C++ Arduino camera IoT ESP32

A DIY Slow-Motion Camera

deangi

10 Jan 2024 CPOL 0

The inexpensive ESP32 platform is used to take periodic photos that can make a slow motion video.

Introduction

The ESP32 platform provides a very capable platform for implementing an IOT (internet of things) application for sensing data and controlling items in the real world In this application we use a camera to take periodic photos and store them on an SD card or to send them via WiFi to some remote server.

If desired these images can be remotely manipulated to recognize events, or combine them into a video (for example searching for "create video from images" reveals many examples in python and OpenCV).

Background

The application is written to configure itself on startup by reading a configuration file which can be stored on the ESP32 flash file system, or on an SD card. Once the configuration information is obtained the ESP32 will enter a deep sleep mode, waking occasionally to take a photo and either store it locally or upload it to a remote server via FTP over WiFi.

The configuration file contains information on the following:

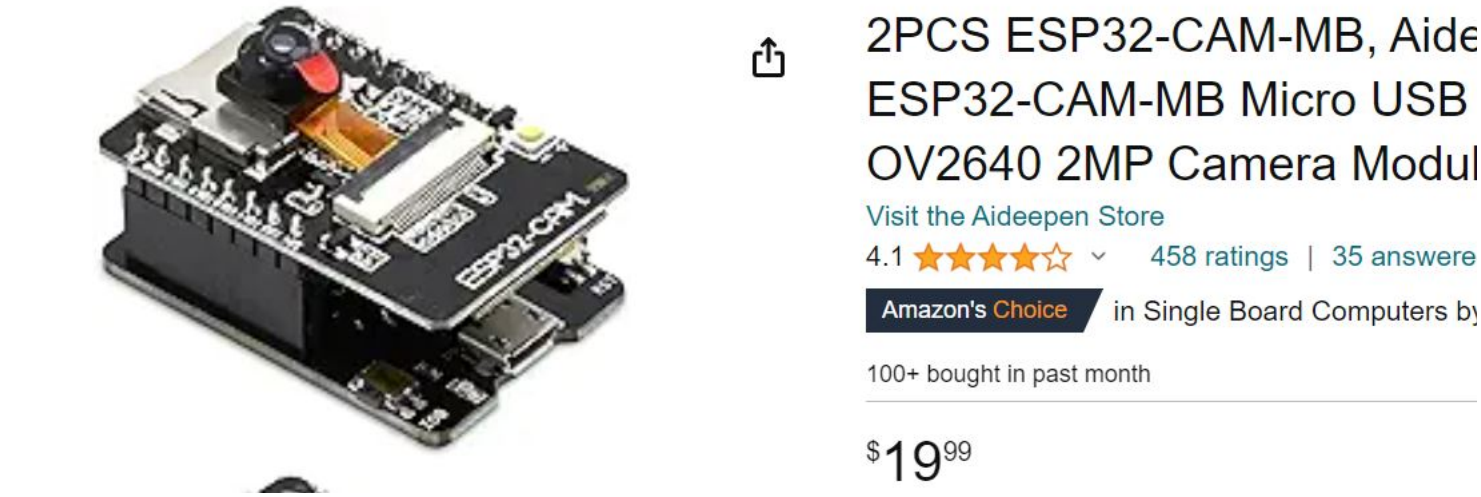
- Information on how to connect to the WiFi (SSID and password)
- Information on how to connect to the FTP host (if desired) (host name, user name, password, and FTP folder)
- Information on how often to take a photo - how many photos per day to take
- Information on whether to store the photos locally on an SD card or to FTP them to some server

One driving issue for architecting the application was to support a battery mode of operation. To keep the power consumption low, the ESP32 system is put into a deep sleep mode most of the time, only waking up occasionally when it's time to take a photograph.

Another consideration was supporting a mode where there is no available WiFi (sort of a trail camera mode) where photographs are stored on a local SD card. In this mode the device could potentially operate for days or weeks at a time, taking just a few photographs per day. For example this could be placed in some remote location in a forest or even under water in a suitable container to observe some slowly changing phenomena.

Hardware Architecture

The hardware is based on the ESP32 platform which is quite compact and very inexpensive. Development boards including a camera module are available on Amazon for around $20 USD for two boards. These boards include both the camera and an SD card slot to store photos. Searching Amazon for ESP32 Camera should turn these right up. There are multiple other ESP32 camera modules available from different sources.



There are no hardware additions or modifications to the board needed for this application - just program the code into the module and make sure there is a valid configuration file in flash to read on startup. Install an SD card to store images if desired.

Software Architecture

Since this application uses the "deep sleep" mode of the ESP32, it's a little different than a normal ESP32 application in one way. As you know these applications classically have a setup() function which is run one time on boot-up, and a loop() function which is run continuously after boot up.

In a deep-sleep application, no code is placed in the loop() function as the ESP32 will be in deep sleep mode when there's no photo to be taken. So the loop() function is empty.

However, all the action takes place in setup() function. This function is executed on a first bootup and is also executed every time the ESP32 wakes up from deep sleep mode.

Deep Sleep Mode

The ESP32 platform includes a number of "modes" which utilize varying degrees of power when they are operational. Normal or Active mode operation means the ESP32 is running it's main CPU continuously. This is the mode which utilizes the most power because most of the chip is operational - the CPU, the memory, the flash, timers, GPIO/ports, and even perhaps the radio (WiFi or Bluetooth).

To conserve power, there are some lower-power modes that can be employed - basically this means that parts of the chip are powered down and some functions and features can not be used.

In Active mode, some power savings can be achieved by turning off the radio when it's features are not needed. Additionally the CPU clock can be "turned down" or run at a slower rate. This causes the chip to execute slower, but reduces the power used in active mode.

In addition to active mode, there are like five different power-saving modes - which is too many to describe here. The chip manufacturer has a good description of these modes here.

For this application we've chosen to use the mode with (almost) the lowest power consumption - around 10 microamps - called deep sleep mode. In this mode almost all the chip's functions are depowered. The only parts still active are:

- RTC controller
- ULP coprocessor
- RTC FAST memory
- RTC SLOW memory

This means the CPU, the RAM, almost all the peripherals are depowered. Power consumption is reduced to the micro-watt level in this deep sleep mode.

The only things "awake" during this mode are some small amount of special RAM memory and the bits of circuitry to keep track of what time it is and how long the chip has been in deep sleep - and also a GPIO for external signaling.

To recover from this deep sleep mode, we need to "wake up" the CPU - which means going back into active mode. However, in deep sleep mode the CPU and memory system are powered down - so no instructions are executed and no RAM memory contents are preserved. Of course flash memory where the program itself is stored is preserved, although it is powered down during deep sleep mode.

"Wake up" means the process of basically rebooting the CPU from a completely powered down mode. During this boot-up process, the code needs to know if it is doing a boot up because of what is called a cold start - meaning the system has just been powered on and is booting up for the first time, or if

the boot up is due to a wake-up event from a low power mode such as deep sleep.

Fortunately the manufacturer has provided special IO registers that can be read to determine what type of boot up is happening. Then the code can handle the boot process differently based on what it has read from this boot-up reason register.

Below you'll see the code at the beginning of the setup() function that executes on every boot or wake up cycle. There's a call to a function esp_sleep_get_wakeup_cause() to get a code that describes the type of wake-up that is being done. Later on in the setup function we will use this code to determine what type of wake up processing to do.

C++

```
  WRITE_PERI_REG(RTC_CNTL_BROWN_OUT_REG, 0); //disable brownout detector

  //----------------------------------------------------------------------
  // Start out with things we want to do every time we wake up or boot up
  //----------------------------------------------------------------------


  // let's see the reason that the ESP32 woke up
  esp_sleep_wakeup_cause_t wakeup_reason; // to see why we booted (either power on, or wake up)
  wakeup_reason = esp_sleep_get_wakeup_cause();


#ifdef WANTSERIAL
  Serial.begin(115200);
  print(SIGNON);
#endif
```

Cold Boot Processing

For a cold boot situation, some different operations are needed for this architecture. In particular we need to read the configuration file and store the values in some special memory that survives deep sleep mode. Most memory contents are lost when we awaken from deep-sleep as the RAM memory system is turned off to save power. However the chip set has provided a small amount (8192 bytes I think) of memory that retains it's contents during a deep sleep event.

So the first item of business to do during a cold boot is to read the needed values from the configuration file and store them in this deep sleep protected memory. This is not totally necessary as the configuration file is stored in flash memory and could be read each time we awaken, but this would cause the CPU to have to execute potentially millions of instructions each time we awake from deep sleep. So to save power, the code will read the configuration file one time at cold boot and store the needed values in this protected memory so they can easily be accessed when we wake up from deep sleep.

We also try to determine the correct time-of-day during the cold boot sequence. This means we need to connect to the internet and query a network-time-protocol (NTP) server to get the current time and date. This data is then programmed into the real-time clock (RTC) on the chip - fortunately the RTC is maintained during deep-sleep mode so we really only need to do this one time on cold boot.

The idea here is that for an "off grid" application (no WiFi available) we can cold boot once in an area where WiFi is available and then the system will know the time, even if it is moved out of range of the WiFi.

This is the code to execute the cold-boot tasks - you'll see it is reading keys from the configuration file and it is also connecting to WiFi to try to read the current time/date via NTP.

C++ Shrink ▲

```
  //------------------------------------------------------------
  // Boot up tasks - Stuff to do on an initial boot from power
  // on or hard reset
  //------------------------------------------------------------
  if ((bootCount == 0) || (wakeup_reason != ESP_SLEEP_WAKEUP_TIMER))
  {
    // read needed data from the config file - stored in RTC memory so it survives deep sleep
    readKey(CONFIGFN,"SSID=",ssid,127);
    readKey(CONFIGFN,"PASSWORD=",password,127);
    readKey(CONFIGFN,"SERVER=",uploadHost,127);
    readKey(CONFIGFN,"FTPUSER=",ftpUser,127);
    readKey(CONFIGFN,"FTPPASSWORD=",ftpPswd,127);
    readKey(CONFIGFN,"FTPFOLDER=",ftpFolder,127);
    readKey(CONFIGFN,"PHOTOSPERDAY=",buf,30);
    delayBetweenPhotosMs=24*60*60*1000/atol(buf);
    if (delayBetweenPhotosMs < 60*1000)
       delayBetweenPhotosMs = 60*1000;
    sprintf(buf,"Delay=%d",delayBetweenPhotosMs);
    print(buf);

    // initialize the RTC and read network time if possible
```

```
  rtc.setTime(0,0,0,1,1,2023); // default time 00:00:00 1/1/2023
  if (connectToWiFi() == 1)
  {
    needNtp = true;
    getNtpTime(); // get NTP time if possible
  }
  turnOffWiFi(); // now we can turn off the WiFi modem to save power
}
```

Wake Up Processing

When awaking from a deep sleep mode, we don't need to read the configuration file because the data needed from that have been stored in deep sleep protected memory (also called RTC memory). What want to do in this step is to take a photo and then either store it on the SD card, or to upload it to the requested FTP server for storage.

The camera is connected to the ESP32 via a multi-pin connection - many IO pins of the processor are used in this process. For different ESP32 development boards the actual pins used may vary - so although the code for setting up the camera, taking the picture, and retrieving the picture may be the same for the same camera, the actual connections to the ESP32 chip may vary. The code in this example is set up for the Aideepen camera connection implementation, if you use a different type of board, you may need to make some changes to the code to account for differences in how the camera is connected.

You'll see in the code segment below that if the wakeup_reason code is deep sleep wakeup we will setup the camera and take a photo (setupMeasurement() and makeMeasurement() in the code below.

C++

```
//------------------------------------------------------------
// *** wakeup from deep sleep tasks ***
//------------------------------------------------------------
if (wakeup_reason == ESP_SLEEP_WAKEUP_TIMER)
{

  // do wakeup tasks here (IoT Loop)
  setupMeasurement();
  makeMeasurement();
}
```

Going Into Deep Sleep Mode

There are some IO registers that can take the ESP32 back into deep sleep mode. There are several ways to awaken from deep sleep mode, these are called wake up triggers. For example the ESP32 can be configured to awaken after a certain amount of time, or by the application of an external signal to one of the IO pins, or someone touching a screen.

For this application we want to sleep until it's time to take the next picture, so we'll program the ESP32 to sleep for a certain period of time. The period of time is the time between photos minus the time it took the ESP32 to wake up and take the last photo. Fortunately the ESP32 provides a way to see how many milliseconds have elapsed since the last boot up or wake up. So we can take the desired photo interval (like one hour or something) and subtract the time the ESP32 has been active since it last woke up. This amount of time is computed in microseconds and is then programmed into the RTC before going into deep sleep mode, so that the ESP32 will awake when it's next time to take a photograph. When the code finally writes the registers to go into deep sleep mode, the CPU will stop and will not execute any other code until the wake up event happens.

In the code segment below you'll see that we count the boot up (the code keeps a count of how many times it's booted / woken up). Turn off the LED to provide an indication that we're going back to sleep.

Then we calculate the number of milliseconds to sleep until it's time to take the next photo and tell the ESP32 to go into deep sleep mode (esp_deep_sleep_start()). The ESP32 processor will never return from this call because it's gone into deep sleep mode. When it is awakened it will essentially reboot and the setup() function will be executed again. So this is the end of the execution of the application for this cycle.

For the first sleep period after a cold boot, the first sleep time is adjusted so that the ESP32 will wake up on an even interval with the number of photos per day requested. For example if the configuration file requests 24 photos per day, that is one per hour. The first sleep period will be adjusted so that the ESP will wake up and take it's first photo right at the top of the hour.

C++

```
//----------------------------------
// *** and go back to sleep here
//----------------------------------

long milliseconds = millis(); // how long have we been awake?
sprintf(buf,"Awake for %d ms",milliseconds);
print(buf);

long tts = (delayBetweenPhotosMs - milliseconds) * ms_TO_S_FACTOR;
esp_sleep_enable_timer_wakeup(tts); // how long to sleep
esp_deep_sleep_start(); // good night!  Have a nice nap.
```

```
// and, the ESP32 never executes any code past the deep_sleep_start!
// on next wakeup or reset, the startup() function will be called again.
```

Miscellaneous Support Code

The following libraries are used in the code as it processes the application features

- NTP library - network time protocol library to query the current time and date from the network
- WIFI library - to connect to a WiFi access point as a client, get an IP address, and access the network
- FTP library - used to send a photograph to a remote server over the WiFi network connection
- Camera library - used to configure the camera and capture a photograph.
- RTC and time/date - real time clock support to set/get the time and date and to keep the time and date current as time passes
- File System - access files on the SD card, read the configuration file, write picture files

Source code can be found in github here.

Future Updates

In the future I would like to turn this into a "trail cam" - where a sensor would detect the presence of an animal (human or not) and wake up to take a picture. There are several methods for detecting creatures - passive infrared sensors, radar sensors, lidar sensors.

Additionally we could easily add code to sense other quantities such as light level, temperature, pressure, humidity - these could be logged if they were of interest to the users application.

Another idea would be to add some key to the configuration file to specify "excluded hours" - for example if you're photographing a daytime phenomena, it's probably not useful to take photos during the night time hours.

Blocks of code should be set as style "Formatted" like this:

C++

```
//
// Any source code blocks look like this
//
```

History

Version 1.0, January 5, 2024, Initial version

License

Written By
deangi
Team Leader
United States

Just a tinkering retired engineer interested in Internet-Of-Things devices, building them, programming them, operating them.

Comments and Discussions