Design with Microprocessors

Lecture 11

The ESP32 Microcontroller - part 2

Year 3 CS
Academic year 2023/2024
1st Semester

Lecturer: Radu Dănescu

The GPIO Matrix and Pin Mux

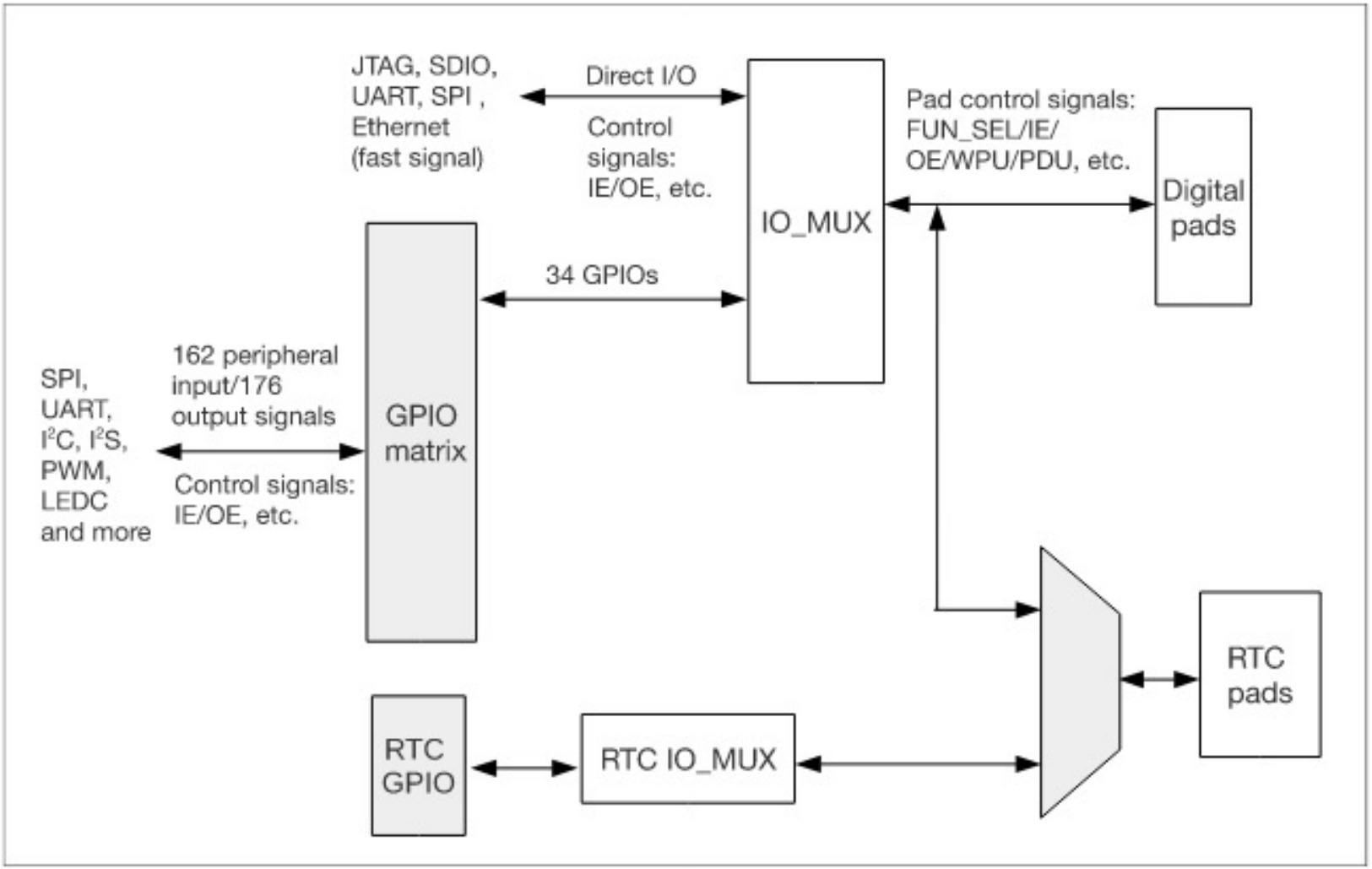
In AVR microcontrollers, integrated peripherals are connected to fixed pins:

- **ATMega2560 (Arduino Mega)**
  - **UART 0 - pins 0 and 1**
  - **I2C - pins 20 and 21**
  - **PWM - pins 2 … 13**

We are forced to use the pins based on the peripheral/interface that is connected to them. Sometimes conflicts may arise.

ESP32 provides a workaround for this situation.

The GPIO Matrix and Pin Mux



https://espressif-docs.readthedocs-hosted.com/projects/arduino-esp32/en/latest/tutorials/io_mux.html

The GPIO Matrix and Pin Mux

Type	Function
ADC	Dedicated GPIOs
DAC	Dedicated GPIOs
Touch Sensor	Dedicated GPIOs
JTAG	Dedicated GPIOs
SD/SDIO/MMC HostController	Dedicated GPIOs
Motor PWM	Any GPIO
SDIO/SPI SlaveController	Dedicated GPIOs
UART	Any GPIO[1]
I2C	Any GPIO
I2S	Any GPIO
LED PWM	Any GPIO
RMT	Any GPIO
GPIO	Any GPIO
Parallel QSPI	Dedicated GPIOs
EMAC	Dedicated GPIOs
Pulse Counter	Any GPIO
TWAI	Any GPIO
USB	Dedicated GPIOs

[1] Except when loading the program, when it has to be connected to the pins connected to the USB controller.

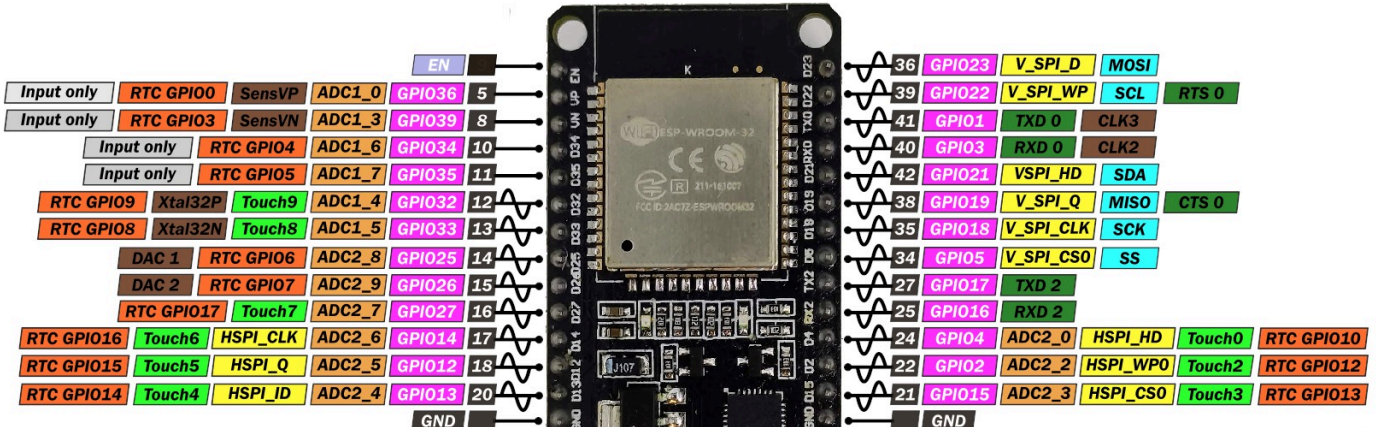
https://espressif-docs.readthedocs-hosted.com/projects/arduino-esp32/en/latest/tutorials/io_mux.html

The GPIO Matrix and Pin Mux

```
// I2C master, default pins
void setup() {
Wire.begin();
}
```

```
// I2C master, custom pins
int sda_pin = 16;
int scl_pin = 17;

void setup() {
Wire.setPins(sda_pin, scl_pin);
Wire.begin();
}
```

https://espressif-docs.readthedocs-hosted.com/projects/arduino-esp32/en/latest/tutorials/io_mux.html



The GPIO Matrix and Pin Mux

Two I2C interfaces used at the same time

```
#include <Wire.h>

TwoWire Wire1 = TwoWire(0);
TwoWire Wire2 = TwoWire(1);

void setup() {
  Wire1.begin(21,22,100000); // SDA pin 21, SCL pin 22, 100kHz frequency
  Wire2.begin(16,17,400000); // SDA pin 16, SCL pin 17, 400kHz frequency
}

void loop() {
  Wire1.beginTransmission(0x38);  // Send data to slave at address 0x38
  Wire1.write(0x80);
  Wire1.write(0x03);
  Wire1.endTransmission();

  Wire2.beginTransmission(0x42);  // Send data to slave at address 0x42
  Wire2.write(140);
  Wire2.endTransmission();
  Wire2.requestFrom(0x42,1);      // Request from slave 1 byte
  if (Wire2.available() == 1) {
    byte value = Wire2.read();
  }
}
```

External Interrupts

- Any GPIO of ESP32 can accept external interrupts
    - External interrupts on AVR Arduino were linked to specific pins

- ESP32 is a dual-core microcontroller
- Each core has 32 interrupts which can be allocated (some docs say only 26 are available)
- Interrupt sources: external pins, timers, interfaces, WiFi, etc
- More sources than available interrupts
- "Interrupt matrix" - mapping between interrupt sources and interrupts of the cores

```
attachInterrupt(digitalPinToInterrupt(PIN), isrName, mode);
```

mode:

- LOW
- HIGH
- RISING
- FALLING
- CHANGE

External Interrupts

```
const int interruptPin = 5;
volatile int numberOfPresses = 0;
volatile int isPressed = 0;

void IRAM_ATTR handleButtonInterrupt() {
  if (isPressed == 0)
      numberOfPresses ++;
  isPressed = 1;
}

void setup() {
  Serial.begin(115200);
  Serial.println("Button interrupts sketch begins");
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin),
handleButtonInterrupt, FALLING);
}

void loop() {
  if (isPressed)
  {
    delay (100);
    Serial.print("Button is pressed. Total count: ");
    Serial.println(numberOfPresses);
    isPressed = 0;
  }
}
```

Tells the compiler to place this code in the RAM instead of FLASH, for faster access

External Interrupts

What happens when we stay too long inside an ISR?
Serial.print takes a significant amount of time!

```
const int interruptPin = 5;
volatile int numberOfPresses = 0;

void IRAM_ATTR handleButtonInterrupt() {
    numberOfPresses++;
    Serial.print("Button is pressed. Total count: ");
    Serial.println(numberOfPresses);
 }


void setup() {

  Serial.begin(115200);
  Serial.println("Button interrupts sketch begins");
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), handleButtonInterrupt,
FALLING);

}

void loop() {

}
```

External Interrupts

What happens when we stay too long inside an ISR?
Serial.print takes a significant amount of time!

```
Button interrupts sketch begins
Button is pressed. Total count: 1
Button is pressed. Total count: 2
Button is pressed. Total count: 3
Button is pressed. Total count: Guru Meditation Error: Core  1 panic'ed (Interrupt wdt
timeout on CPU1).

Core  1 register dump:

PC      : 0x4008ab92  PS      : 0x00060335  A0      : 0x80089b0a  A1      : 0x3ffbf05c
A2      : 0x3ffc56f0  A3      : 0x3ffbd5c8  A4      : 0x00000004  A5      : 0x00060323
A6      : 0x00060323  A7      : 0x00000001  A8      : 0x3ffbd5c8  A9      : 0x00000018
A10     : 0x3ffbd5c8  A11     : 0x00000018  A12     : 0x3ffc22ec  A13     : 0x00060323
A14     : 0x007bf2f8  A15     : 0x003fffff  SAR     : 0x0000001b  EXCCAUSE: 0x00000006
EXCVADDR: 0x00000000  LBEG    : 0x40086405  LEND    : 0x40086415  LCOUNT  : 0xfffffff7

Core  1 was running in ISR context:
EPC1    : 0x400da597  EPC2    : 0x00000000  EPC3    : 0x00000000  EPC4    : 0x00000000

...
```

Timers

- 4 hardware timers
- Each timer is a 64-bit up/down counter
- Each timer has a 16-bit prescaler
- Clock source: the APB bus timer (80 MHz)
- The 16-Bit prescaler can divide the APB_CLK by a factor from 2 to 65536.

Period of timer-generated events (”Alarms”):

$$T_{OUT} = TimerTicks \times \frac{Prescaler}{APB\_CLK}$$

Assuming that we want to generate an event every 1 ms, we can use a prescaler of 80, and get:

$$1ms = TimerTicks \times \frac{80}{80{,}000{,}000}$$

Because 1 ms = 0.001 s, TimerTicks will be **1000**.

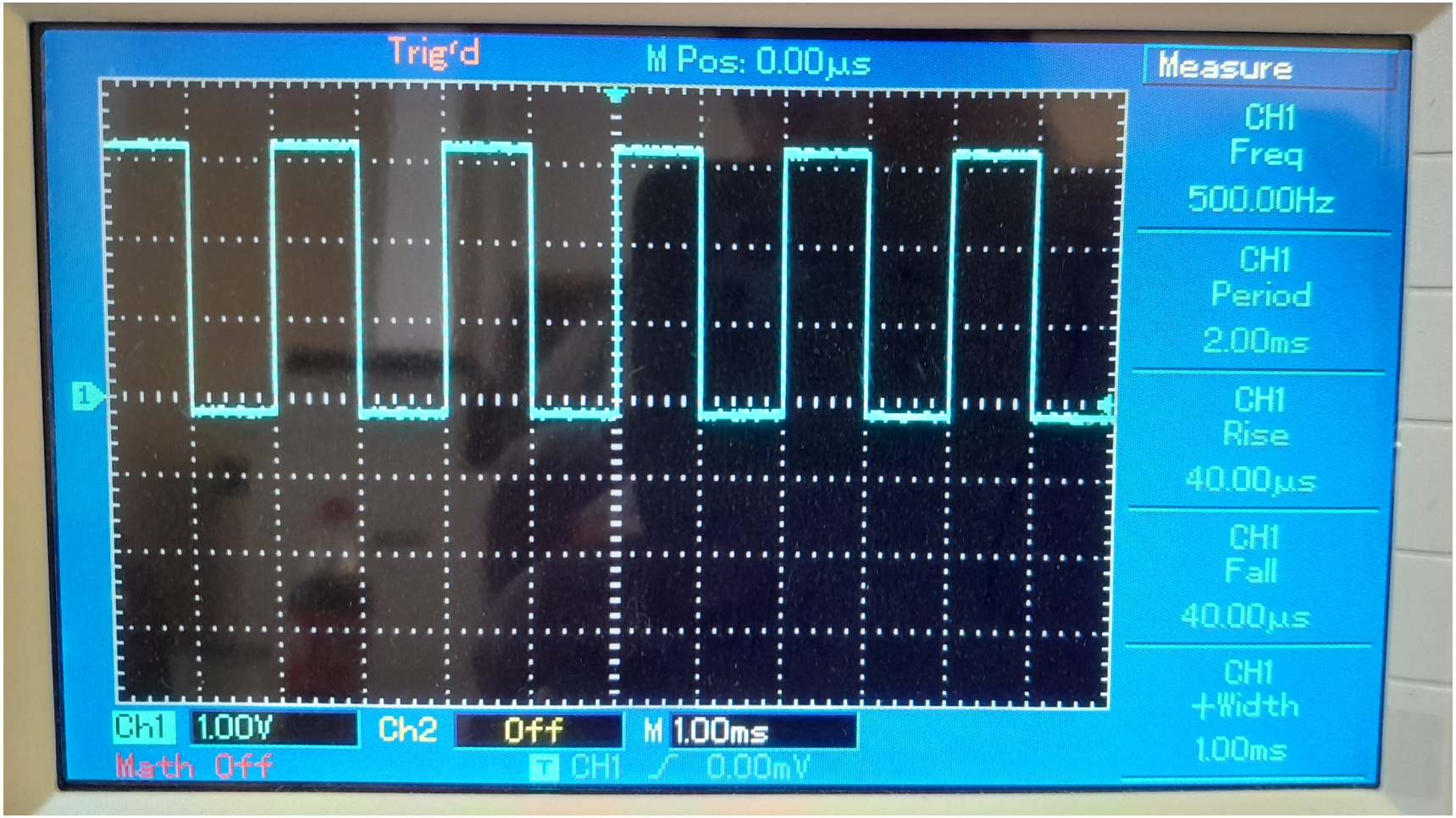
https://deepbluembedded.com/esp32-timers-timer-interrupt-tutorial-arduino-ide/

Timers

```
#define LED 5

hw_timer_t *Timer0_Cfg = NULL;

void IRAM_ATTR Timer0_ISR()
{
    digitalWrite(LED, !digitalRead(LED));
}
void setup()
{
    pinMode(LED, OUTPUT);
    Timer0_Cfg = timerBegin(0, 80, true);
    timerAttachInterrupt(Timer0_Cfg, &Timer0_ISR, true);
    timerAlarmWrite(Timer0_Cfg, 1000, true);
    timerAlarmEnable(Timer0_Cfg);
}
void loop()
{
    // Do Nothing!
}
```

https://deepbluembedded.com/esp32-timers-timer-interrupt-tutorial-arduino-ide/

Timers


Trig'd
M Pos: 0.00µs
Measure
CH1
Freq
500.00Hz
CH1
Period
2.00ms
CH1
Rise
40.00µs
CH1
Fall
40.00µs
CH1
+Width
1.00ms
Ch1 1.00V
Ch2 Off
M 1.00ms
Math Off
CH1 0.00mV

Timers

```
Timer0_Cfg = timerBegin(0, 80, true);
    // Creates a timer handle. Parameters: 0 - timer 0
                                           80 - prescaler value
                                           true - count upwards

timerAttachInterrupt(Timer0_Cfg, &Timer0_ISR, true);
    // Attaches the ISR to the timer. Please note the &!
                              true means trigger on EDGE, not on FLAT


timerAlarmWrite(Timer0_Cfg, 1000, true);
    // Sets the interval for the event, in "ticks"
                              true - the timer will auto-reload!
                              false - the event will happen only once

timerAlarmEnable(Timer0_Cfg);
    // start!
```

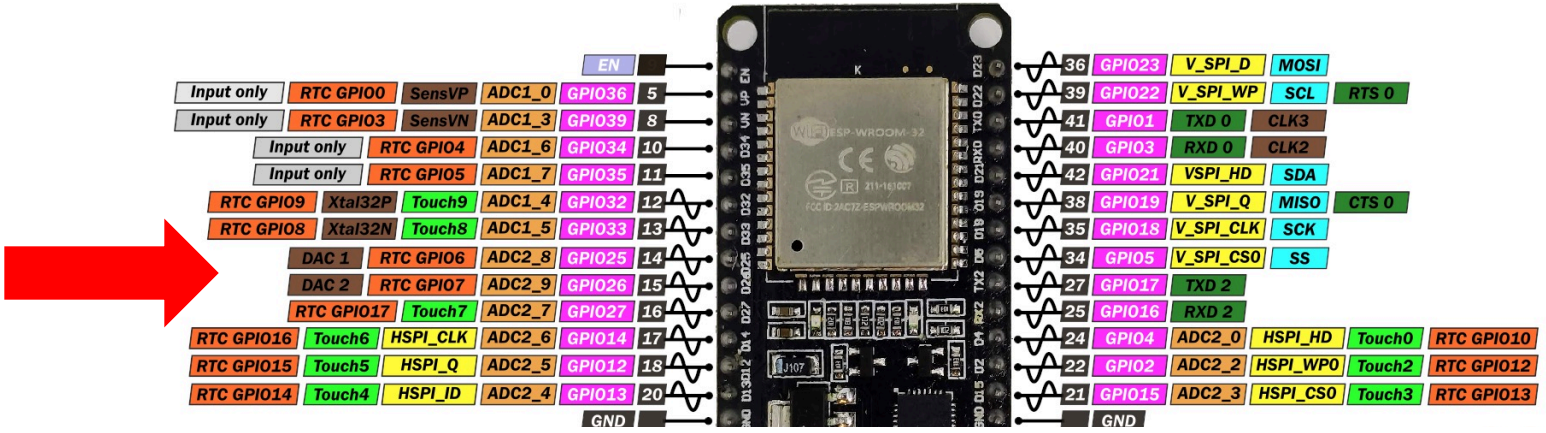
https://deepbluembedded.com/esp32-timers-timer-interrupt-tutorial-arduino-ide/

Timers and DAC

Signal generation using DAC - Digital to Analog Converter

- **2 DAC channels available**
- **8 bit resolution**
- **Output between 0 and 3.3 V**

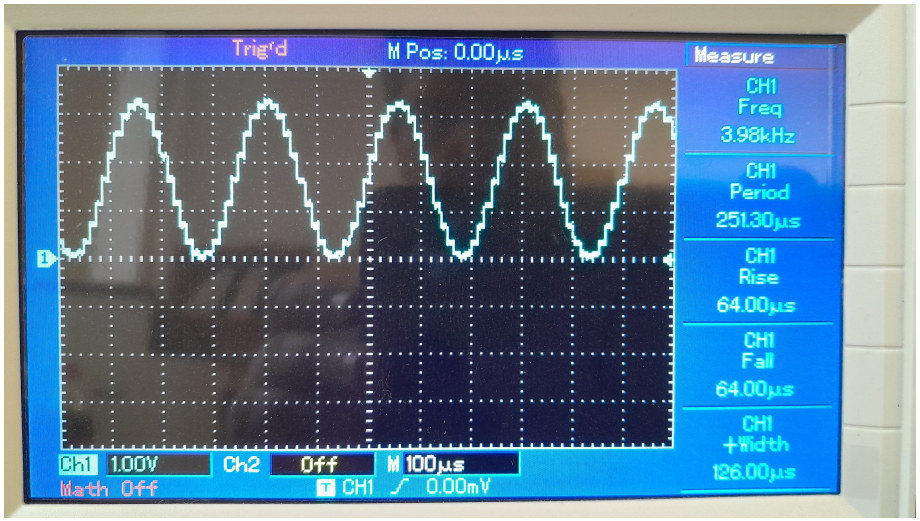
$$DAC(OutputVoltage) = DigitalVal \times \frac{3.3}{255}$$



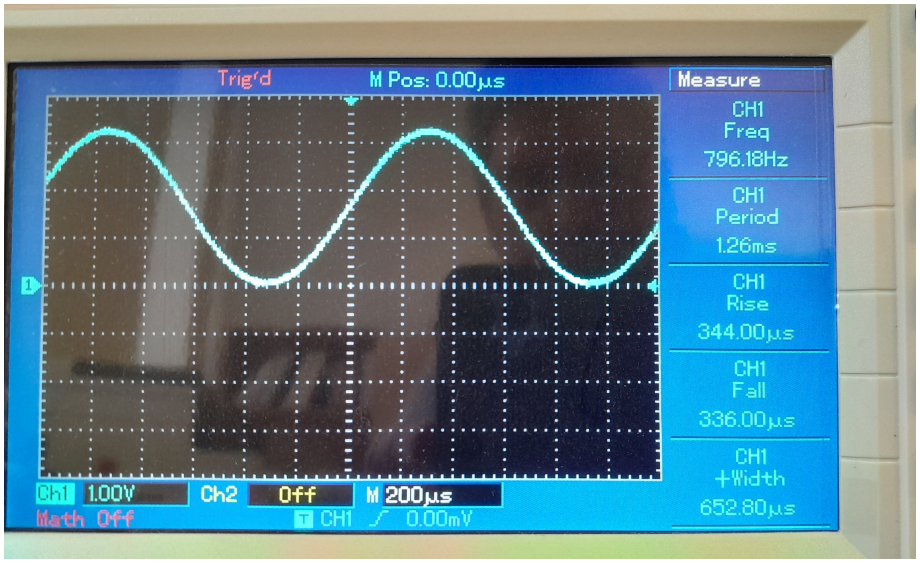
Timers and DAC

```
#include <driver/dac.h>

const int N = 20; // resolution of the sine signal (how many samples per period)
volatile int SampleIdx = 0; // current position in the sample LUT
int sinSamples[N];   // the sample LUT
hw_timer_t *Timer0_Cfg = NULL;

void IRAM_ATTR Timer0_ISR()
{
     // Send sine Values To DAC One By One
     dac_output_voltage(DAC_CHANNEL_1, sinSamples[SampleIdx++]);
     if(SampleIdx == N)
          SampleIdx = 0;
}

void setup()
{ for (int i=0; i<N; i++)
     {
       float s = (sin ((float)i/N*2*PI)+1)/2 * 255;
       sinSamples[i] = (int)s;
     }
Timer0_Cfg = timerBegin(0, 2, true);
timerAttachInterrupt(Timer0_Cfg, &Timer0_ISR, true);
timerAlarmWrite(Timer0_Cfg, 500, true);
timerAlarmEnable(Timer0_Cfg);

dac_output_enable(DAC_CHANNEL_1);
}

void loop(){}
```

Timers and DAC



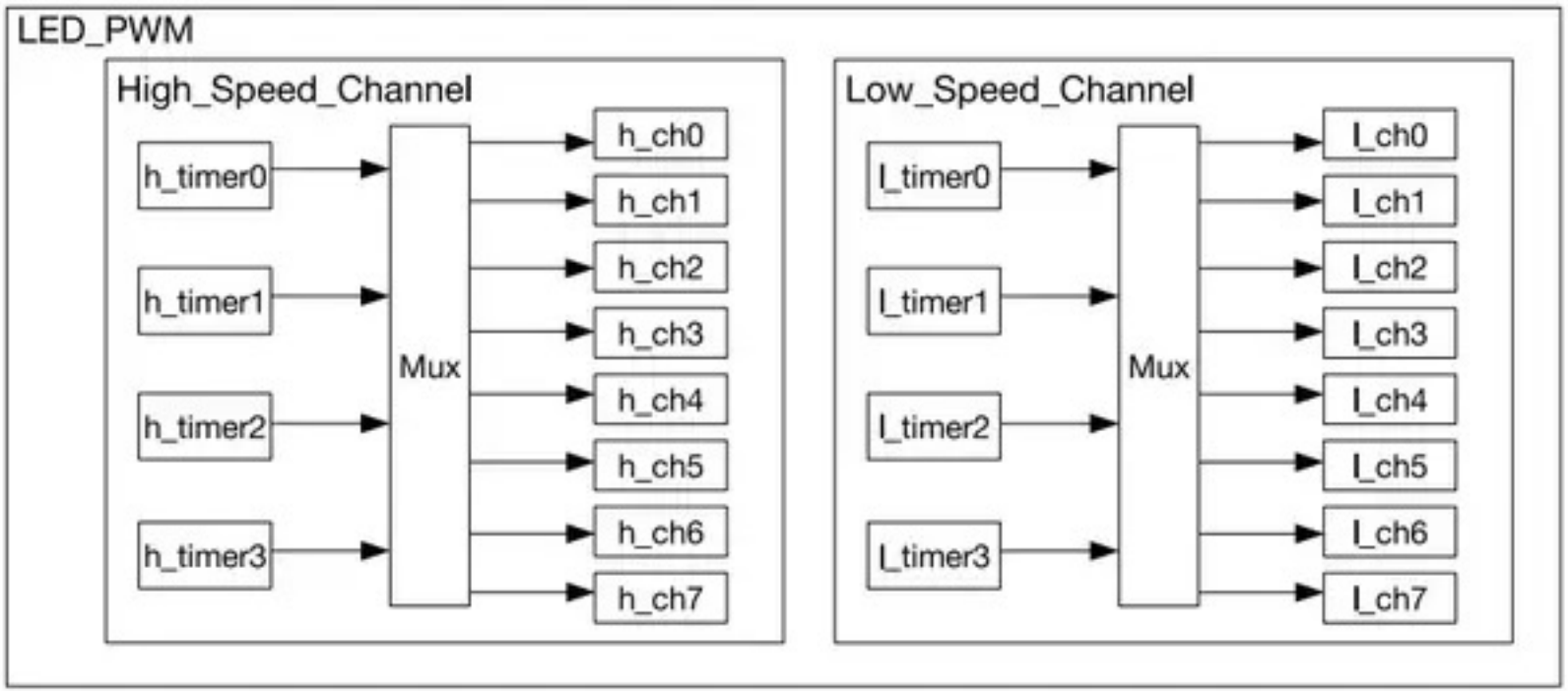
N = 20
f = 4 kHz



N = 100
f = 0.8 kHz

PWM signal generation

- 16 different PWM channels
- Each channel can be assigned to any GPIO pin
- The ESP32 PWM controller has 8 high-speed channels and 8 low-speed channels, which gives us a total of 16 channels
- Two channels share the same timer
  - Using the same timer means two PWM channels have the same frequency



https://deepbluembedded.com/esp32-pwm-tutorial-examples-analogwrite-arduino/

PWM signal generation

```
#define LED_GPIO   2      // LED pin
#define PWM1_Ch    0      // PWM channel
#define PWM1_Res   4      // values from 0 to 15
#define PWM1_Freq  1000   // PWM frequency

int PWM1_DutyCycle = 0;   // starting duty cycle

void setup()
{
  ledcAttachPin(LED_GPIO, PWM1_Ch);
  ledcSetup(PWM1_Ch, PWM1_Freq, PWM1_Res);
}

void loop()
{
  while(PWM1_DutyCycle < 15)
  {
    ledcWrite(PWM1_Ch, PWM1_DutyCycle++);
    delay(100);
  }
  while(PWM1_DutyCycle > 0)
  {
    ledcWrite(PWM1_Ch, PWM1_DutyCycle--);
    delay(100);
  }
}
```

PWM signal generation

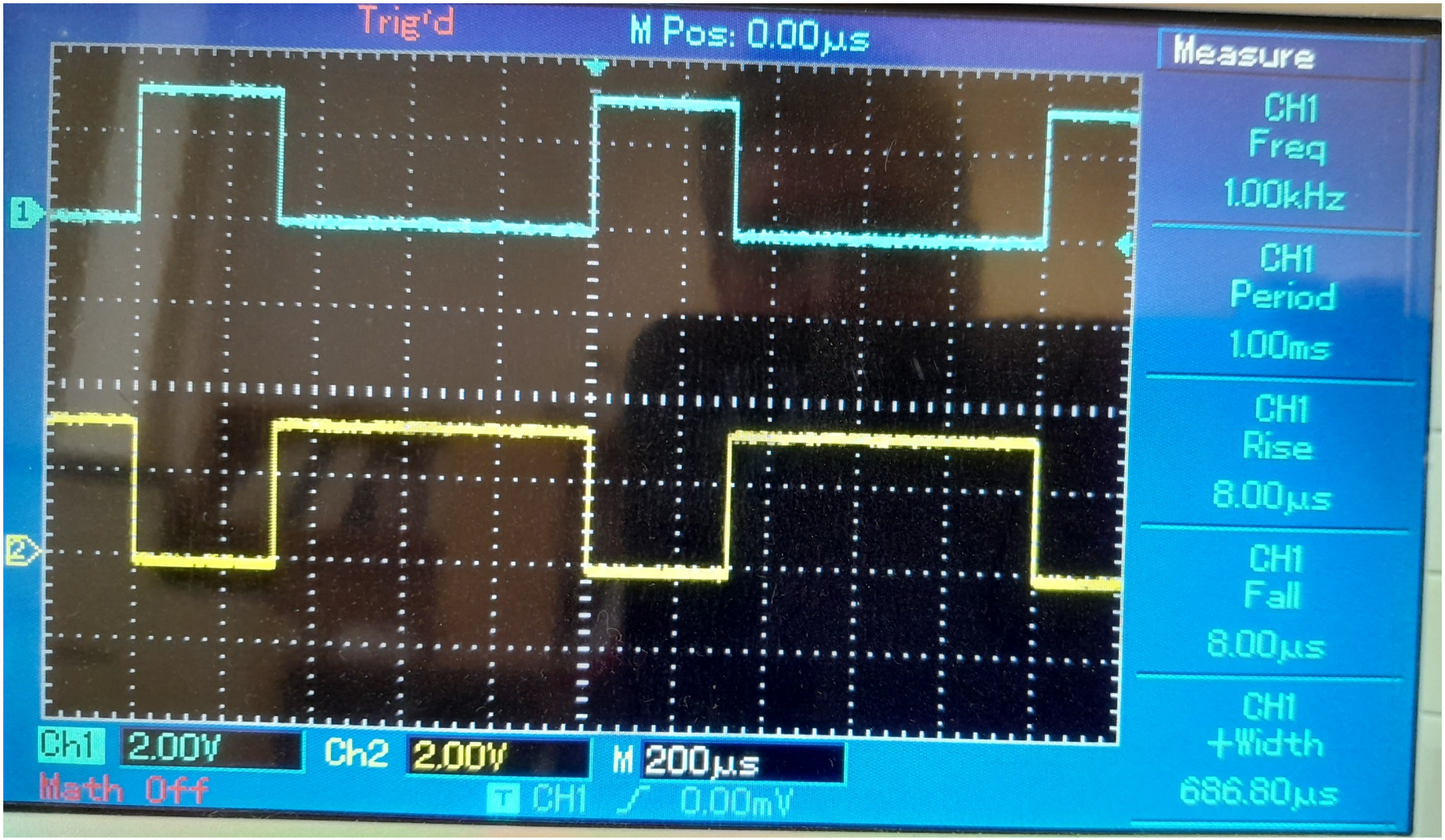
PWM signal generation

```
#define LED_GPIO   2       // LED pin
#define LED_GPIO_2   25    // Another LED pin

#define PWM1_Ch    0       // PWM channel
#define PWM1_Res   4       // values from 0 to 15
#define PWM1_Freq  1000    // PWM frequency

int PWM1_DutyCycle = 0;    // starting duty cycle

void setup()
{
  ledcAttachPin(LED_GPIO, PWM1_Ch);
  ledcAttachPin(LED_GPIO_2, PWM1_Ch);       // We can attach the PWM signal to
                                            // more than one pin!
  ledcSetup(PWM1_Ch, PWM1_Freq, PWM1_Res);
}


void loop()
{
… Same code as before
}
```

PWM signal generation

```
#define LED_GPIO   2       // LED pin
#define LED_GPIO_2   25    // Another LED pin

#define PWM1_Ch    0       // PWM channel
#define PWM1_Res   4       // values from 0 to 15
#define PWM1_Freq  1000    // PWM frequency

int PWM1_DutyCycle = 0;    // starting duty cycle

void setup()
{
  ledcAttachPin(LED_GPIO, PWM1_Ch);
  ledcAttachPin(LED_GPIO_2, PWM1_Ch);       // We can attach the PWM signal to
                                            // more than one pin!

  GPIO.func_out_sel_cfg[25].inv_sel = 1;    // We can even invert the output on
                                            // one pin!

  ledcSetup(PWM1_Ch, PWM1_Freq, PWM1_Res);
}

void loop()
{
… Same code as before
}
```

PWM signal generation



Measuring time between events

- The timers can be read or written, started and stopped
- The timer counts the number of “ticks” (increments) in a 64-bit register
- The time between ticks depends on the prescaler (the clock source for the timer)

$$T_{OUT} = TimerTicks \times \frac{Prescaler}{APB\_CLK}$$

- Frequency of a measured signal: f = 1 / T_{OUT}

$$f = \frac{APB\_CLK}{Prescaler\ x\ TimerTicks}$$

Measuring time between events

```
#define IN_SIGNAL_GPIO 25
hw_timer_t *Timer0_Cfg = NULL;
bool Measurement_InProgress = false;
uint64_t Measured_Time = 0;
uint64_t Measured_Freq = 0;

void IRAM_ATTR Ext_INT1_ISR()
{
    if(Measurement_InProgress == false)
    {
      Measurement_InProgress = true;
      timerWrite(Timer0_Cfg, 0);
      timerStart(Timer0_Cfg);
    }
    else
    {
      Measured_Time = timerRead(Timer0_Cfg);
      timerStop(Timer0_Cfg);
      Measurement_InProgress = false;
      if(Measured_Time == 0)
      {
        Measured_Freq = 0;
      }
      else
      {
        Measured_Freq = (40000000/Measured_Time);
      }
    }
}
```

Adapted from https://deepbluembedded.com/esp32-timers-timer-interrupt-tutorial-arduino-ide/

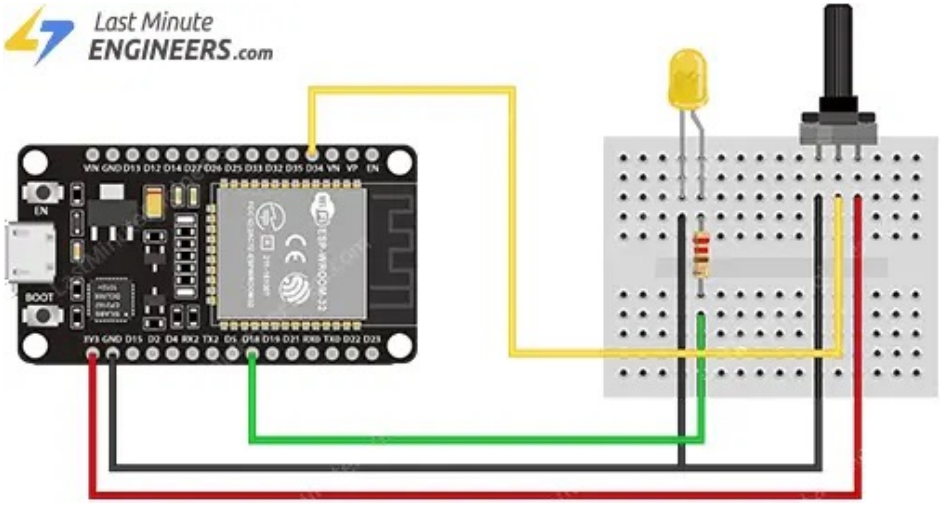
Measuring time between events

```
void setup()
{
    // Initialize The I2C LCD
    Serial.begin(115200);

    // Enable External Interrupt Pin
    pinMode(IN_SIGNAL_GPIO, INPUT);
    attachInterrupt(IN_SIGNAL_GPIO, Ext_INT1_ISR, RISING);
    // Configure Timer0
    Timer0_Cfg = timerBegin(0, 2, true);
}

void loop()
{

    Serial.print(Measured_Freq);
    Serial.println(" Hz");

    delay(250);
}
```

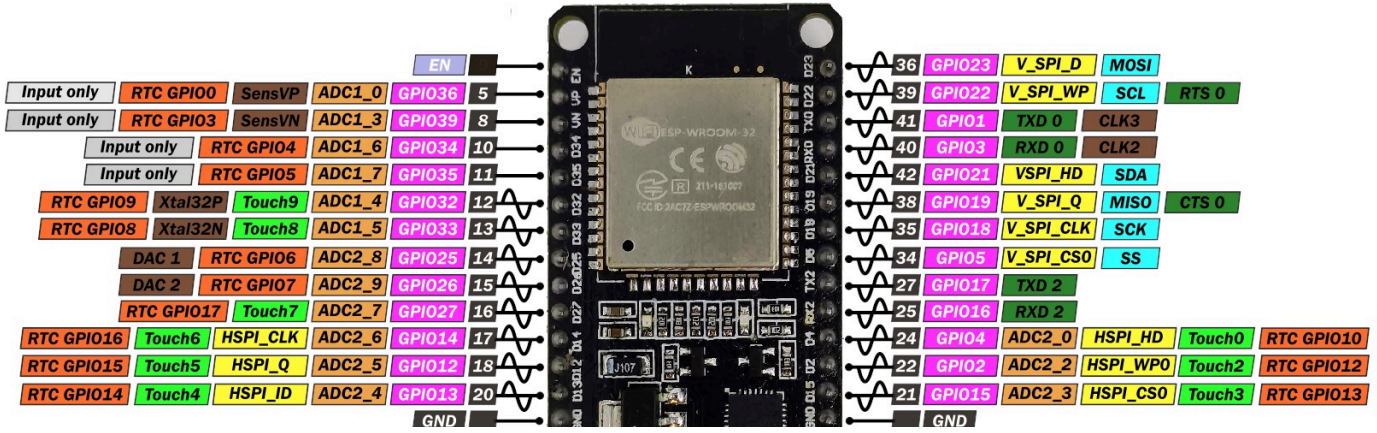
- The code was tested using the Oscilloscope's **1KHz**, 3V signal
- The measured frequency was **1001 Hz**.

https://deepbluembedded.com/esp32-timers-timer-interrupt-tutorial-arduino-ide/

Reading analog signals



From https://lastminuteengineers.com/esp32-pwm-tutorial/?utm_content=cmp-true



Reading analog signals

```
const int PWM_CHANNEL = 0;    // ESP32 has 16 channels which can generate 16 independent
waveforms
const int PWM_FREQ = 500;     // To be similar to Arduino's 490 Hz
const int PWM_RESOLUTION = 8; // We'll use same resolution as Uno (8 bits, 0-255) (max is 16)

// The max duty cycle value based on PWM resolution (will be 255 if resolution is 8 bits)
const int MAX_DUTY_CYCLE = (int)(pow(2, PWM_RESOLUTION) - 1);
const int LED_OUTPUT_PIN = 18;
const int POT_PIN = 34;
const int DELAY_MS = 100;  // delay between fade increments

void setup() {

  // Sets up a channel (0-15), a PWM duty cycle frequency, and a PWM resolution (1 - 16 bits)
  ledcSetup(PWM_CHANNEL, PWM_FREQ, PWM_RESOLUTION);
  ledcAttachPin(LED_OUTPUT_PIN, PWM_CHANNEL);
}

void loop() {
  int dutyCycle = analogRead(POT_PIN);
  dutyCycle = map(dutyCycle, 0, 4095, 0, MAX_DUTY_CYCLE);
  ledcWrite(PWM_CHANNEL, dutyCycle);

  delay(DELAY_MS);
}
```

From https://lastminuteengineers.com/esp32-pwm-tutorial/?utm_content=cmp-true

Summary

- **Most useful interfaces can be mapped to any pin, using the GPIO matrix**
- **External interrupts can be attached to any pin**
- **An ISR that takes too much time will trigger the WatchDog Timer and reboot the system**
- **Timers can be used to generate events and signals**
- **Timers can be used to measure the time between events**
- **The generated signals can be mapped to any pin**
- **The pin value can be inverted**
- **ESP32 can generate analog signals on dedicated DAC pins**
- **Analog signals can be read from most GPIO pins, using a 12-bit resolution ADC**