `GPIOA` (0) and `GPIOB` (1)
  - `pin` - Pin to be configured
  - `dir` - `GPIO_OUT` (1) or `GPIO_IN` (0)
- **Return** :
  - `OK` on successful configuration.
  - `FAIL` on invalid argument input .

---

### `GPIO_SetPin`

```
int GPIO_SetPin( uint16_t GPIOx,
                 uint16_t pin)
```

- **Description** : Writes a digital HIGH to GPIO Pin configured as Output.
- **Parameters** :
  - `GPIOx` - GPIO instance. Acceptable values - `GPIOA` (0) and `GPIOB` (1).
  - `pin` - Pin to be written to.
- **Return** :
  - `OK` on successful configuration.
  - `FAIL` on invalid argument input

---

### `GPIO_ResetPin`

```
int GPIO_ResetPin( uint16_t GPIOx,
                   uint16_t pin)
```

- **Description** : Writes a digital LOW to GPIO Pin configured as Output.
- **Parameters** :
  - `GPIOx` - GPIO instance. Acceptable values - `GPIOA` (0) and `GPIOB` (1).
  - `pin` - Pin to be written to.
- **Return** :
  - `OK` on successful configuration.
  - `FAIL` on invalid argument input

---

### `GPIO_TogglePin`

```
int GPIO_TogglePin( uint16_t GPIOx,
                    uint16_t pin)
```

- **Description** : Toggles the specified pin (configured as Output) from last configured output state.
- **Parameters** :
  - `GPIOx` - GPIO instance. Acceptable values - `GPIOA` (0) and `GPIOB` (1)
  - `pin` - Pin to be toggled.
- **Return** :
  - Returns `OK` on successful configuration.
  - Returns `FAIL` on invalid argument input

---

### `GPIO_ReadPin`

```
int GPIO_ReadPin( uint16_t GPIOx,
                  uint16_t pin)
```

- **Description** : Reads the data for pin configured as digital Input.
- **Parameters** :
  - `GPIOx` - GPIO instance. Acceptable values - `GPIOA` (0) and `GPIOB` (1).
  - `pin` - pin to be toggled
- **Return** :
  - **1** for a a logical HIGH.
  - **0** for a logical LOW.
  - `FAIL` on invalid argument input.

---

### `GPIO_IT_Enable`

```
int GPIO_IT_Enable( uint16_t pin )
```

- **Description** : Enables interrupt on PLIC for specified pin on GPIOA. The ISR is handled using `GPIO_EXTICallback` function.
  Remark - Only GPIOA Pin 0 - 11 support external interrupts.
- **Parameters** :
  - `pin` - GPIOA pin

- **Return** :
  - `OK` on successful configuration.
  - `FAIL` on invalid argument input.

---

### `GPIO_IT_Disable`

```
int GPIO_IT_Disable( uint16_t pin )
```

- **Description** : Disable interrupt on PLIC for specified pin on GPIOA.
  Remark - Only GPIOA Pin 0 - 11 support external interrupts.
- **Parameters** :
  - `pin` - GPIOA pin
- **Return** :
  - `OK` on successful configuration.
  - `FAIL` on invalid argument input.

---

### `GPIO_EXTICallback`

```
void GPIO_EXTICallback( uint16_t pin )
```

- **Description** : GPIO External Interrupt Callback function. Implement this routine to handle interrupt service routines for GPIO(A) interrupts.
- **Parameters** :
  - `pin` - GPIOA pin
- **Return** : `void`

---

## Timer

Timer APIs expect the Timer Handle as input. There are pre-defined handles in HAL library, namely `htimer0`, `htimer1` and `htimer2` which are associated with TIMER0, TIMER1 and TIMER2 respectively.

All timer handles are initialized with :

- `LoadCount` = 50000000 for 500 milli-sec delays.
- `Mode` = `TIMER_MODE_PERIODIC` (1) for periodic mode.

---

### `TIMER_Init`

```
int TIMER_Init( TIMER_Handle_t *htimer )
```

- **Description** : Initialises Timer module according to attributes specified by `htimer`.
- **Parameters** :
  - `htimer` - Timer Handle.
- **Return** :
  - `OK` for successful configuration.
  - `FAIL` for failure.

Failure could be encountered by either timer already in running state or bad address being passed as argument. Check `errno` for debugging.

---

### `TIMER_Start`

```
void TIMER_Start( TIMER_Handle_t *htimer)
```

- **Description** : Starts the timer counter for specified Timer.
- **Parameters** :
  - `htimer` - Timer Handle.
- **Return** :
  - `OK` for successful configuration.
  - `FAIL` for failure.

  Failure could be encountered by either timer already in running state or bad address being passed as argument. Check `errno` for debugging.

---

### `TIMER_Start_IT`

```
void TIMER_Start_IT( TIMER_Handle_t *htimer)
```

- **Description** : Starts the timer counter for specified Timer with interrupt enabled in PLIC. The ISR is handled using `TIMER_ElapsedCallback` function.
- **Parameters** :
  - `htimer` - Timer Handle.
- **Return** :
  - `OK` for successful configuration.
  - `FAIL` for failure.

  Failure could be encountered by either timer already in running state or bad address being passed as argument. Check `errno` for debugging.

---

### `TIMER_Stop`

```
void TIMER_Stop(TIMER_Handle_t *htimer)
```

- **Description** : Stops the timer counter and disables interrupt associated with timer.
- **Parameters** :
  - `htimer` - Timer Handle.
- **Return** :
  - `OK` for successful configuration.
  - `FAIL` for failure.

---

### `TIMER_GetCount`

```
int TIMER_GetCount(TIMER_Handle_t *htimer)
```

- **Description** : Reads the timer counter value.
- **Parameters** :
  - `htimer` - Timer Handle
- **Return** :
  - Returns counter value
  - Returns `FAIL` for invalid argument or bad address.

---

### `TIMER_ElapsedCallback`

```
void TIMER_ElapsedCallback(TIMER_Handle_t *htimer)
```

- **Description** : Timer period elapsed callback function. Implement this to handle timer interrupts.
- **Parameters** :
  - `htimer` - Timer Handle
- **Return** : void

---

## Auxiliary functions

### `delayms`

```
void delayms(uint32_t time)
```

- **Description** : Produces a blocking delay in milli-seconds.

WARNING : The default implementation uses TIMER0. When invoked, this function will disable timer0 interrupt, stop the timer and load a different value in Load Count register.

The implementations are declared with weak attribute, so you may choose to override this behavior by re-defining the function.

- **Parameters** :
  - `time` - time in milliseconds
- **Return** : void

---

### `delayus`

```
void delayus(uint32_t time)
```

- **Description** : Produces a blocking delay in micro-seconds.

WARNING : The default implementation uses TIMER0. When invoked, this function will disable timer0 interrupt, stop the timer and load a different value in Load Count register.

The implementations are declared with weak attribute, so you may choose to override this behavior by re-defining the function.

- **Parameters** :
  - `time` - time in micro-seconds
- **Return** : void

---

# Interrupts

## `__enable_irq`

```
void __enable_irq(void)
```

- **Description** : Enables interrupts globally.
- **Parameters** : None
- **Return** : void

---

## `__disable_irq`

```
void __disable_irq(void)
```

- **Description** : Disables interrupts globally.
- **Parameters** : None
- **Return** : void

---