Articles / Internet of Things / Arduino

C++ Arduino Bluetooth Low Energy

A DIY Bluetooth LE to WiFi Bridge

deangi
4 Mar 2024 CPOL 0

A Bluetooth LE scanner gathers data on a schedule from one or more BLE servers and forwards it to WiFi.

Introduction

Bluetooth low energy (BLE) is a 2.4 GHz radio communications technology that supports communication in the local domain between devices. It is well suited for low power communications, but it's range is limited to roughly 10 meters - although it can be extended with additional power and more sophisticated antenna systems.

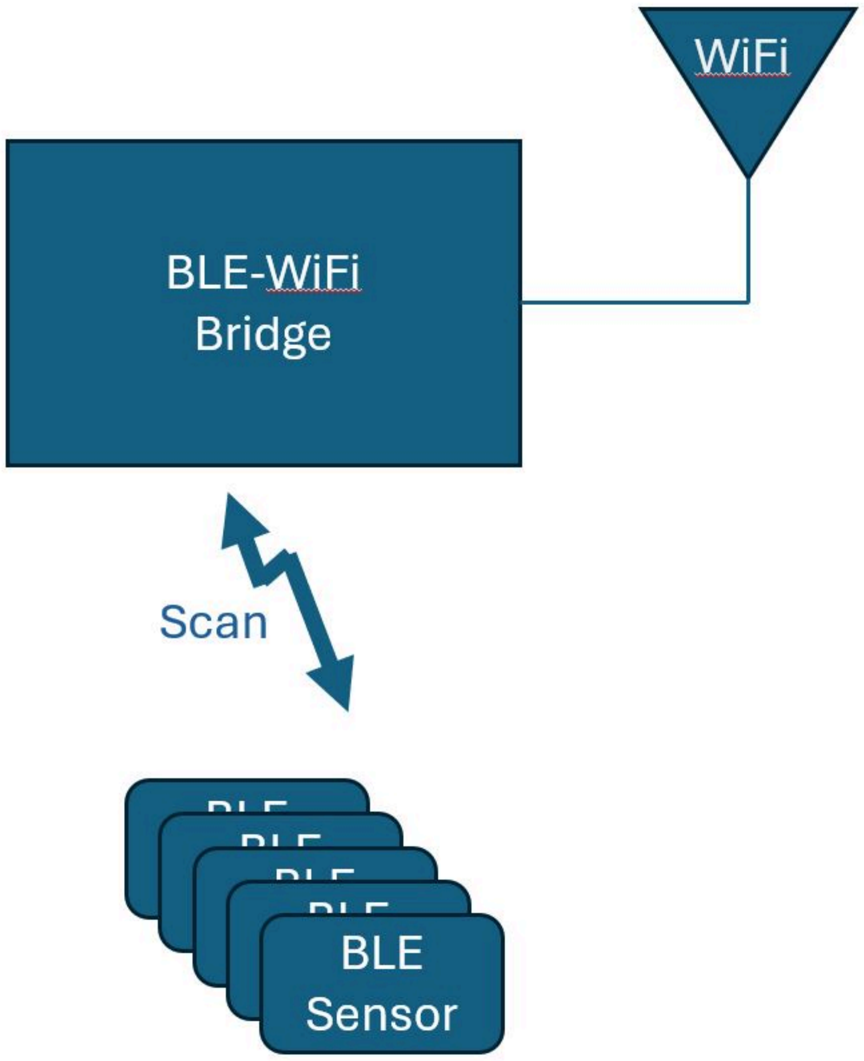
A remote installation may have multiple BLE devices sensing data in a home or industrial application. This project will allow information from BLE devices to be queried on a defined schedule and forwarded to a remote server over a WiFi network link.

In one example things like temperature, humidity, door openings, alarm conditions, water leaks, gas leaks, electrical use, water use, etc. could be sensed by BLE devices. A BLE-WiFi bridge enables this data to be collected from multiple devices and forwarded to a networked device such as a server for storage and/or further processing.

BLE WiFi Bridge Overview

The bridge is implemented using an ESP32 platform - a small and low cost embedded platform that supports both BLE and WiFi operation. No hardware external to the small development board is required for the application.

The main features are a configurable BLE scanner which allows you to define up to 100 different server characteristics, and how often each characteristic is to be read. In operation, defined server characteristics are read and moved to a WiFi connected server. (See previous article on IoT Edge Hub)



BLE Servers and Clients

The Bluetooth communications specifications are quite extensive and if printed out may reach the size of a multi-thousand page book. For purposes of this article we're starting at a bit of a simplified level - so buckle up!

For purposes of this application we're viewing the BLE world as having two types of devices - SERVERS and CLIENTS. A BLE server hosts some kind of measurement and/or control capability that is presented by one or more pieces of data called "Characteristics" - which are pieces of data that might be read or written by a BLE client.

Some examples of characteristics might be temperature, humidity, relay control (on/off), volume control, time, date, GPS location, and etc.

Some characteristics are "read only", some "write only", and some are "read/write".

A server may have one characteristic, or it may have hundreds of characteristics. Inside a server, characteristics are organized into groups called "services". A single BLE client can have one or multiple services. In some sense, services are like the "top level directory" of a BLE server. In communication with a BLE server, you can request a directory of it's services.

Under each service are multiple characteristics - each characteristic might be viewed as a tiny file - where you can read or write a piece of data. Some files are "read only" and some are "write only" and some are "read/write" - depending on how the server has defined each characteristic.

Bluetooth Low Energy (BLE) Overview

BLE defines 40 channels in the 2.4 GHz band for RF communications. The available data rates are from 125 kbps to 2 mbps depending on conditions using Gaussian frequency shift modulation. The data is sent in packets and over a frequency shifting pattern to allow multiple BLE devices to share the spectrum. An encryption option is available using 128 bit AES encoding.

The ESP32 SDK provides software to implement BLE devices. BLE consists of a local-area-network communicating over 2.4 GHz radio frequencies using the BLE protocol (link) From a software viewpoint, the main features of BLE we need to deal with are:

- Endpoint type (client / server)
- Services with Characteristics

Essentially, a BLE server allows other devices to connect to it, and may allow other devices to read or write characteristics. When another device reads a characteristic, it may be something like a measured value such as temperature that is "served" to a connected client.

When another device writes a characteristic, it may be something like opening or closing a relay to turn a light on or off that is "served" to a connected client.

The complete specifications are about 100 cm thick of printed paper, so we won't try to go too deep in this discussion!

Characteristics may have a descriptor attached - a string that describes the characteristic. For example, a characteristics that represents a measured temperature might have a descriptor of "temperature, F".

In BLE, one or more characteristics are grouped under a "service". And a single BLE server may have one or more "services". So a list of services provided by a BLE server is sort of like a top level directory of the server. Characteristics are sort of a second level directory under each service.

So a typical operation for a client wishing to read characteristics would be:

- Connect to the device and query its services - it finds a single service by matching the service's UUID against the UUID of the service it's looking for.
- Query a list of characteristics under that service - it finds one or more characteristics - again each characteristic is identified by a UUID. The client will want to match the UUID of the characteristic it's looking for against the list provided by the server.
- Read the value of the desired characteristic(s) using the service UUID and characteristics UUID.

So now there's another topic to discuss - what is a UUID? It turns out that BLE services and characteristics all are identified by a UUID. They are uniquely named using a long hex string of 36 digits called a UUID (Universally Unique Identifier). These UUIDs are used in many domains (not just BLE) whenever something needs to be tagged with a unique name in the universe. These can be generated several ways, but one easy way is to use this special web site.

Each service and each characteristic must have its own UUID for BLE.

Now it turns out that there are some services and characteristics that are so common they are assigned special UUIDs that are the same - these are part of a universe of BLE jargon called GATT (Generic Attribute Profile). You can look on this website for further information. For this BLE server, we have not used a GATT - although one probably exists for temperature and humidity. Feel free to experiment with this!

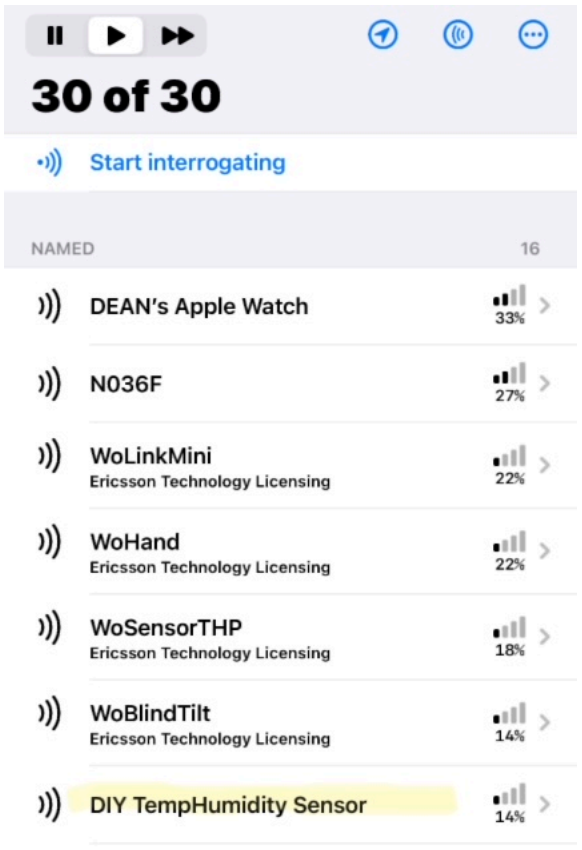
BLE Advertising

In order to connect to the device, a client must know what device to connect to. To solve this issue, BLE defines something called "advertising" where an active BLE Server will periodically "advertise" itself by sending a special BLE radio packet. A client can listen for these packets to discover what BLE servers might be in its range. It can then pick one of these servers to connect to.

Each particular BLE device is assigned a unique number (not a UUID - but similar). This number is called a MAC address (media access control). See this website for further information. Basically a MAC address is a unique 48 bit number assigned to a particular instance of some hardware. Ethernet, WiFi, and Bluetooth all use these MAC addresses, which are typically specified by 6 bytes in hex separated by colons - looking something like this: 05:6c:22:1f:be:3a. Each BLE device is supposed to be shipped with it's own unique MAC address. So with a MAC address, even if we have 100 servers that are all identical - each one will have a unique MAC address that we can use to connect to one of the 100 identical servers.

There is a very handy application (iPhone and Android I think) called BT Inspector which can be used to illustrate this. This is a diagnostic BLE client that allows the phone to listen for BLE servers that are advertising within its range. You can then use the application to try to connect to a server and interrogate its services and characteristics. The application will allow reading and writing of characteristics as well. It's an amazing testing device!

Here are some examples of using BT Inspector. First item is to scan for advertising BLE servers. Press the **Scan** button and the app on your phone will start listening for BLE servers transmitting advertising packets. When these are received, the data is displayed on the phone screen as shown below. Note at this point these BLE devices are not "connected" - the application is just displaying the data that BLE servers in range are sending in the advertising packets they send.


30 of 30

Start interrogating

NAMED 16

DEAN's Apple Watch	33%
N036F	27%
WoLinkMini Ericsson Technology Licensing	22%
WoHand Ericsson Technology Licensing	22%
WoSensorTHP Ericsson Technology Licensing	18%
WoBlindTilt Ericsson Technology Licensing	14%
DIY TempHumidity Sensor	14%

BLE Connections

When a client has identified a particular BLE server by decoding it's advertising packet, the client can attempt to negotiate a connection to the server. This may or may not involve an authentication process depending on the server. For purposes of this article we're not assuming any of the server devices require authentication.

Once a connection is established, the client can ask the server for a list of it's available services, and for each service, the client can ask for a list of available characteristics.

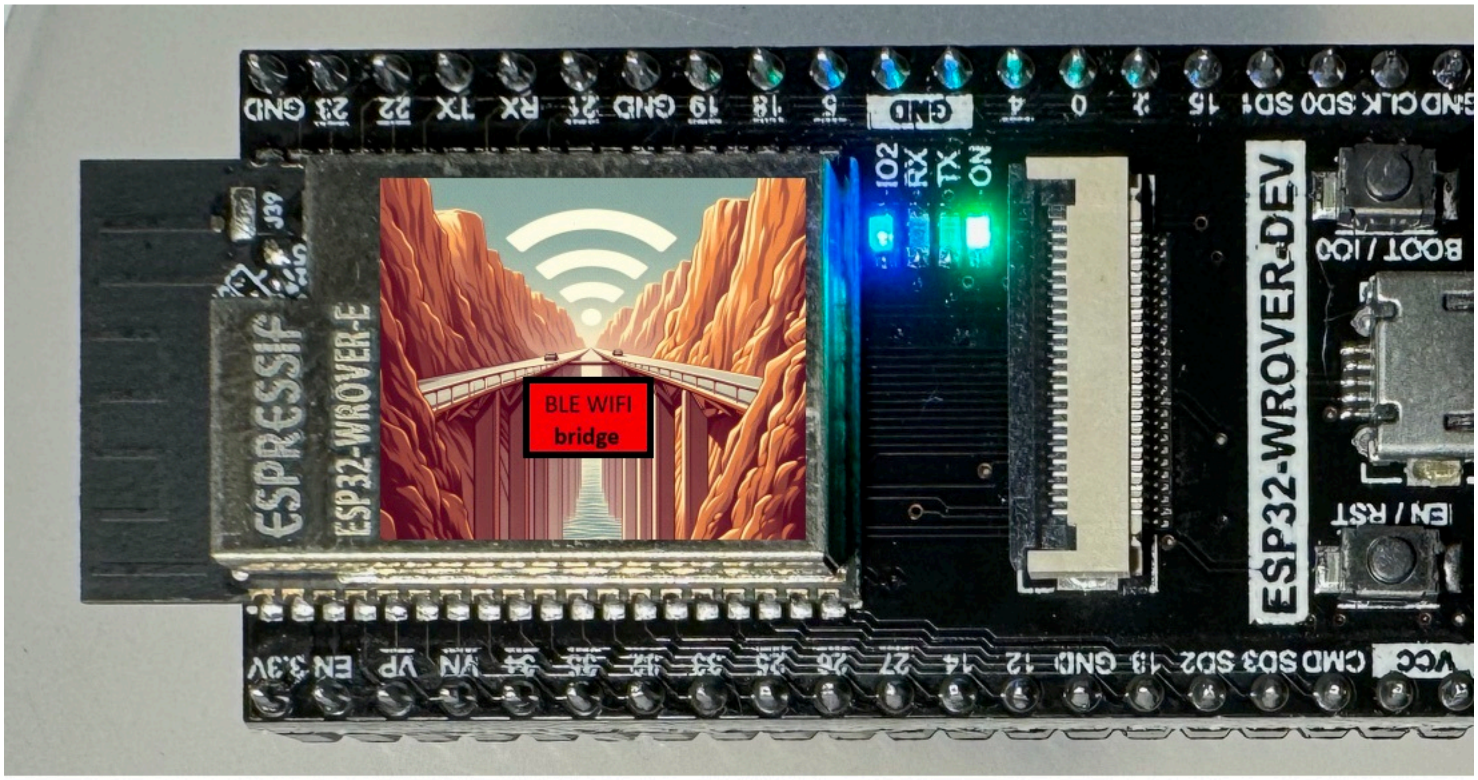
Here is some ESP32 code to query all of this data from a client. You'll see that the code tries to connect to a server, and if that is successful, it requests a list of services. Then for each service, it asks the server for a list of characteristics associated with that service. This information is printed to the ESP32 serial port.

C++ Shrink ▲

```cpp
//------------------------------------------------------------------------
// Completely explore a server - list all services and all characteristics
//   This will connect to a server by MAC address (serverAddress) and
//   query all of it's services.   For each service it will query all
//   of the associated characteristics.

void serverExplorer(BLEClient * pClient, BLEAddress serverAddress)
{
  pClient->connect(serverAddress);
  if (!pClient->isConnected())
  {
    Serial.println("Connection failed");
  }
  else
  {
    Serial.println("Connected");
    std::map<std::string, BLERemoteService*>* services = pClient->getServices(); // get list of services

    Serial.printf("Number of services found: %d\n",services->count);

    // print services
    int svcidx = 1;
    for (std::map<std::string, BLERemoteService*>::iterator it = services->begin(); it!=services->end();++it)
    {
      Serial.printf("Service %d\n", svcidx++);
      delay(200);
      std::string svcName = it->first;
      std::string svcSvid = it->second->toString();
      Serial.print(svcName.c_str());
      Serial.print("=");
      Serial.println(svcSvid.c_str());
      BLERemoteService* p = it->second;
      BLEUUID svcUUID = p->getUUID();
      //std::string value = p->getValue(p->getUUID());
      //Serial.println(value.c_str());
      // get characteristics
      std::map<std::__cxx11::basic_string<char>, BLERemoteCharacteristic*>* characteristics = p->getCharacteristics();
      for (std::map<std::__cxx11::basic_string<char>, BLERemoteCharacteristic*>::iterator itch = characteristics->begin(); itch != 
characteristics->end(); ++itch)
      {
        BLERemoteCharacteristic * rc = itch->second;
        BLEUUID charUUID = rc->getUUID();
        std::string cuuid = charUUID.toString();
        std::string val = p->getValue(charUUID);
        Serial.print('['); Serial.print(cuuid.c_str()); Serial.print("]="); Serial.println(val.c_str());
      }
    }

    pClient->disconnect();
  }
}
```

Hardware Architecture

Since this application uses only the built-in hardware of the ESP32 development board, most any ESP32 (WROOM or WROVER preferred) will work. The application blinks the on-board LED which in different development boards may be connected to a different GPIO pin. In the source code, GPIO2 is defined for the LED, but if you use a different platform, check to see if this needs to be changed.



Software Architecture

Three main components:

- WiFi communications - passing data to a networked device, NTP
- BLE communications - looking for advertised devices, connecting, querying data
- Scheduling and control, BLE scan and query scheduling, etc.
- Setup - reading config file and parsing it, etc.

WiFi Communications

For the WiFi end of things we use mostly the built-in SDK drivers for WiFi, UDP, and HTTPClient features. In addition an NTP client is used to try to fetch the correct date/time from the network.

To post some data that was read from a BLE server on to a WiFi connected server, an HTTP client is created and a log request is sent to a remote HTTP server. This utilizes the IoT Edge Hub (article referred to earlier) to capture the data and add it to a log file. At some later point this log file can be accessed by a back-end IoT processor using FTP to retrieve it from the IoT Edge Hub. The back-end processor can archive and/or process the BLE data as needed for the IoT application.

BLE Communications

The ESP32 BLE SDK is used to create a BLE Client device. This client device is used for two duties:

- Advertising **scanning** to locate BLE devices of interest (referenced in a configuration file).
- On a scheduled basis, up to 100 BLE server **characteristics are read** and the data is transferred via WiFi to the IoT Edge Hub.

BLE Characteristics defined in configuration file

BLE server characteristics to be scanned are defined by one or more lines in a configuration file stored on the device. This file can define up to 100 values to be scanned from BLE devices and transferred to the IoT Edge Hub. Values can be scanned periodically on a minute based schedule - so some values can be read every minute, others may be read every 10 or 15 minutes. The read period is also defined in the configuration file.

If more than 100 values need to be scanned - the defined constant MAXVALUES2READ can be increased in the source code.

In the configuration file, values to be scanned include the following information:

- An identifying tag - to identify the value for future processing - something like "TEMP-Garage" might be used to identify a BLE device that monitors temperature in a garage.
- A schedule period - some number of minutes for the time between each read of the parameter - this can be from 1 to 1440 (once per minute to once per day).
- The advertised name of the BLE device, or the MAC address of the BLE device in xx:xx:xx:xx:xx:xx format
- The UUID of the service on the BLE device that has the desired characteristic
- The UUID of the characteristic on the BLE device to read

Some examples might look like this (VALUE1 ... VALUE100 will be scanned)

```
VALUE1=TempF,15,DIY TempHumidity Sensor,b7972d95-e930-4144-beb0-6a6e8b9a3d23,20b5e09a-f998-47f0-aae3-4b361ebc8233
VALUE2=Time-Date,15,DIY TempHumidity Sensor,b7972d95-e930-4144-beb0-6a6e8b9a3d23,711d51a8-f76b-4c24-ad4a-8a3059d2489b
VALUE3=Humidity,15,DIY TempHumidity Sensor,b7972d95-e930-4144-beb0-6a6e8b9a3d23,5ed64822-1dc1-4ebd-8e23-f0847e380841
```

BLE Bridge Initialization Tasks

There are some initialization tasks done when the BLE Bridge is first booted up. These tasks set up the various hardware devices and software structures needed to being operations. These are done in the setup() function of the application.

There is usually a built-in LED on an ESP32 development boards. For this app the LED is used to indicate some activity - it's called a "comfort" LED - indicating that the application is doing something. The comfort LED is used for:

- Turns on during startup initialization processing
- Blinks when advertising scan is receiving BLE advertising packets
- Blinks on when reading characteristics from a BLE device
- Blinks on and off at 1 second intervals when the BLE bridge is idle and has nothing to do

Here is the code for the initialization tasks - the setup() function.

You'll see the following sections:

- Initialize and light up the comfort LED
- Initialize the real-time-clock (RTC) to January 1, 2024 at 00:00:00
- Initialize the diagnostic serial port and output a startup signon message
- Try to mount the on-chip file system SPIFFS - if this can't be accomplished, then the application stops and blinks an error code.
- Read the configuration file from the SPIFFS - this is a file /config.ini stored in the root folder of the SPIFFS with between 5 and 105 lines - a text file that looks like this:

    -
      ```
      WIFISSID=MySsid
      WIFIPWD=MyWiFiPassword
      IOTHUBADDR=192.168.5.3
      BLENAME=BLE WIFI Bridge V1.1
      VALUE1=TempF,15,DIY TempHumidity Sensor,b7972d95-e930-4144-beb0-6a6e8b9a3d23,20b5e09a-f998-47f0-aae3-4b361ebc8233
      VALUE2=Time-Date,15,DIY TempHumidity Sensor,b7972d95-e930-4144-beb0-6a6e8b9a3d23,711d51a8-f76b-4c24-ad4a-8a3059d2489b
      VALUE3=Humidity,15,DIY TempHumidity Sensor,b7972d95-e930-4144-beb0-6a6e8b9a3d23,5ed64822-1dc1-4ebd-8e23-f0847e380841
      # EOF
      ```
    - The first two lines define the ssid and password of the WiFi network to connect to.
    - Third line is the IP address of the IoT Edge Hub where data is to be stored
    - Fourth line is the name of the BLE client that will be created to scan BLE servers for data
    - 1 to 100 lines defining BLE characteristics to be read on schedule

- Connect to WiFi - if this cannot be done, the application will stop and blink an error code
- Try to retrieve the current date/time from NTP
- Create the BLE client and start a scan for BLE advertising packets
- Initialize the ESP32 watch-dog-timer to reset the ESP32 if the application were to crash
- Initialize some misc application variables needed for operation

C++ Shrink ▲

```
//--------------------------------------------------
void setup()
{
  char tmpbuf[256];
  char tmpname[32];

  //--- initialize the comfort (signal) LED
  pinMode(SIGNALLED, OUTPUT);
  ledmode=HIGH;
  // turn comfort LED on for the setup processing
  digitalWrite(SIGNALLED,ledmode);

  //--- initialize the RTC
  rtc.setTime(0,0,0,1,1,2024);

  //--- initialize the diagnostic serial port
  Serial.begin(115200);
  Serial.println(SIGNON);

  //--- initialize SPIFFS (file system)
  if(!SPIFFS.begin(true))
  {
    print("An Error has occurred while mounting SPIFFS");
    blink(ERR_NOSPIFFS);
    //return; what to do here?  We can't do much without the file system
  }

  //--- read the configuration file
  readKey(CONFIGFN,"WIFISSID=",wifissid,63);
  readKey(CONFIGFN,"WIFIPWD=",wifipwd,63);
  readKey(CONFIGFN,"IOTHUBADDR=",iothubip,63);
  readKey(CONFIGFN,"BLENAME=", btlebridgename, 63);
  // values to read
  nValues=0;
  for (int i = 1; i < MAXVALUES2READ; i++)
  {
    // VALUE1=15,11:22:33:44:55:66,b7972d95-e930-4144-beb0-6a6e8b9a3d23,20b5e09a-f998-47f0-aae3-4b361ebc8233
    // VALUEn=minutes,deviceId,ServiceUuid,CharacteristicUuid
    sprintf(tmpname,"VALUE%d=",i);
    readKey(CONFIGFN,tmpname, tmpbuf, 255); // try to read it
    if (tmpbuf[0] != '\0') // found "VALUEn="
    {
      // yes, found one, allocate and initialize an entry to read it and keep track of it
      Serial.print("\nParsing value: "); Serial.println(tmpbuf);
      vals2read[nValues] = new ValueToRead();
      char* result = vals2read[nValues]->set(tmpbuf);
      Serial.println(result);
      Serial.println(vals2read[nValues]->toString());
      nValues++;
    }
  }
  Serial.printf("%d values read from config file\n",nValues);

  //--- connect to WiFi
  if (!connectToWiFi()) blink(ERR_NOWIFI);

  //--- see if we can get date/time from WiFi - Network Time Protocol (NTP)
  getNtpTime();

  //--- initialize BLE client and start scanning for BLE advertising packets
  BLEDevice::init(btlebridgename);
  pBLEScan = BLEDevice::getScan(); //create new scan
  pBLEScan->setAdvertisedDeviceCallbacks(new MyAdvertisedDeviceCallbacks());
  pBLEScan->setActiveScan(true); //active scan uses more power, but get results faster
  pBLEScan->setInterval(100);
  pBLEScan->setWindow(99);  // less or equal setInterval value
  pClient = BLEDevice::createClient(); // create client to use for connecting, reading values

  //--- Initialize watch-dog-timer to reset the BLE bridge if it goes off line
  esp_task_wdt_init(WDT_TIMEOUT, true); // enable WDT so the ESP32 restarts after 10 seconds
  esp_task_wdt_add(NULL); // enable WDT for this task

  //--- initialize some application variables
  minuteCounter = -1;
  timeForScan = true;

  lastMin = rtc.getMinute();

  //--- LED off at end of setup
  LEDOFF;
}
```

BLE Bridge Operational Tasks

During operational mode, the BLE bridge has some tasks done at one second intervals, some at one minute intervals, and some done on-demand.

The following tasks are defined in operational mode:

- Blink the comfort LED - done at 1 second rate
- Scan for values to read - every minute we check if there are BLE values to read
  - For each BLE value found in the configuration file - check if the current minute count is evenly divisible by the read-period for that value. If it is, the value is read and added to a queue to be sent to the IoT Edge Hub
  - Sometimes there won't be any values to read - so nothing much happens on those minutes. Other minutes there may be multiple values to read.
  - One time per hour, kick off another scan for BLE advertising packets. Do this on initial startup after the setup() function is finished as well.
- Scan for advertising packets - hourly
  - on start-up and every hour we start the BLE client scanning for advertising packets.
  - If a packet is discovered, a call-back function in MyAdvertisedDeviceCallbacks()
  - This call-back will look through the list of values to be read (that was obtained from the configuration file) . If the advertised name or mac address from the received advertising packet matches one of the devices we're interested in, we can mark it as available for reading. If a value is defined for a device that we don't receive any advertising packet from, the we have to assume that device is unavailable and we ignore it for the next hour. Each hour we'll scan again - and if we find it, we'll activate scanning for that value.
- WiFi reconnect - on demand - if WiFi disconnects for some reason, we'll try to reconnect. This might happen if there is some temporary power interruption, signal interference or blockage.

C++ Shrink ▲

```cpp
//--------------------------------------------------
// Operational tasks
void loop()
{
  LEDOFF;
  for (;;)
  {
    int currentSec = rtc.getSecond();
    if (currentSec != lastSec)
    {
      //--- one second tasks
      lastSec = currentSec;
      ledmode ^= 1;
      digitalWrite(SIGNALLED,ledmode);
      // The ESP32 has a watch-dog timer (WDT).  This is a timer that counts from a starting
      // point down to 0.   By calling esp_task_wdt_reset() you can reset the WDT back to it's
      // starting point.  This is "kicking the WDT".
      //
      // If for some reason, the WDT doesn't get reset before it gets down to 0, then
      // the ESP32 will be reset and it will reboot.
      //
      // This is a "last ditch" effort to recover the application in case it should run
      // into some unexpected dead state such that the WDT isn't getting kicked in time.
      esp_task_wdt_reset(); // reset watch-dog timer
    }

    // keep track of minutes going by in a variable minuteCounter
    int currentMin = rtc.getMinute();
    if (currentMin != lastMin)
    {
      //--- one minute tasks
      LEDON;
      lastMin = currentMin;
      minuteCounter++;
      Serial.print("Minute processing: ");
      Serial.println(minuteCounter);

      if ((minuteCounter % 60) == 59) timeForScan = true; // rescan BLE each hour

      // now read all devices that are due for this minute
      for (int i = 0; i < nValues; i++)
      {
        ValueToRead* p = vals2read[i];
        long mod = minuteCounter % p->minutesBetweenReads;
        if ((mod == 0) && (p->deviceAddr[0] != '\0'))
        {
          // it's the right minute to read the value,
          // and we have seen the device advertise in a BLE scan

          readValue(pClient, p, buf,256);
          String ttag = rtc.getTime("%Y/%m/%d,%H:%M:%S");
          sprintf(buf,"%s,%s,%s",(char*)ttag.c_str(), p->valueTag,buf);
          int sts = measuredData.push(buf);
          if (sts == 0) Serial.println("Measurement queue overflow");
        }
      }
      LEDOFF;
    }

    //-- on demand task - rescan for BLE devices
    if (timeForScan)
    {
      BLEScanResults foundDevices = pBLEScan->start(scanTime, false);
      Serial.print("Devices found: ");
      Serial.println(foundDevices.getCount());
      Serial.println("Scan done!");
      pBLEScan->clearResults();

      timeForScan = false;
    }

    //--- on demand task - if WiFi disconnects, try to reconnect
    if (WiFi.status() != WL_CONNECTED)
    {
      // WiFi is disconnected, so try a reconnect
      delay(200);
      connectToWiFi();
      if (wifiIsConnected) getNtpTime();
      lastMin = rtc.getMinute();
    }

    // push any queued data to the IoT hub
    if (measuredData.isEmpty() == 0)
    {
      // send any queued measurement data to server
      buf1[0] = '\0';
      int sts = measuredData.pop(buf1,384);
      if (sts && (strlen(buf1)>0))
      {
        sts = forwardValueToIotHub(buf1);
        if (!sts)
        {
          measuredData.push(buf1); // didn't go, push it back to try later
          Serial.println("Send to IoT hub failed");
        }
      }
    }

    // set RTC
    if (timeFromIotHub != "")
    {
      Serial.print("Set time: "); Serial.println(timeFromIotHub);
      String ttag = rtc.getTime("%Y/%m/%d,%H:%M:%S");
      Serial.println(ttag);
      timeFromIotHub = "";
    }

  }

}
```

The code is available on github.

Follow on tasks

For future additional functionality the following are under consideration:

- If data can not be sent to the WiFi IoT Edge Server for some reason (WiFi down, IoT Edge Server down) - save data in a backup log file and try to re-send it when things come back up.
- If we can't determine the date/time from NTP, try to get it from the IoT Edge Server - or implement an NTP server as part of the IoT Edge Server application

- Implment some "alarm" indicator - so that certain values could be used to trigger some immediate response - for example if a water leak sensor triggered or a smoke detector triggered, we might want to process that at a higher priority in the IoT Edge Hub.

History

Version 1.0, March, 2024

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