# ESP32 Background Information

This document is here to provide some useful information for using the ESP32 for EECS300 labs.

### Peripherals

A peripheral is a piece of hardware that **performs a specific function**. For example, a timer peripheral is tasked with accurately incrementing a counter based on some specified real-time rate. Another example is the analog to digital converter (ADC), which is a common peripheral that reads in an analog voltage signal and converts it to an X-bit digital signal.

Peripherals must first be configured in code, then can be accessed using the provided ESP32 libraries. We have written interfaces for some of the ESP32 peripherals (see table 1), but you can consult the ESP32 datasheet, linked below, for the full list.

[Link to ESP32 datasheet](https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf)

The API for interfacing with these, and generally running the ESP32 can be found here:

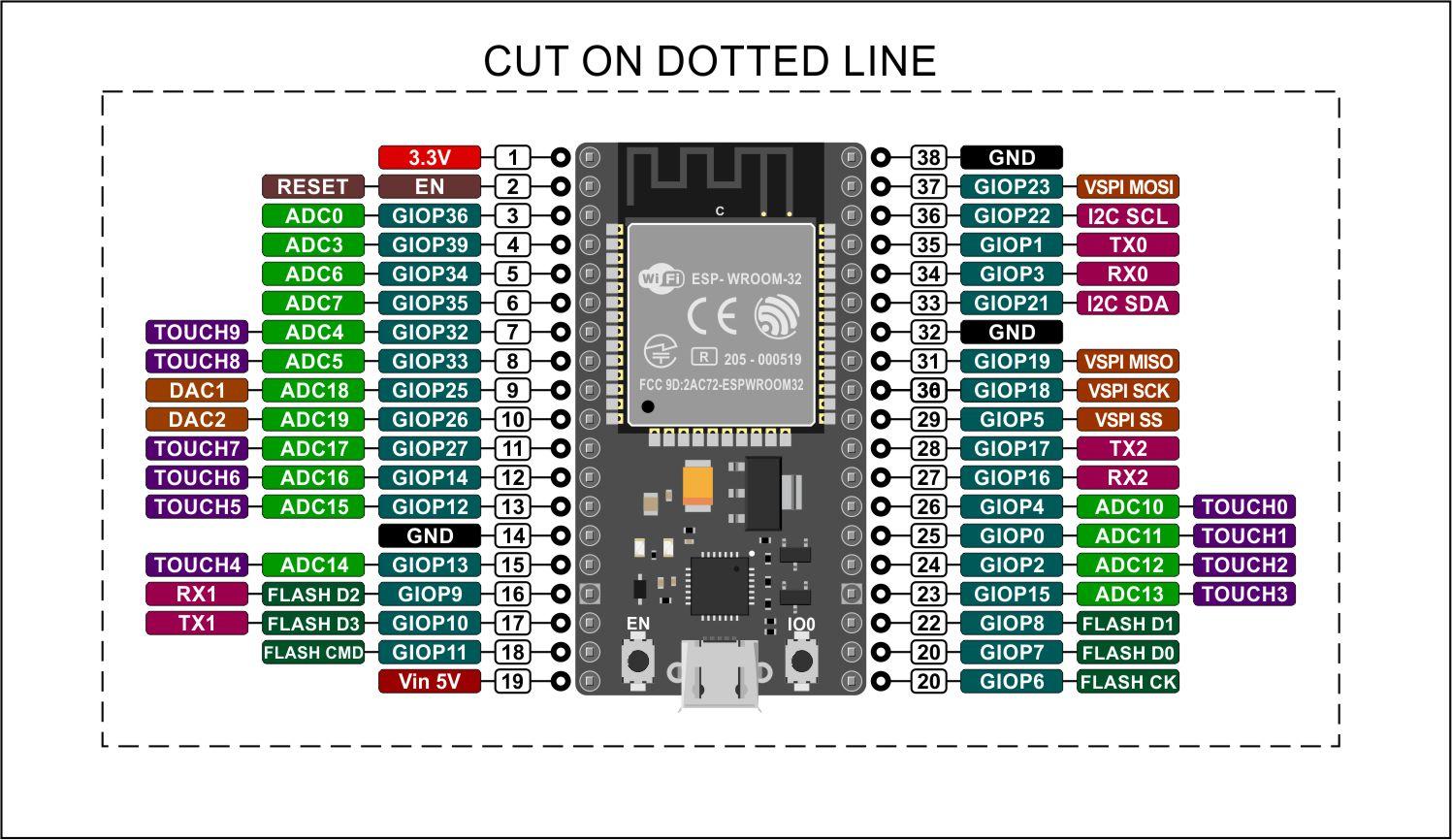
[Link to ESP32 API documentation](https://docs.espressif.com/projects/esp-idf/en/latest/esp32/index.html)

There are many peripherals on the ESP32, and the Arduino core provides a layer of abstraction for several features. In addition, we have added support for some features in “hal.cpp”/”hal.h” (see *table 1*). The functions provided by the Arduino core and the HAL will be all you need for the labs, but you are welcome to explore configuring more for the final project by extending the HAL. Many peripherals only can use specific pins, these are generally pre-defined in the HAL, so check that file to see what pins to use. Otherwise, appropriate pins can be found in *table 1.* Note, when we refer to pin numbers, we are referring to the GPIO pin number.

*Table 1*

| **Peripheral** | **Configuration Notes** | **Description & Use** |
| --- | --- | --- |
| Analog to Digital Converter (ADC) & Direct Memory Access (DMA) (1X) | Can either be read directly or set to store values in memory using DMA, which then can be read using i2s protocol (the latter is not implemented in the HAL). Uses timer 3. | Read analog voltage values between 0 and 3.3V from pin 36. |
| Digital to Analog Converter (DAC) (2X) | Basic operation | Write analog voltage values between 0 and 3.3V to pins 18 or 19. |
| Timer (4X) | Timers 1 & 2 are used for UART channels 1 & 2, but otherwise are free to use. Timers are numbered 0-3. | Generates a periodic interrupt/ |
| Universal Asynchronous Receiver/Transmitter (UART) (2X) | Free for any use. UART channels are labeled 1 & 2. | Serial communication peripheral. |

There are many different ESP32 development boards, we are using the HiLetgo ESP-WROOM-32 ESP32. The pinout for this board is below. Don’t use any other pinout you find online, as different breakout boards have different configurations, and you may fry your chip.



*Figure 1: ESP32 Pinout*

### Interrupts

The ESP32 (and almost any modern microcontroller) has support for Interrupts. This allows the microcontroller to be interrupted by time-critical events so that it can quickly respond and resume what it was doing.

An “Interrupt Service Routine (ISR)”, is a function that is called when an interrupt is triggered. These triggers could be a timer peripheral completing its count cycle, a button being pressed, a signal being received etc.

Interrupts typically occur *asynchronously*, which means they can happen at any point of the main program (in other words, they will not wait for a convenient time to trigger). Therefore, we have to be careful how we deal with data shared by both the main program and a given interrupt to avoid issues such as trying to write two different values to a variable simultaneously. We will introduce two topics involving the synchronization of interrupts and the main program called *critical sections* and the *volatile* keyword.

### Volatile

C's volatile keyword is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time--without any action being taken by the code the compiler finds nearby. The implications of this are quite serious. However, before we examine them, let's take a look at the syntax.

To declare a variable volatile, include the keyword volatile before or after the data type in the variable definition. For instance both of these declarations will declare an unsigned 16-bit integer variable to be a volatile integer:

volatile uint16\_t x;

uint16\_t volatile y;

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

1. Memory-mapped peripheral registers

2. Global variables modified by an interrupt service routine

3. Global variables accessed by multiple tasks within a multi-threaded application

Consider the following pseudo code. The intended functionality of the code is to receive characters from a serial port and place them into a global buffer global\_buffer[]. Once the buffer is full of characters, the program will print the contents of the buffer. The ISR rx\_isr runs every time a character is received, and the main()function has a while loop that continuously checks whether or not the buffer is full.

char global\_buffer[5];

bool global\_buffer\_full = false;

void main()

{

while (1)

{

if(global\_buffer\_full == true)

{

printBuffer();

}

}

}

interrupt void rx\_isr(void)

{

static int buffer\_index = 0;

if(is\_char\_received)

{

if(buffer\_index == 4)

{

buffer\_index = 0;

}

else

{

++buffer\_index;

}

global\_buffer[buffer\_index] = received\_char;

if(buffer\_index == 4)

{

global\_buffer\_full = true;

}

}

}

With compiler optimization turned off, this program might work. However, any half decent optimizer will "break" the program. The problem is that the compiler has no idea that global\_buffer\_full or global\_buffer[] can be changed within the ISR function. As far as the compiler is concerned. The compiler thinks that these may as well be local variables. Furthermore, it will think that they will never change, and the compiler will optimize them away.

The solution is to declare the two as volatile. This tells the compiler that they can’t be touched by the optimizer.

There is a second issue with the code. Suppose that the ISR function has just set global\_buffer\_full to true and exited. What happens if we receive another character, and the interrupt fires again before the buffer is printed? Global\_buffer[] will be changed and the wrong value will be printed.

This introduces us to the next topic: critical sections.

### Critical Sections

A critical section is a section of code that should not be interrupted.

For example, this was a critical section:

if(global\_buffer\_full == true)

{

printBuffer();

}

A simple technique to “protect” this critical section (and others) is to temporarily disable interrupts. Here is the fixed pseudo code:

blockInterrupts();

if(global\_buffer\_full == true)

{

printBuffer();

}

unBlockInterrupts();

Keep in mind that this method, while simple, is also a bit crude. We only need to stop the rx\_isr() interrupt from firing, not all of them. Other potentially important interrupts could get delayed/ignored due to this. Operating systems (and other code platforms) typically offer more advanced capabilities for handling critical sections.

### HAL

The Hardware Abstraction Layer, HAL, is a library that interfaces with our given embedded chip (ESP32 in this case) using device-specific code and abstracts its functions to device-agnostic code. The HAL allows you to make most of your code compatible with many different embedded platforms. In other words, it allows for more generic code. If you correctly implement a HAL, then when changing between compatible platforms the only thing you should need to edit is the HAL itself. We have written a HAL for the ESP32, which you should use.

### Timers

A timer is a simple, but also a very useful peripheral found in almost any microcontroller.

The ESP32 has 4 timers.

[Here is a useful resource which explains how they work](https://diyprojects.io/esp32-timers-alarms-interrupts-arduino-code/#.YTqopWZKhQI)

Note that the HAL functions we provide set the prescaler to 80, which is why the count frequency is 1 Mhz.

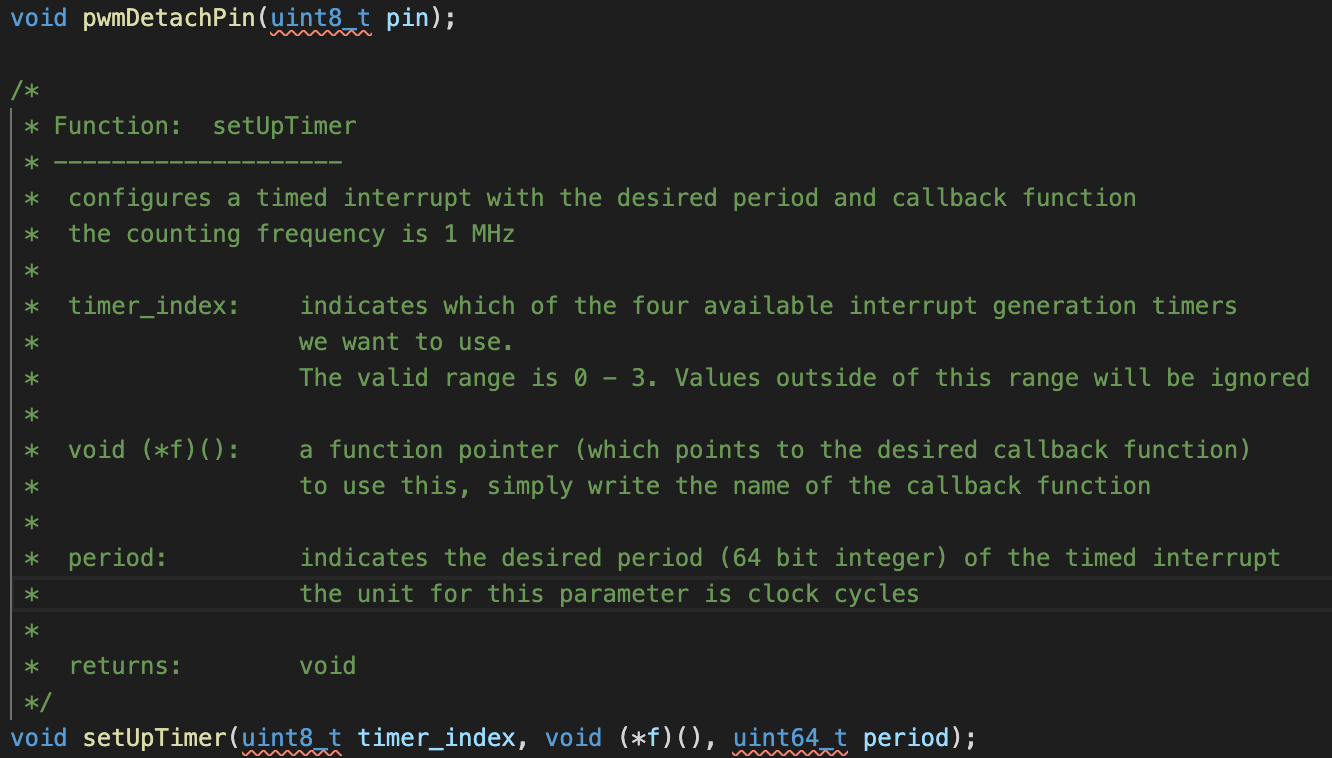
Note that some features rely on a timer for operation. As a result, you may not be able to use certain timers and certain features at the same time. This is described in Table 1.

#### How to use the HAL functions for the timers

You need to use two functions to configure and start a timer:

1. setUpTimer(uint8\_t timer\_index, void (\*f)(), uint64\_t period)
2. startTimer(uint8\_t timer\_index)

We can use the documentation from the HAL in order to figure out how to use it.



*Figure 2: setupTimer() from the HAL*

Say we want to use timer 0 to generate interrupts to call the function timer0ISR() at 10 kHz.

We know that we have to pass in 0 to timer\_index indicate that we want to use timer 0

For the second parameter, we need to pass in a function pointer (which is just the name of the function we want to be called) - in this case it is timer0ISR().

Lastly we need to calculate the value for the period parameter.

We know that the unit for this in clock cycles, and the clock is running at 1 MHz

We can also calculate the period of our desired 10 kHz interrupt rate

We need to know how many clock cycles is equal to one period at 10Khz

So now we know that we need to pass in 100 for the period parameter

Here is the resulting code to setup the timed interrupt we want:

setupTimer(0, timer0ISR, 100);

startTimer(0);

### ESP32 ADC

The ESP32 has 2 12-bit ADC channels.

The ESP32’s ideal transfer function (the equation to convert from the digital reading to a voltage) is the same as the one depicted the following picture:

#### How to calculate the ESP32’s Resolution and Transfer Function

First, let us calculate the resolution, which is equivalent to the size of the LSB (least significant bit). This is because the LSB is the smallest amount the reading can increment or decrement.

Since the ESP32 ADC has N = 12 bits, and VREF is 3.3v, we can calculate the size of the LSB with

Next, we want to find the transfer function. This actually depends on the type of ADC (e.g. straight binary, ½ LSB compensated). For the ESP32, the formula we want to use is

Where reading is the binary value the adc returns

You can think of this as simply scaling from (0 - 4096) to (0 - 3.3)

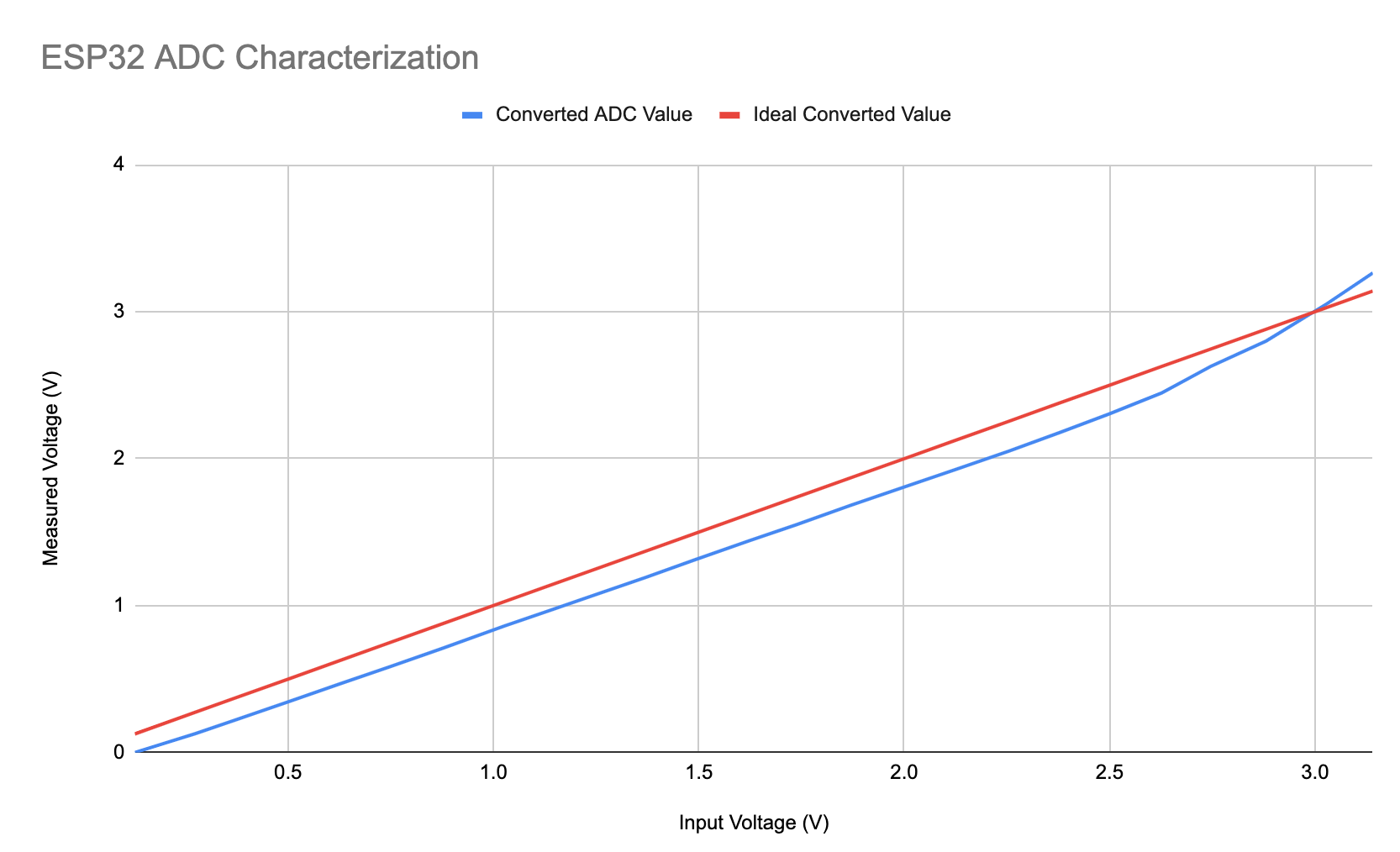
Note: it is 4096, not 4095 because this is ½ LSB compensated ADC, but you don’t need to worry about this. However, not all ADCs follow this. Some will have different transfer functions

#### Non-Ideal Behavior of the ESP32 ADC

The ESP32 ADC is far from perfect. Unfortunately, the deviance from ideal behavior is not consistent. In other words, different ESP32s will have slightly different characteristics. This is analogous to cars coming out of the factory. They all will perform within certain specifications, but some will be a bit faster than others.

Here is a chart showing the behavior of the ESP32 ADC vs Ideal

*Figure 3: ESP32 ADC Characterization*



This is far from ideal, but we can account for this in our software. However, every ADC is slightly unique. This is analogous to Ford F150s coming out of a factory. They are all tested to meet the same standards, but some will be slightly faster than others. Integrated circuit production is no different; different ADCs (even on the same microcontroller) will have different amounts of precision.

For this reason, we must calibrate every ADC we want to use individually (if we need the precision). We will no required any sort of calibration in lab, however if you project requires better performance, refer to this project for guidance: [ESP32 ADC Calibrate](https://github.com/e-tinkers/esp32-adc-calibrate)