C++ web Arduino FTP IoT

IoT Data Hub for remote edge deployment

deangi
9 Feb 2024 CPOL 0

IoT data edge hub is designed for remote low power unattended applications. IoT devices can post data to the Hub which will store the data and make it available for later download to core servers. Design is based on a low cost ESP32 platform with an attached SD card.

Introduction

In this application an inexpensive ESP32 platform is used to store data on an SD card forming an 'edge hub'. Stored data can be transferred via direct swap of the SD card, or can be periodically downloaded via an FTP connection. The design criteria included remote deployment in a potentially 'off grid' area where constant internet connection is not available and where power availability is severely limited.

Background

IoT sensors are often deployed in a very low power situation - and may be placed in very remote locations. Gathering this data and storing it temporarily at the "edge" is often desirable.

This application allows gathering of data and storing it on an SD card on a tiny edge based hub. The hub can connect to an existing WiFi signal, or it can host it's own WiFi access point - allowing IoT devices to momentarily connect and push data to the IoT edge hub.

Server devices may gather this data at a later time by connecting to the hub and using FTP to pull the data for later use. The IoT Hub uses less than one watt of power so is suitable for remote operation on battery and/or solar power.

One example application allows the IoT hub to be positioned on a mountaintop without internet connections and severe power limitations. Perhaps it is taking periodic measurements of weather parameters and/or photographs. At a later date a truck or even a drone may visit the site and upload the collected data from the IoT hub.

Another application might be subterranean placement, or even a floating buoy placement where data can be gathered and stored at the edge for eventual retrieval.

Design Architecture

The ESP32 platform was chosen for this application based on good ratings for low cost, low power consumption, and high availability. The ESP32 has an attached SD card reader that can be used to provide storage for data uploaded from sensors. At periodic intervals this data can be retrieved and transferred to a non-edge server.

In this design the ESP32 WiFi interface is used to either connect to an existing WiFi network, or to provide an access point hosting it's own WiFi service. The WiFi interface supports two services for sensors to use to push data to be stored. First - an FTP server is implemented to allow remote sensors with higher data collection volumes to upload data - for example a remote camera may push photographs at periodic intervals. An HTTP service is also implemented to allow lower rate sensors to push data to the edge server.

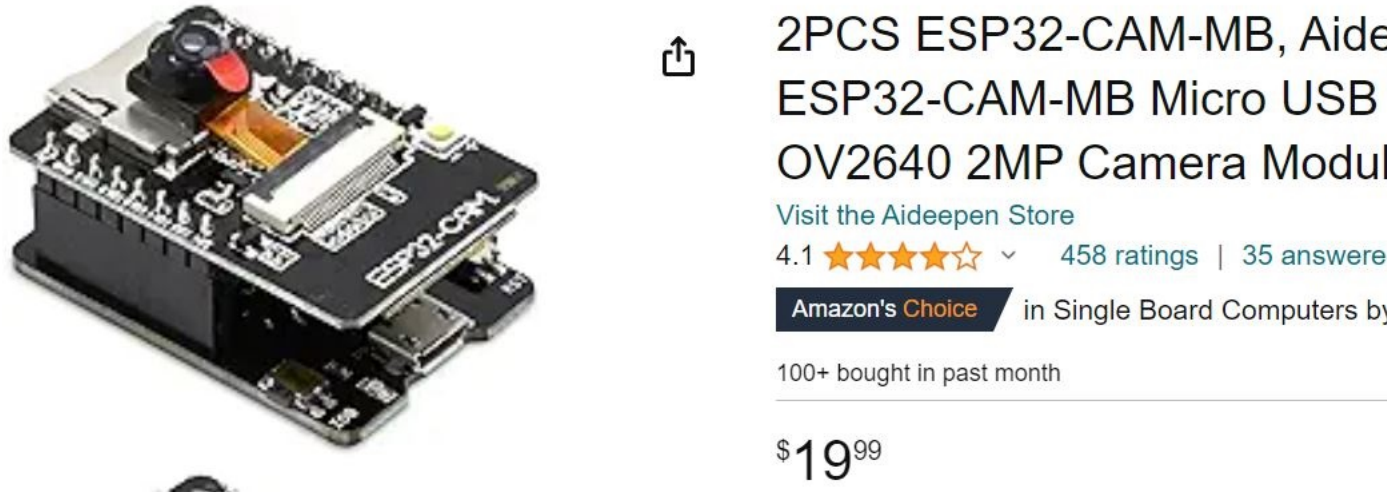
For further capabilities the following are considered for follow on projects:

- An interface for serial port data to be collected from hard wired sensors for logging - this could include standard RS232 type signaling as well as other serial protocols such as 1Wire, IIC, or SPI.
- A Bluetooth gateway that allows sensors with BT or BTLE connections to post through the WiFi service.
- A LoRa gateway that allows sensors with LoRa connections to post data through the WiFi service.

In all these cases a secondary ESP32 with it's own application can be used to push data to the edge server. It's even possible that a single ESP32 could server as all three gateways. This is a follow on project for a bit later.

Hardware Design Notes

For a hardware platform I selected that Aideepen ESP32-CAM board. This board is quite small and it has an SD card reader attached. It also has one other important feature - there is a tiny connector for an external WiFi antenna. This may be important in some applications where increased range of the WiFi signal is important - for remotely located sensors or weak signal from an existing WiFi signal source.



These boards are readily available on Amazon for a relatively low cost ($20 for 2 boards). The SD card reader is documented for up to a 4GB card, however I've been successfully using a 32GB Samsung A1 micro-sd card.

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. The board comes with a camera which isn't used in this application. You can remove it if you want. Or you can use it to make a remote camera that takes periodic photos and uploads them to this edge server. For an example, see this article.

For power, these boards take a 5V supply - could be powered directly from a USB port or from some of the common charger batteries that are sold to recharge your phone.

I've been told that this ESP32-CAM board design is often cloned by low cost manufacturers - so be aware that some of those manufacturers may have quality issues. I purchased two of the boards above from Amazon - fortunately, both of them seemed to work fine.

Software Architecture

The software uses several components including an FTP Server and HTTP server. Credit to Rui Santos at Random Nerd Tutorials for some great work on the ESP32 platform!

The following existing components were used in building this application:

- ESP32Time - time keeping and calendar module
- NTPClient - retrieve network time and date from an NTP server
- FTPServer - modified slightly for a bit more functionality
- WiFiClient - implements a very minimal HTTP server for status and posting sensor data

Much of the code is dedicated to setting up these services and using them.

Configuration File

When the application boots up it will read a file from the SD card called config.ini. This file contains information required for the correct operation of the edge server. It is a simple text file with lines that are in Key=Value format.

You should prepare the configuration file and place it in the root folder of the SD card that you put into the SD card slot before the ESP32 is powered up.

The information required in this file defines the WiFi configuration and FTP server configuration. The configuration file has a minimum of 5 lines. It is case sensitive, so be aware!

SSID=wifissid

PASSWORD=wifipassword

MODE=xxx AP or STATION

MODE=AP

AP means the edge server configures itself as a WiFi access point. This means that other devices can connect directly to the ESP32 for communication. The SSID and PASSWORD are used in this mode for the remote devices to connect to it. In this mode the ESP32 will have an address of 192.168.4.1. Remote sensors will need to be configured to contact the edge server on this address to upload data to it.

MODE=STATION

STATION means the edge server connects to some existing WiFi network with the SSID and PASSWORD. In this mode, the ESP32 will be assigned it's own IP address by the AP it connects to. This IP address will need to be used by any sensors to send data to it.

FTPUSER=ftpusernaee

FTPPASSWORD=ftppassword

These two lines determine the user name and password that sensors need to be configured for in order to access the FTP server to upload data. Sensors that post data do not need to be configured with this information.

```
SSID=iothub1445
PASSWORD=soWhat1445
FTPUSER=iothub
FTPPASSWORD=soWhat1445pwd
MODE=AP
```

Error Notification

Certain errors that may be encountered - especially during start up - will make it impossible for the edge server to fully boot and become operational. To report such errors, the on-board LED is used to blink an error code.

The ESP32 will blink out "SOS" followed by 1 or more blinks. The number of blinks after the SOS is the error code. The error codes you might see are:

- 1 blink - SD card could not be accessed for some reason. Make sure it's formatted as FAT32 and that there is a config.ini file on the root folder with the above 5 lines in it. You can use Windows Notepad to create the file and save it as config.ini
- 2 blinks - Could not connect to WiFi - either the signal is too weak or perhaps incorrect SSID or PASSWORD is in the config.ini file

Log Files

Data from sensors will be stored on the SD card. For FTP uploads, each sensor or sensor group could upload to it's own log file, or all sensors could upload to their own log file. The FTP server supports APPEND mode so sensor data can be appended to a specific log file (that way each upload doesn't have to be a separate file, although it can be if that suits your requirements).

For HTTP post service uploads, a line is appended to a common log file - iotdata.log in the root folder. It's wise for the data that a sensor data would include the id of the sensor that took the data. POSTed data is time-tagged. This file can be retrieved using the FTP server. Typically at the time some remote FTP client connects to collect the data, it would grab the appropriate data file(s) including iotdata.log, and upon successful upload, it can use FTP to delete the files so that sensor data won't be duplicated on the remote server.

In addition an internal log file, there is a log file for activities of the IoT Hub - it is called iothub.log. This file will have entries that indicate significant events such as boot up, WiFi status, NTP status, and so forth.

Setting Date and Time

The IoT Hub will time tag logged data from the post service lines, and will time-tag uploaded files with the time of upload. However, it must somehow be notified of the correct date and time to form accurate time tags.

In STATION mode the IoT Hub will try to automatically obtain the date and time from an NTP (Network Time Protocol) server somewhere on the internet. If the IoT Hub can not reach an NTP server this will fail.

In AP mode the IoT Hub isn't connected to the internet, so NTP servers are not available.

To allow accurate time stamps - a special post service command is implemented to set the date and time. Typically this would be done one time when the IoT Hub is first powered up. After that it will be able to keep track of the time - although the accuracy of that clock is dependent on the ESP32's internal clock so it may drift over time.

The IoT Hub internal clock can be set by sending a special post service command that starts with a tilde (~) character followed by 14 digits (without spaces or any separators). This is in the format of yyyymmddhhmmss, for example 20240206143456 for February 6, 2024 at 14:34:56. This would look like an HTTP command of the following format sent to the HTTP server (on default port 80) of the IoT Hub.

```
GET /?LOG=~20240206143456 HTTP/1.1
```

Logging Data with the HTTP server

Sensors with lower data rates can send a message to the HTTP server (on default port 80) to send a line of data to the iotdata.log file. The form of this would be:

```
GET /?LOG=my%20temperature%20is%2056%20degrees
```

Note that anything besides alpha (A-Z, a-z) or numeric (0-9) characters must be 'hex escaped' in traditional URL format - for example %20 is a space character, so the above command would log

my temperature is 56 degrees

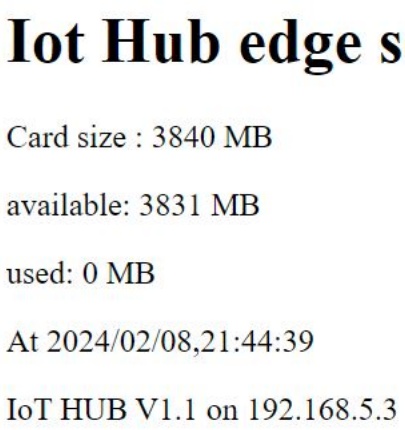
to the iotdata.log file.

This is an HTTP GET command with a parameter LOG set to the line that the remote sensor wants to send to the IoT Hub's ioddata.log file. This line will be appended to the log file with a time-tag (using the date/time of the IoT Hub) so it will look like this:

```
2024/02/06,14:34:56,my temperature is 56 degrees
```

Web Page

When a browser requests the home page of the IoT Hub - for example typing http://192.168.5.3 (in this example for STATION mode, or http://192.168.4.1 for AP mode) into the address bar, the IoT Hub will serve up a web page that shows some internal information such as the utilization of the SD card and the current date and time that the IoT Hub is maintaining.



Startup Tasks

When the power is first turned on, the ESP32 boots up the IoT Hub application. The first things done are to set up the hardware and communication features that are used.

You'll see below the setup() function which does the following:

- Initialize the serial port an output a sign-on message
- Initialize the signal LED so we could output any error messages
- Mount the SD card - if this fails - we stop and flash an error message
- Read the configuration file (config.ini)
- Initialize the real-time clock (RTC)
- If the configuration file indicates AP mode, start up the ESP32 WiFi as an access point
- If the mode is STATION, then connect to the WiFi signal and attempt to get the date and time from NTP
- Initialize and start up the web server
- Initialize and start up the FTP server

Shrink ▲

```
//------------------------------------------------------
// Setup - called once on boot-up
void setup()
{
  char buf[32];

  Serial.begin(115200);
  Serial.println(SIGNON);

  pinMode(SIGNALLED,OUTPUT);

  // Mount SD card file system
  if(!SD_MMC.begin())
  {
    PRINTLN("SD_MMC Card Mount Failed");
    blink(ERR_NOCARD);
  }
  else
  {
    uint8_t cardType = SD_MMC.cardType();

    if(cardType == CARD_NONE)
    {
      PRINTLN("No SD card attached");
      blink(ERR_NOCARD);
    }
    else
    {
      PRINT("Mounted SD_MMC card successfully, type is ");

      if(cardType == CARD_MMC){
        PRINTLN("MMC");
      } else if(cardType == CARD_SD){
        PRINTLN("SDSC");
      } else if(cardType == CARD_SDHC){
        PRINTLN("SDHC");
      } else {
        PRINTLN("UNKNOWN CARD TYPE");
      }
    }
  }


  // read configuration file
  readKey(CONFIGFN, "SSID=", ssid, 63);
  readKey(CONFIGFN, "PASSWORD=", password, 63);
  readKey(CONFIGFN, "FTPUSER=", ftpuser, 63);
  readKey(CONFIGFN, "FTPPASSWORD=", ftppwd, 63);
  readKey(CONFIGFN, "MODE=", buf, 31); // MODE=AP or MODE=STA, station mode by default
  isApMode = (strcmp(buf,"AP") == 0);
  Serial.println(isApMode ? "AP MODE" : "STATION MODE");


  // initialize the RTC and read network time if possible
  rtc.setTime(12,0,0,1,1,2024); // default time 12:00:00 1/1/2024

  // set up WiFi in AP or STATION mode
  if (isApMode)
  {
    // Access point mode
    WiFi.softAP(ssid, password);
    IPAddress IP = WiFi.softAPIP();
    Serial.print("Access Point IP address: ");
    Serial.println(IP);
  }
  else
  {
    // WiFi station mode - connect to an access point
    // on first boot, try to connect to WiFi and get time/date from NTP
    connectToWiFi();
    if (!connectToWiFi) blink(ERR_NOWIFI);
    needNtp = true;
    getNtpTime(); // get NTP time if possible
  }

  PRINTLN("Starting Web service");
  server.begin(); // start web server

  PRINT("Initializing FTP server - user ");
  PRINT(ftpuser);
  PRINT(" password ");
```

```
  PRINTLN(ftppwd);
  ftpSrv.begin(ftpuser,ftppwd); // start FTP server
}
```

Operational Tasks

After the startup tasks are completed, the loop() function is executed repeatedly until power is removed. In this function we place the operational tasks - serving communications tasks mainly.

The code loops forever doing the following:

- Handle any FTP clients that connect, transferring file and so forth
- Handle any HTTP clients that connect, reading the request that the client sends in, processing that request if there is a request to post some data to the log file or to set the date and time.

We need to be careful when servicing this so that if an HTTP request and an FTP request come in at the same time, we don't ignore one and handle the other one. It's sort of a simple time-sharing system to make sure we give time to both communications methods. Normally these won't come in at the same time - but it could happen that some sensor tries to post data while another sensor or remote user is connected to FTP at the same time.

Shrink ▲

```
//--------------------------------------------------
// loop() - called forever after setup() is called
void loop()
{
  for (;;)
  {
    // sort of a crude round-robin scheduler
    ftpSrv.handleFTP();

    if (webClientConnected)
    {
      if (httpScanCounter++ >= MAXHTTPSCANCOUNTER)
      {
        webClientConnected = false;
        if (client)
        {
          client.stop();
        }
        Serial.println("\nForced client disconnect");
      }
      // a client is connected, try to read the request line
      // request line will be something like this:
      // GET /?var1=val1 HTTP/1.1
      //    GET /?LOG=123%20ABC HTTP/1.1
      //   ?var1=val1&var2=val2 ,,,
      //   values have embedded hex for some characters like  "1%202" means "1 2"
      //   basically any char that isn't alpha/numeric is %xx
      //
      else if (!client)
      {
        webClientConnected = false;
        client.stop(); // client unexpectedly disconnected
        //Serial.println("\nunexpected client disconnect");
      }
      else if (client.available())
      {
        char c = client.read();
        //Serial.write(c); // echo for debugging
        if (c == '\n')
        {
          //Serial.println();
          processHeader(httpHeader);
          replyWithHomePage(client);
          client.flush();
          client.stop();
          webClientConnected = false;
          //Serial.println("Web client disco");
        }
        else if (httpHeaderPtr < (HTTPHEADERMAXLEN-1))
        {
          httpHeader[httpHeaderPtr++] = c;
          httpHeader[httpHeaderPtr] = '\0';
        }
      }
    }
    else
    {
      // at the present time no client is connected,
      // see if one is knocking on our door
```

```
      client = server.available();
      if (client)
      {
        webClientConnected = true;
        httpHeader[0] = '\0';
        httpHeaderPtr = 0;
        httpScanCounter = 0;
        Serial.println("Web client connected");
      }
    }
  }
}
```

The complete code is available on GitHub

History

V1.0 February 2, 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