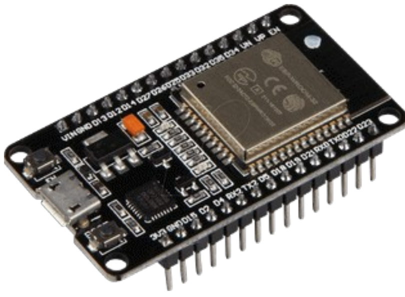
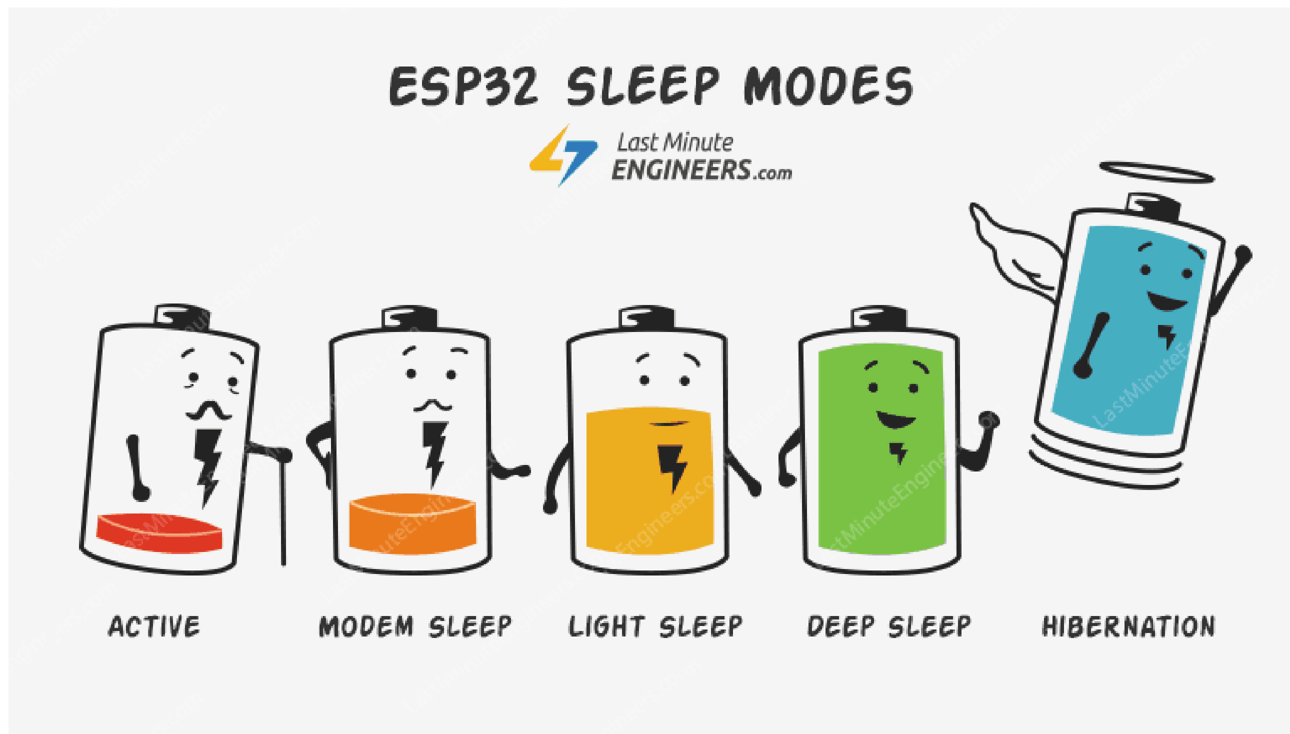


Part 06
-
Sleep Modes

Version: 2021-03-25

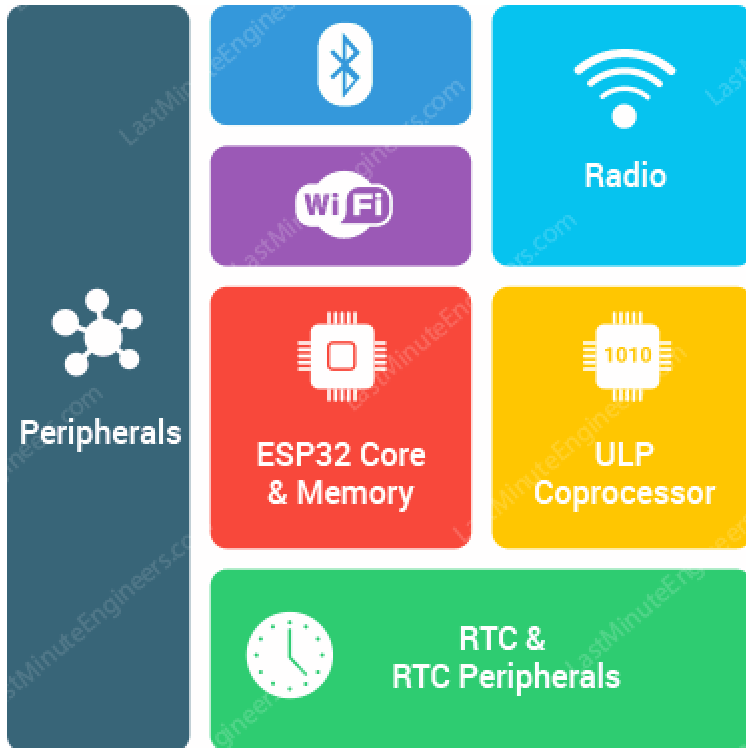
Insight Into ESP32 Sleep Modes & Their Power Consumption



There is no question that ESP32 is a worthy competitor to many WiFi/MCU SoCs out there, often beating it on both performance and price. But, depending on which state it's in, the ESP32 can be a relatively power-hungry device.
When your IoT project is powered by a plug in the wall, you tend not to care too much about power consumption. But if you are going to power your project by batteries, every mA counts. The solution here is to cut back ESP32's power usage by leveraging one of its Sleep Modes. It's really a great strategy for dramatically extending the battery life of a project that doesn't need to be active all the time.

What is ESP32 sleep mode?
ESP32 Sleep mode is a power-saving state that ESP32 can enter when not in use. The ESP32's state is maintained in RAM. When ESP32 enters sleep mode, power is cut to any unneeded digital peripherals, while RAM receives just enough power to enable it to retain its data.

Inside ESP32 chip
In order to understand how ESP32 achieves power saving, we need to know what's inside the chip. The following illustration shows function block diagram of ESP32 chip.

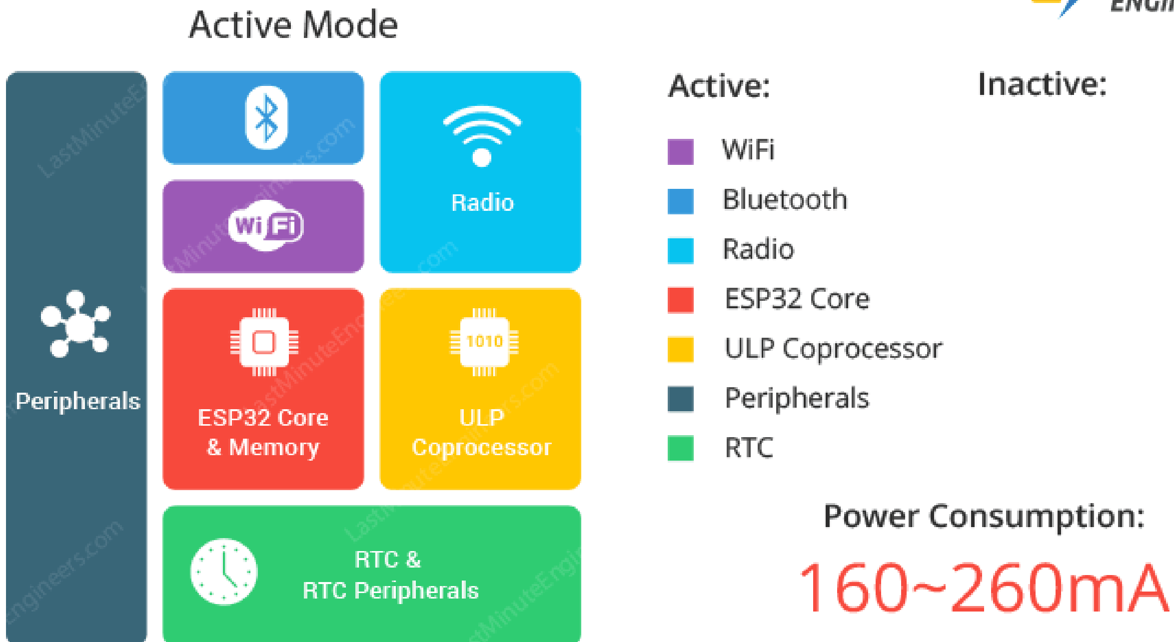


At the heart of the ESP32 chip is a Dual-Core 32-bit microprocessor along with 448 KB of ROM, 520 KB of SRAM and 4MB of Flash memory.
It also contains WiFi module, Bluetooth Module, Cryptographic Accelerator (a co-processor designed specifically to perform cryptographic operations), the RTC module, and lot of peripherals.
ESP32 Power Modes
Thanks to the ESP32's advanced power management, it offers 5 configurable power modes. As per the power requirement, the chip can switch between different power modes. The modes are:

- Active Mode
- Modem Sleep Mode
- Light Sleep Mode
- Deep Sleep Mode
- Hibernation Mode

Each mode has its own distinct features and power saving capabilities. Let's look in to them one by one.

ESP32 Active Mode
The normal mode is also known as Active Mode. In this mode all the features of the chip are active.
As the active mode keeps everything (especially the WiFi module, the Processing Cores and the Bluetooth module) ON at all times, the chip requires more than 240mA current to operate. Also we observed that if you use both WiFi and Bluetooth functions together, sometimes high power spikes appear (biggest was 790mA).

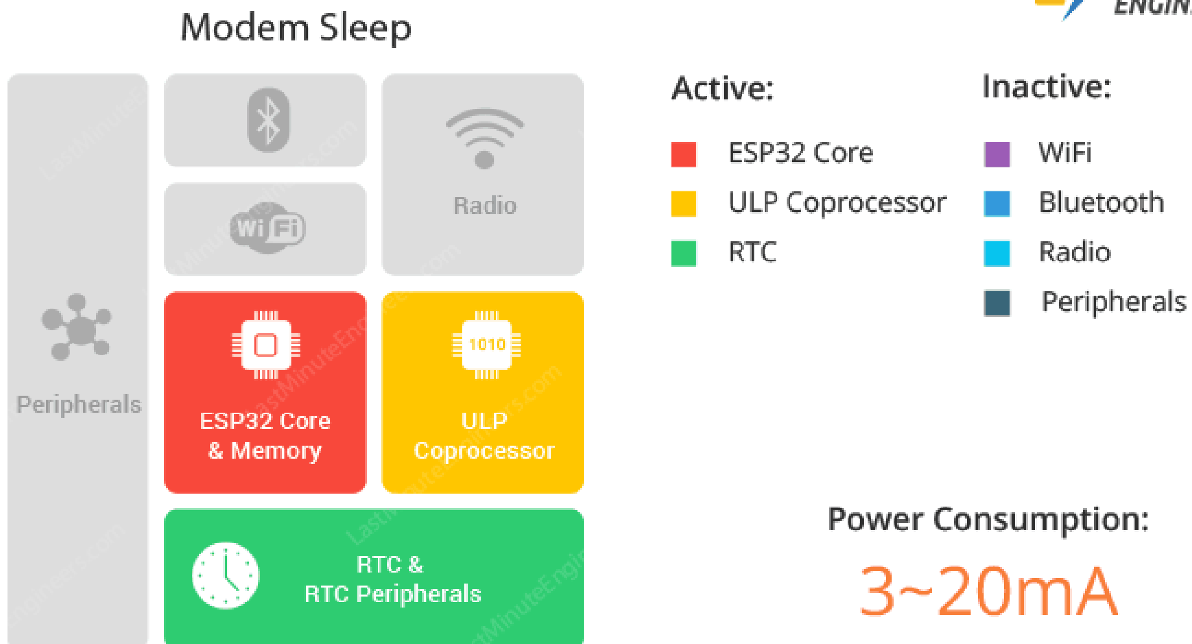


If you look at the ESP32 datasheet, power consumption during Active power mode, with RF working is as follows:

Mode	Power Consumption
Wi-Fi Tx packet 13dBm~21dBm	160~260mA
Wi-Fi/BT Tx packet 0dBm	120mA
Wi-Fi/BT Rx and listening	80~90mA

Obviously, this is the most inefficient mode and will drain the most current. So, if we want to conserve power we have to disable them (by leveraging one of the other power modes) when not in use.

ESP32 Modem Sleep
In modem sleep mode everything is active while only WiFi, Bluetooth and radio are disabled.
The CPU is also operational and the clock is configurable.
In this mode the chip consumes around 3mA at slow speed and 20mA at high speed.



To keep WiFi/Bluetooth connections alive, the CPU, Wi-Fi, Bluetooth, and radio are woken up at predefined intervals. It is known as Association sleep pattern.
During this sleep pattern, the power mode switches between the active mode and Modem sleep mode.
ESP32 can enter modem sleep mode only when it connects to the router in station mode.
ESP32 stays connected to the router through the DTIM beacon mechanism.
In order to save power, ESP32 disables the Wi-Fi module between two DTIM Beacon intervals and wakes up automatically before the next Beacon arrival.
The sleep time is decided by the DTIM Beacon interval time of the router which is usually 100ms to 1000ms.

What is DTIM beacon mechanism?
DTIM is acronym for Delivery Traffic Indication Message.
In this mechanism, the access point(AP)/router transmits beacon frames periodically. Each frame contains all the information about the network. It is used to announce the presence of a wireless network and synchronize all the connected members.

To stop the Bluetooth modem:

```
btStop()
```

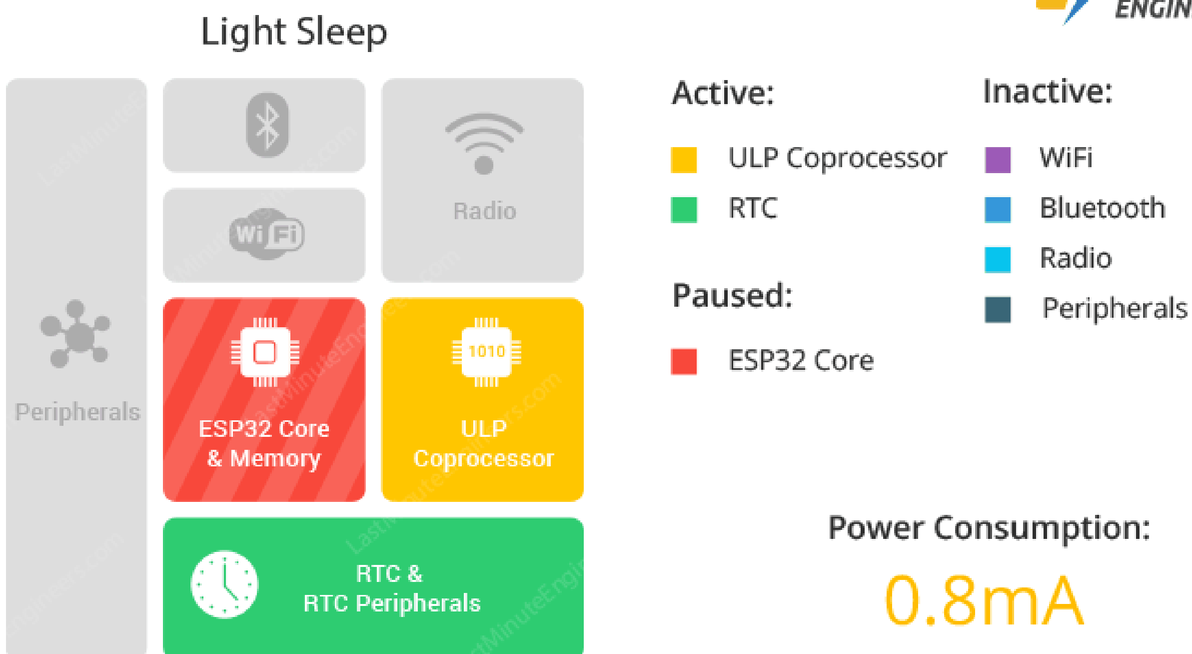
To stop the WiFi modem:

```
WiFi.mode(WIFI_OFF)
```

ESP32 Light Sleep
The working mode of light sleep is similar to that of modem sleep. The chip also follows association sleep pattern.
The difference is that, during light sleep mode, digital peripherals, most of the RAM and CPU are clock-gated.

What is Clock Gating?
Clock gating is a technique for reducing the dynamic power consumption.
It disables portions of the circuitry by powering off clock pulses, so that the flip-flops in them do not have to switch states. As switching states consumes power, when not being switched, the power consumption goes to zero.

During light sleep mode, the CPU is paused by powering off its clock pulses, while RTC and ULP-coprocessor are kept active. This results in less power consumption than in modem sleep mode which is around 0.8mA.



Before entering light sleep mode, ESP32 preserves its internal state and resumes operation upon exit from the sleep. It is known Full RAM Retention.
`esp_light_sleep_start()` function can be used to enter light sleep once wake-up sources are configured.

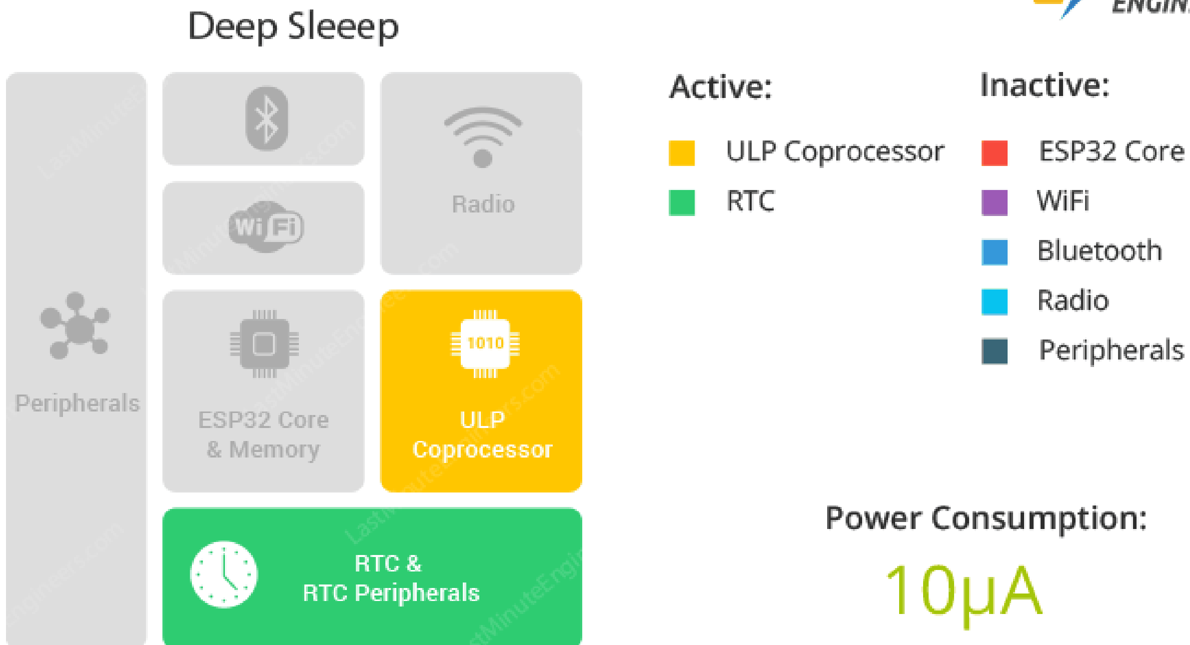
In setup:

```
// allow timer to wake-up
esp_sleep_enable_timer_wakeup(SleepSecs * uS_TO_S_FACTOR);
//allow button press to wake-up
//Pin pulled high => 1 = Low to High wakeup, 0 = High to Low wakeup.
esp_sleep_enable_ext0_wakeup(WAKEPIN,0);
delay(100);
esp_light_sleep_start();
```

ESP32 Deep Sleep

In deep sleep mode, the CPU, most of the RAM and all the digital peripherals are powered off. The only parts of the chip that remains powered on are: RTC controller, RTC peripherals (including ULP co-processor), and RTC memories (slow and fast).

The chip consumes around 0.15 mA(if ULP co-processor is powered on) to 10µA.



During deep sleep mode, the main CPU is powered down, while the ULP co-processor does sensor measurements and wakes up the main system, based on the measured data from sensors. This sleep pattern is known as ULP sensor-monitored pattern.

Along with the CPU, the main memory of the chip is also disabled. So, everything stored in that memory is wiped out and cannot be accessed.

However, the RTC memory is kept powered on. So, its contents are preserved during deep sleep and can be retrieved after we wake the chip up. That's the reason, the chip stores Wi-Fi and Bluetooth connection data in RTC memory before disabling them.

So, if you want to use the data over reboot, store it into the RTC memory by defining a global variable with `RTC_DATA_ATTR` attribute. For example, `RTC_DATA_ATTR int bootCount = 0;`

In Deep sleep mode, power is shut off to the entire chip except RTC module. So, any data that is not in the RTC recovery memory is lost, and the chip will thus restart with a reset. This means program execution starts from the beginning once again.

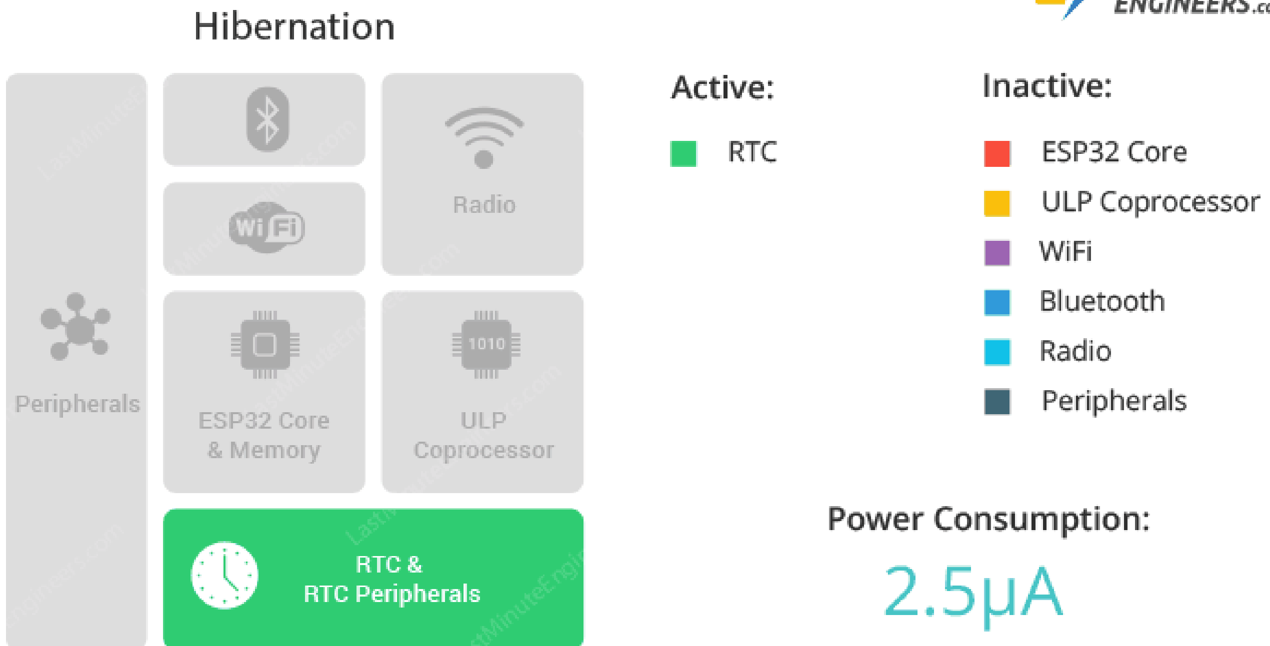
TIP

> ESP32 supports running a `deep sleep wake stub` when coming out of deep sleep. This function runs immediately as soon as the chip wakes up – before any normal initialization, bootloader code has run. After the wake stub runs, the chip can go back to sleep or continue to start normally.

Unlike the other sleep modes, the system cannot go into Deep-sleep mode automatically. `esp_deep_sleep_start()` function can be used to immediately enter deep sleep once wake-up sources are configured.

By default, ESP32 will automatically power down the peripherals not needed by the wake-up source. But you can optionally decide what all peripherals to shut down/keep on. For more information, check out API docs.

ESP32 Hibernation mode
Unlike deep sleep mode, in hibernation mode the chip disables internal 8MHz oscillator and ULP-coprocessor as well. The RTC recovery memory is also powered down, meaning there's no way we can preserve any data during hibernation mode.
Everything else is shut off except only one RTC timer on the slow clock and some RTC GPIOs are active. They are responsible for waking up the chip from the hibernation mode.
This reduces power consumption even further. The chip consumes around 2.5μA only in hibernation mode.



	Method	Modem	Light	Deep	Hibernate
Timer	esp_sleep_enable_timer_wakeup(uint64_t *time_in_us*)	✓	✓	✓	✓
Touch	esp_sleep_enable_touchpad_wakeup()	✓	✓	✓	
Ext0	esp_sleep_enable_ext0_wakeup(gpio_num_t *gpio_num*, int *level*)	✓	✓	✓	
Ext1	esp_sleep_enable_ext1_wakeup(gpio_num_t *gpio_num*, int *level*)	✓	✓	✓	
UART	esp_sleep_enable_uart_wakeup(int uart_num)	✓	✓		
ULP	esp_sleep_enable_ulp_wakeup()	✓	✓	✓	

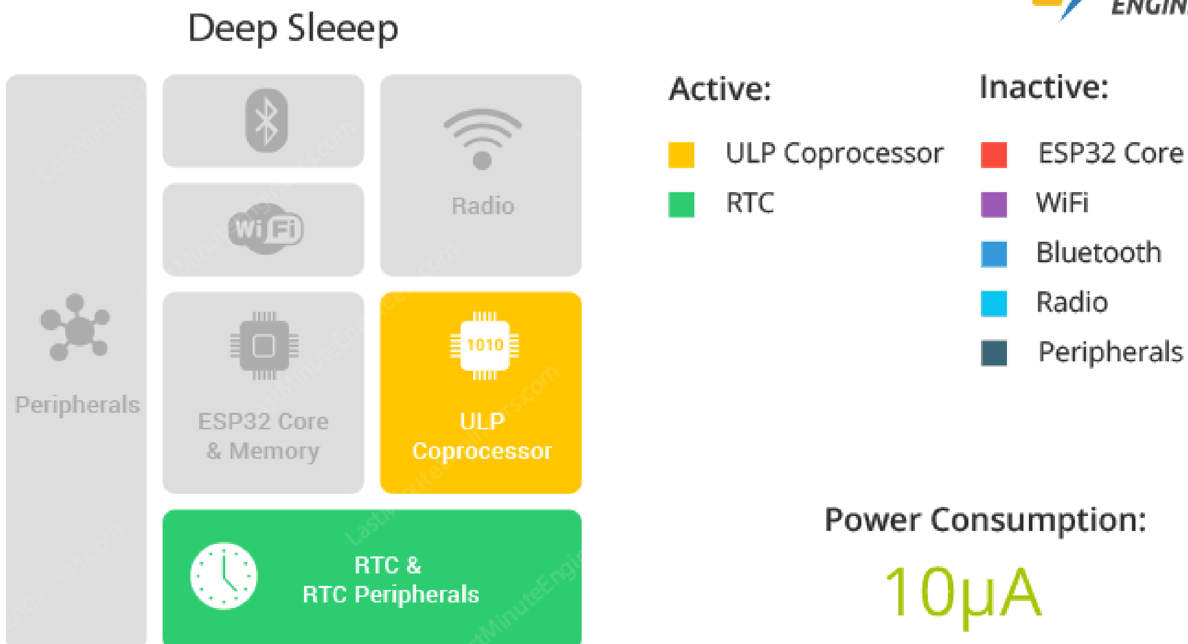
ESP32 Deep Sleep & Its Wake-up Sources

Have you ever wanted your IoT project to last on batteries for almost 5 years? Yes. It might sound ridiculous, but this is possible with the ESP32's deep sleep feature.

Why do ESP32 need deep sleep?
Depending on which state it's in, the ESP32 can be a relatively power-hungry device. It usually pulls about 75mA in normal operation and hits about 240mA while transmitting data over WiFi. When your IoT project is powered by a plug in the wall, you tend not to care too much about power consumption. But if you are going to power your project by batteries, every mA counts. The solution here is to cut back ESP32's power usage by leveraging Deep Sleep Mode.
To know more about other sleep modes of ESP32 and their power consumption, please visit below tutorial.

There is no question that ESP32 is a worthy competitor to many WiFi/MCU SoCs out there, often beating it on both performance and price. But,...

ESP32 Deep Sleep
In deep sleep mode, the CPU, most of the RAM and all the digital peripherals are powered off. The only parts of the chip that remains powered on are: RTC controller, RTC peripherals (including ULP co-processor), and RTC memories (slow and fast).
The chip consumes around 0.15 mA (if ULP co-processor is powered on) to 10µA.



During deep sleep mode, the main CPU is powered down, while the ULP co-processor does sensor measurements and wakes up the main system, based on the measured data from sensors. This sleep pattern is known as ULP sensor-monitored pattern.
Along with the CPU, the main memory of the chip is also disabled. So, everything stored in that memory is wiped out and cannot be accessed.
However, the RTC memory is kept powered on. So, its contents are preserved during deep sleep and can be retrieved after we wake the chip up. That's the reason, the chip stores Wi-Fi and Bluetooth connection data in RTC memory before disabling them.
So, if you want to use the data over reboot, store it into the RTC memory by defining a global variable with RTC_DATA_ATTR attribute. For example, `RTC_DATA_ATTR int bootCount = 0;`
In Deep sleep mode, power is shut off to the entire chip except RTC module. So, any data that is not in the RTC recovery memory is lost, and the chip will thus restart with a reset. This means program execution starts from the beginning once again.

ESP32 supports running a deep sleep wake stub when coming out of deep sleep. This function runs immediately as soon as the chip wakes up – before any normal initialization, bootloader code has run. After the wake stub runs, the chip can go back to sleep or continue to start normally.

Unlike the other sleep modes, the system cannot go into Deep-sleep mode automatically. `esp_deep_sleep_start()` function can be used to immediately enter deep sleep once wake-up sources are configured.
By default, ESP32 will automatically power down the peripherals not needed by the wake-up source. But you can optionally decide what all peripherals to shut down/keep on. For more information, check out API docs.

ESP32 Deep Sleep Wake-up sources
Wake up from deep sleep mode can be done using several sources. These sources are:

- Timer
- Touch pad
- External wakeup(ext0 & ext1)

Wake-up sources can be combined, in this case the chip will wake up when any one of the sources is triggered.
These sources can be configured at any moment before entering in to sleep mode.

Warning:
It is also possible to go into deep sleep with no wake-up sources configured, in this case the chip will be in deep sleep mode indefinitely, until external reset is applied.

ESP32 Wake-up Source : Timer
RTC controller has a built in timer which can be used to wake up the chip after a predefined amount of time.
Time is specified at microsecond precision, but the actual resolution depends on the clock source selected.
`esp_sleep_enable_timer_wakeup()` function can be used to enable deep sleep wake up using a timer.
Here’s a code that demonstrates the most basic deep sleep example with a timer as wake-up source and how to store data in RTC memory to use it over reboots.

```
#define uS_TO_S_FACTOR 1000000  //Conversion factor for micro seconds to seconds
#define TIME_TO_SLEEP  5        //Time ESP32 will go to sleep (in seconds)

RTC_DATA_ATTR int bootCount = 0;

void setup(){
        Serial.begin(115200);
        delay(1000); //Take some time to open up the Serial Monitor

        //Increment boot number and print it every reboot
        ++bootCount;
        Serial.println("Boot number: " + String(bootCount));

        //Print the wakeup reason for ESP32
        print_wakeup_reason();

        //Set timer to 5 seconds
        esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
        Serial.println("Setup ESP32 to sleep for every " + String(TIME_TO_SLEEP) +
        " Seconds");

        //Go to sleep now
        esp_deep_sleep_start();
}

void loop(){}

//Function that prints the reason by which ESP32 has been awaken from sleep
void print_wakeup_reason(){
        esp_sleep_wakeup_cause_t wakeup_reason;
```

```
        wakeup_reason = esp_sleep_get_wakeup_cause();
        switch(wakeup_reason)
        {
                case 1  : Serial.println("Wakeup caused by external signal using RTC_IO"); break;
                case 2  : Serial.println("Wakeup caused by external signal using RTC_CNTL");
break;
                case 3  : Serial.println("Wakeup caused by timer"); break;
                case 4  : Serial.println("Wakeup caused by touchpad"); break;
                case 5  : Serial.println("Wakeup caused by ULP program"); break;
                default : Serial.println("Wakeup was not caused by deep sleep"); break;
        }
}
```

ESP32 Wake-up Source : Touch Pad

RTC IO module contains logic to trigger wake up when a touch sensor interrupt occurs. You need to configure the touch pad interrupt before the chip starts deep sleep. `esp_sleep_enable_touchpad_wakeup()` function can be used to enable this wake-up source. Here's a code that demonstrates the most basic deep sleep example with a touch as wake-up source and how to store data in RTC memory to use it over reboots.

```
//Define touch sensitivity. Greater the value, more the sensitivity.
#define Threshold 40

RTC_DATA_ATTR int bootCount = 0;
touch_pad_t touchPin;

void callback(){
  //placeholder callback function
}

void setup(){
  Serial.begin(115200);
  delay(1000);

  //Increment boot number and print it every reboot
  ++bootCount;
  Serial.println("Boot number: " + String(bootCount));

  //Print the wakeup reason for ESP32 and touchpad too
  print_wakeup_reason();
  print_wakeup_touchpad();

  //Setup interrupt on Touch Pad 3 (GPIO15)
  touchAttachInterrupt(T3, callback, Threshold);

  //Configure Touchpad as wakeup source
  esp_sleep_enable_touchpad_wakeup();

  //Go to sleep now
  esp_deep_sleep_start();
}

void loop(){}

//Function that prints the reason by which ESP32 has been awaken from sleep
void print_wakeup_reason(){
  esp_sleep_wakeup_cause_t wakeup_reason;
  wakeup_reason = esp_sleep_get_wakeup_cause();
  switch(wakeup_reason)
  {
    case 1  : Serial.println("Wakeup caused by external signal using RTC_IO"); break;
    case 2  : Serial.println("Wakeup caused by external signal using RTC_CNTL"); break;
    case 3  : Serial.println("Wakeup caused by timer"); break;
    case 4  : Serial.println("Wakeup caused by touchpad"); break;
    case 5  : Serial.println("Wakeup caused by ULP program"); break;
    default : Serial.println("Wakeup was not caused by deep sleep"); break;
  }
}


//Function that prints the touchpad by which ESP32 has been awaken from sleep
void print_wakeup_touchpad(){
  touch_pad_t pin;
  touchPin = esp_sleep_get_touchpad_wakeup_status();
  switch(touchPin)
  {
    case 0  : Serial.println("Touch detected on GPIO 4"); break;
    case 1  : Serial.println("Touch detected on GPIO 0"); break;
```

```
      case 2  : Serial.println("Touch detected on GPIO 2"); break;
      case 3  : Serial.println("Touch detected on GPIO 15"); break;
      case 4  : Serial.println("Touch detected on GPIO 13"); break;
      case 5  : Serial.println("Touch detected on GPIO 12"); break;
      case 6  : Serial.println("Touch detected on GPIO 14"); break;
      case 7  : Serial.println("Touch detected on GPIO 27"); break;
      case 8  : Serial.println("Touch detected on GPIO 33"); break;
      case 9  : Serial.println("Touch detected on GPIO 32"); break;
      default : Serial.println("Wakeup not by touchpad"); break;
    }
  }
```

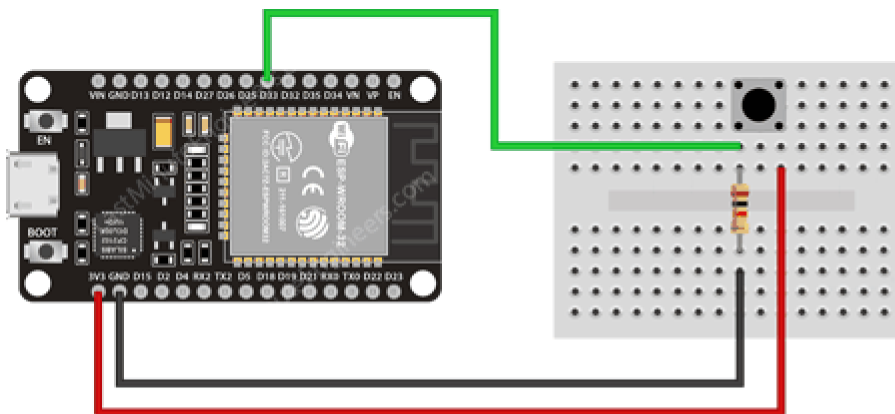
ESP32 Wake-up Source : External Wake-up

There are two types of external triggers to wake ESP32 up from deep sleep.

- ext0 – Use it when you want to wake-up the chip by one particular pin only.
- ext1 – Use it when you have several buttons for the wake-up.

ext0 External Wake-up Source

RTC controller contains logic to trigger wake-up when one particular pin is set to a predefined logic level. That pin can be one of RTC GPIOs 0,2,4,12-15,25-27,32-39.
`esp_sleep_enable_ext0_wakeup(GPIO_PIN,LOGIC_LEVEL)` function can be used to enable this wake-up source. The function takes two parameters. First one is a pin number to which button is connected and second one decides if we want to trigger the wake up by a LOW or a HIGH state of the pin.
ext0 uses RTC IO to wake up, so RTC peripherals will be kept powered ON during deep sleep if this wake-up source is requested.
Because RTC IO module is enabled in this mode, internal pullup or pulldown resistors can also be used. They need to be configured by the application using `rtc_gpio_pullup_en()` and `rtc_gpio_pulldown_en()` functions, before calling `esp_sleep_start()`.
Below schematic shows how to connect a push button to ESP32 GPIO that serves as a ext0 external wake-up source.



Here's a code that demonstrates the most basic deep sleep example with a ext0 as wake-up source.

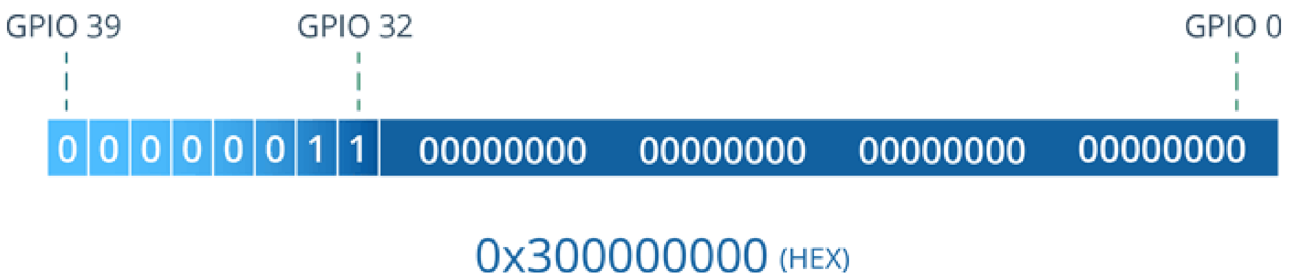
```
RTC_DATA_ATTR int bootCount = 0;

void setup(){
  Serial.begin(115200);
  delay(1000);

  //Increment boot number and print it every reboot
  ++bootCount;
  Serial.println("Boot number: " + String(bootCount));

  //Print the wakeup reason for ESP32
```

```
  print_wakeup_reason();

  //Configure GPIO33 as ext0 wake up source for HIGH logic level
  esp_sleep_enable_ext0_wakeup(GPIO_NUM_33,1);

  //Go to sleep now
  esp_deep_sleep_start();
}

void loop(){}

//Function that prints the reason by which ESP32 has been awaken from sleep
void print_wakeup_reason(){
  esp_sleep_wakeup_cause_t wakeup_reason;
  wakeup_reason = esp_sleep_get_wakeup_cause();
  switch(wakeup_reason)
  {
    case 1  : Serial.println("Wakeup caused by external signal using RTC_IO"); break;
    case 2  : Serial.println("Wakeup caused by external signal using RTC_CNTL"); break;
    case 3  : Serial.println("Wakeup caused by timer"); break;
    case 4  : Serial.println("Wakeup caused by touchpad"); break;
    case 5  : Serial.println("Wakeup caused by ULP program"); break;
    default : Serial.println("Wakeup was not caused by deep sleep"); break;
  }
}
```

ext1 External Wake-up Source
ESP32 can be woken up from deep sleep using multiple GPIO pins. Those pins can be one of RTC GPIOs 32-39.
As ext1 wake-up source uses RTC controller, it does't need RTC peripherals and RTC memories to be powered ON. In this case internal pullup and pulldown resistors will not be available.
To use internal pullup or pulldown resistors, we need to request RTC peripherals to be kept on during sleep, and configure pullup/pulldown resistors using rtc_gpio_ functions, before entering sleep.
`esp_sleep_enable_ext1_wakeup(BUTTON_PIN_MASK,LOGIC_LEVEL)` function can be used to enable this wake-up source. The function takes two parameters. First one is a pin mask to let ESP32 know which all pins we are going to use
The second parameter can be one of the two logic functions can be used to trigger wake-up:

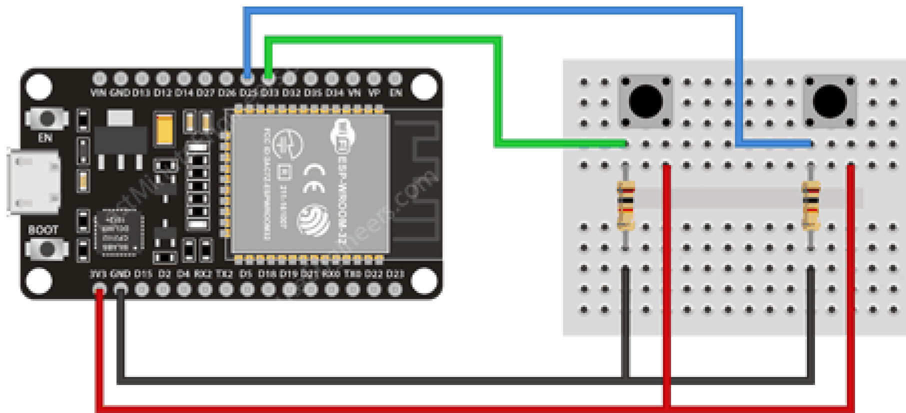
- Wake up if any of the selected pins is HIGH (`ESP_EXT1_WAKEUP_ANY_HIGH`)
- Wake up if all the selected pins are LOW (`ESP_EXT1_WAKEUP_ALL_LOW`)

The easiest way to understand this pin masking technique is to write it in a binary format.



- 0 represents masked pins
- 1 represents pins that will be enabled as a wake-up source

The bit numbering is so simple and based on a normal GPIO numbering. The least significant bit(LSB) represents GPIO0 and the most significant bit(MSB) represents GPIO39.
As the the first pin available is GPIO32, the mask contains 32 times 0 on the right. And then for each enabled pin we write a 1.
If you don't want to enable any GPIO for wake up you have to write a 0 at its corresponding place.



Once you are done, you need to convert it into HEX before using it as a parameter.
Below schematic shows how to connect multiple push buttons to ESP32 GPIOs that serves as a ext1 external wake-up source.

Here's a code that demonstrates the most basic deep sleep example with a ext1 as wake-up source.

```
//Pushbuttons connected to GPIO32 & GPIO33
#define BUTTON_PIN_BITMASK 0x300000000

RTC_DATA_ATTR int bootCount = 0;

void setup(){
  Serial.begin(115200);
  delay(1000);

  //Increment boot number and print it every reboot
  ++bootCount;
  Serial.println("Boot number: " + String(bootCount));

  //Print the wakeup reason for ESP32
  print_wakeup_reason();

  //Configure GPIO32 & GPIO33 as ext1 wake up source for HIGH logic level
  esp_sleep_enable_ext1_wakeup(BUTTON_PIN_BITMASK,ESP_EXT1_WAKEUP_ANY_HIGH);

  //Go to sleep now
  esp_deep_sleep_start();
}

void loop(){}

//Function that prints the reason by which ESP32 has been awaken from sleep
void print_wakeup_reason()
{
  esp_sleep_wakeup_cause_t wakeup_reason;
  wakeup_reason = esp_sleep_get_wakeup_cause();
  switch(wakeup_reason)
  {
    case 1  : Serial.println("Wakeup caused by external signal using RTC_IO"); break;
    case 2  : Serial.println("Wakeup caused by external signal using RTC_CNTL"); break;
    case 3  : Serial.println("Wakeup caused by timer"); break;
    case 4  : Serial.println("Wakeup caused by touchpad"); break;
    case 5  : Serial.println("Wakeup caused by ULP program"); break;
    default : Serial.println("Wakeup was not caused by deep sleep"); break;
  }
}
```

```
#include <Arduino.h>
/*
Wakeup ESP32 with Touch Pad
====================================
http://esp-idf.readthedocs.io/en/latest/api-reference/system/deep_sleep.html
code inspired by Pranav Cherukupalli <cherukupallip@gmail.com>
*/

#define Threshold 40    /* Greater the value, more the sensitivity */

// Display touchpad origin
void print_wakeup_touchpad(){
  touch_pad_t touchPin;
  touchPin = esp_sleep_get_touchpad_wakeup_status();

  switch(touchPin)
  {
    case 0  : Serial.println("Touch detected on GPIO 4"); break;
    case 1  : Serial.println("Touch detected on GPIO 0"); break;
    case 2  : Serial.println("Touch detected on GPIO 2"); break;
    case 3  : Serial.println("Touch detected on GPIO 15"); break;
    case 4  : Serial.println("Touch detected on GPIO 13"); break;
    case 5  : Serial.println("Touch detected on GPIO 12"); break;
    case 6  : Serial.println("Touch detected on GPIO 14"); break;
    case 7  : Serial.println("Touch detected on GPIO 27"); break;
    case 8  : Serial.println("Touch detected on GPIO 33"); break;
    case 9  : Serial.println("Touch detected on GPIO 32"); break;
    default : Serial.println("Wakeup not by touchpad"); break;
  }
}

// Display wakeup origine
void print_wakeup_reason(){
  esp_sleep_wakeup_cause_t wakeup_reason;

  wakeup_reason = esp_sleep_get_wakeup_cause();

  switch(wakeup_reason)
  {
    case ESP_SLEEP_WAKEUP_EXT0 : Serial.println("Wakeup caused by external signal using RTC_IO");
break;
    case ESP_SLEEP_WAKEUP_EXT1 : Serial.println("Wakeup caused by external signal using
RTC_CNTL"); break;
    case ESP_SLEEP_WAKEUP_TIMER : Serial.println("Wakeup caused by timer"); break;
    case ESP_SLEEP_WAKEUP_TOUCHPAD : Serial.println("Wakeup caused by touchpad"); break;
    case ESP_SLEEP_WAKEUP_ULP : Serial.println("Wakeup caused by ULP program"); break;
    case ESP_SLEEP_WAKEUP_GPIO : Serial.println("Wakeup caused by GPIO"); break;
    case ESP_SLEEP_WAKEUP_UART : Serial.println("Wakeup caused by UART"); break;
    default : Serial.printf("Wakeup was not caused by deep sleep: %d\n",wakeup_reason); break;
  }
}

// Execute this function when Touch Pad in pressed
void callback() {
  Serial.println("Do something when Touch Pad is pressed");
}

void setup(){
  Serial.begin(115200);

  print_wakeup_reason();

  //Print the wakeup reason for ESP32 and touchpad too
  print_wakeup_reason();
  print_wakeup_touchpad();

  //Setup interrupt on Touch Pad 6 (GPIO14)
  touchAttachInterrupt(T6, callback, Threshold);

  //Configure Touchpad as wakeup source
  esp_sleep_enable_touchpad_wakeup();

  //esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
  Serial.println("Setup ESP32 to sleep until Touch Pad will be pressed");

  Serial.println("Going to sleep now");
  delay(1000);
  Serial.flush();
  esp_deep_sleep_start();
  Serial.println("This message will never be printed");
```

```
}

void loop(){
}
```

Saving battery life tips

Use the right battery & board
Want to increase battery life? Then pick an ESP32 board that has a built-in battery connector. These boards likely use a much more efficient LDO voltage regulator.

Use the right battery
The ESP32 needs an input voltage of around 3.3V, so pick a battery that delivers a voltage close to this. I tested some LIPO batteries with a voltage of 3.7V, which seems to do the trick just fine.

Pick the right ESP32 board (single-core)
Most IoT devices are relatively simple and don't require a lot of computing power. Yet the ESP32 has a dual-core processor. Consider buying a board that uses the single-core version of the ESP32 (ESP32-SOLO-1). The difference between dual-core and single-core? Well, a regular ESP32 will consume between 27-44mA when running at 160MHz while the single-core consumes about 30% less, coming in at 27mA-34mA.

The CPU is powered on.	240 MHz *	Dual-core chip(s)	30 mA ~ 68 mA
		Single-core chip(s)	N/A
	160 MHz *	Dual-core chip(s)	27 mA ~ 44 mA
		Single-core chip(s)	27 mA ~ 34 mA
	Normal speed: 80 MHz	Dual-core chip(s)	20 mA ~ 31 mA
		Single-core chip(s)	20 mA ~ 25 mA

Reduce the clock speed
Fewer cores consume less power. The same thing can be said for slower cores. If you can get away with a single-core ESP32, chances are you can get away with running that core at lower clock speeds. Reducing the default clock speed from 160MHz to 80MHz can drop the energy consumption another 20%! In Arduino it's just a one-liner:

```
setCpuFrequencyMhz(80);
```

Turn off everything in deep sleep
When running an ESP32 on a battery, you'll want to keep it in deep sleep for as long as possible. In case you don't know how: you configure a wakeup timer and then start the deep sleep mode:
// How many minutes the ESP should sleep
#define DEEP_SLEEP_TIME 15

```
// Configure the timer to wake us up!
esp_sleep_enable_timer_wakeup(DEEP_SLEEP_TIME * 60L * 1000000L);

// Go to sleep! Zzzz
esp_deep_sleep_start();
```

However, I found that this is not sufficient to keep energy consumption to a minimum. I've seen various bug reports claiming that this doesn't always turn off the WiFi or Bluetooth radio before going to deep sleep.
So instead of calling `esp_deep_sleep_start()` directly, I do some preparations first. I disconnect the WiFi, turn off the WiFi and Bluetooth radio, turn off the ADC and then turn off the WiFi and Bluetooth radios again, but with the ESP API:

```
#include "WiFi.h"
#include "driver/adc.h"
#include <esp_wifi.h>
#include <esp_bt.h>

// How many minutes the ESP should sleep
#define DEEP_SLEEP_TIME 15

void goToDeepSleep()
{
  Serial.println("Going to sleep...");
  WiFi.disconnect(true);
  WiFi.mode(WIFI_OFF);
  btStop();

  adc_power_off();
  esp_wifi_stop();
  esp_bt_controller_disable();

  // Configure the timer to wake us up!
  esp_sleep_enable_timer_wakeup(DEEP_SLEEP_TIME * 60L * 1000000L);

  // Go to sleep! Zzzz
  esp_deep_sleep_start();
}
```

Note: If you're using the built-in ADC, don't forget to turn it back on before using it:

```
adc_power_on();
```

WiFi or Bluetooth doesn't have to be explicitly turned on. The regular is enough.

```
WiFi.begin(NETWORK, PASSWORD);
```

Don't forget to put your peripherals into sleep mode when you're not using them. Many sensors have built-in power-saving features that you can trigger. So definitely check out the libraries you use to interface with them and check if they have this.

Add a WiFi connection timeout

The ESP32 is so great for IoT projects because it has built-in WiFi. No need for proprietary wireless signals and protocols. The only downside is that WiFi is pretty power-hungry, so you want to minimize the time spent with the radio on. I always add a timeout for setting up a WiFi connection. You don’t want your ESP32 to keep looking for a particular WiFi network endlessly. It’ll drain the battery if your network is down for just a couple of minutes (modem restart, power outage, …) or of its temporarily out of range.

I usually implement a 10-second timeout. If no WiFi connection can be established in this time frame, the ESP32 goes back to deep sleep (hoping WiFi is available when it wakes back up):

```
#define WIFI_NETWORK “YOUR_WIFI_NETWORK_NAME”
#define WIFI_PASSWORD “YOUR_WIFI_PASSWORD”
#define WIFI_TIMEOUT 10000 // 10seconds in milliseconds

void connectToWiFi()
{
  Serial.print("Connecting to WiFi... ");
  WiFi.mode(WIFI_STA);
  WiFi.begin(WIFI_NETWORK, WIFI_PASSWORD);

        // Keep track of when we started our attempt to get a WiFi connection
  unsigned long startAttemptTime = millis();

  // Keep looping while we're not connected AND haven't reached the timeout
  while (WiFi.status() != WL_CONNECTED &&
          millis() - startAttemptTime < WIFI_TIMEOUT){
    delay(10)
  }

  // Make sure that we're actually connected, otherwise go to deep sleep
  if(WiFi.status() != WL_CONNECTED){
    Serial.println("FAILED");
    goToDeepSleep();
  }

  Serial.println("OK");
}
```

Use RTC memory to reduce WiFi connections
Finally, the biggest tip of them all: WiFi connections are super rough on battery life, so making fewer will improve battery life dramatically.
Let's say you want to build a temperature sensor that takes a new measurement every 15 minutes. Do you really need to know the temperature in real-time? Or is it okay to send a batch of readings every 6 hours, for instance?
If so, consider using the RTC memory of the ESP to store your measurements without connecting to WiFi. Many of my sensors employ the following algorithm:

- Wake up from deep sleep
- Measure something
- Store the measurement in the RTC memory
- If we have X readings, connect to WiFi and send them all to the cloud
- If not, go back to deep sleep

Translated into Arduino code, that would give you:

```
// How many readings we want to store before sending them over WiFi
// -> 20 readings with 15min between each reading = only connecting to WiFi
every 300 minutes (5hours) instead of every 15min
#define MAX_OFFLINE_READINGS 20

// Place in the RTC memory to store the offline readings (and how many we
have so far)
RTC_DATA_ATTR unsigned char offlineReadingCount = 0;
RTC_DATA_ATTR unsigned int readings_temp[MAX_OFFLINE_READINGS + 1];

// Write the temperature (20.9°C) to the RTC memory. You might want to
replace this with your actual measurement code ;)
readings_temp[offlineReadingCount] = 209;
offlineReadingCount++;

// If we collected less then the maximum amount of readings, go back to deep
sleep!
if(offlineReadingCount <= MAX_OFFLINE_READINGS){
    goToDeepSleep();
    return;
}

// If we get here we have to send the readings over WiFi
sendReadings();

// Don't forget to reset the counters!
OfflineReadingCount = 0;
```

Note: this code doesn't handle a situation in which the ESP32 can't connect to WiFi when the offline readings buffer is full. You might want to add this.

Use static IP & avoid domain names
Use a static IP address and avoid using hostnames (which have to be resolved with DNS).
By hard coding an IP address into your ESP32, you reduce the time it takes to get an address through DHCP.
Avoid using domain names when connecting to remote servers. Creating an MQTT connection with cloudmqtt.com, for instance, requires your ESP32 to figure out the IP address behind the name. This happens through a DNS resolver, and this can take anywhere between a few milliseconds and several hundred. If you can, use the IP address of servers you want to connect to. This will allow you to turn off the WiFi sooner.
Although these tips will reduce the amount of time spent with the WiFi radio on, they aren't always practical. A static IP address is doable on your home network, but not when you want to ship devices to customers. Likewise, avoiding DNS is likely not possible if you use a service like AWS IoT.