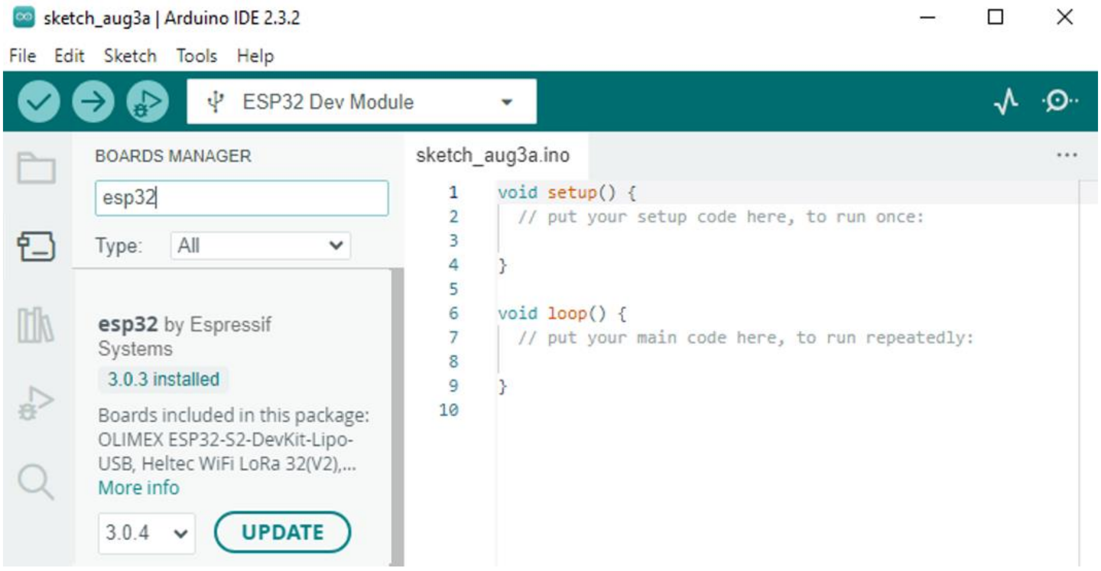
1. ARDUINO ESP32 CORE V3

The Espressif IoT (Internet of Things) Development Framework, *esp-idf*, for Espressif SoC (System on a Chip) supports ESP32 microcontrollers by providing Application Programming Interfaces (API). Example APIs are Bluetooth Low Energy, Wi-Fi and memory management. Details of the *esp-idf* are available at idf.espressif.com and at github.com/espressif/esp-idf. The current version for the ESP32 microcontroller is ESP-IDF 5.3.

The *esp-idf* supports the Arduino ESP32 core, which was migrated from versions 2.0.N, based on ESP-IDF 4.4, to version 3.0, based on ESP-IDF 5.1. Currently Arduino IDE version 3.0.4 and ESP-IDF 5.1.4 are available. Details of the Arduino ESP32 core and mapping to the latest ESP-IDF version are available at docs.espressif.com/projects/arduino-esp32/en/latest and github.com/espressif/arduino-esp32/releases, respectively. For the APIs included in "*ESP32 formats and communication*" the migration impacts the BLE, ESP-NOW, Hall sensor, I2S and LEDC APIs.

ESP32 API libraries are installed in the Arduino IDE by selecting the *Boards Manager* icon in the left side panel or selecting *Board> Boards Manager*, from the *Tools* menu, and entering *esp32* in the *Filter* option to display *esp32 by Espressif Systems* and clicking *Install* or *Update* (see Figure 1-1). The *esp32* files are located at *User\AppData\Local\Arduino15\packages\esp32\hardware\esp32\3.0.4.*



esp32

Type: All

esp32 by Espressif Systems

3.0.3 installed

Boards included in this package: OLIMEX ESP32-S2-DevKit-Lipo-USB, Heltec WiFi LoRa 32(V2),...

More info

3.0.4

UPDATE

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Figure 1-1 *Install esp32 Boards Manager*

BLE (Bluetooth Low Energy)

Following the migration to Arduino ESP32 core 3.0, the parameter type `std::string` is replaced with the Arduino style `String` and the `BLEScanResults` is changed to `BLEScanResults*`, which includes a pointer. Several Listings require changing `std::string` and Listing 5-10 requires the `BLEScanResults` update.

In the *setBeacon* function of Listings 5-2, 5-3, 5-4 and 5-11, replace `std::string` with `String`:

```
//  advertData.setServiceData(BLEUUID(UUID), std::string(beaconData, N));
    advertData.setServiceData(BLEUUID(UUID), String(beaconData, N));
```

in Listing 5-5,

```
//  std::string serviceData = "";
    String serviceData = "";
```

and in the `class RXCharCallback`, `void onWrite` function of Listing 5-8,

```
//  std::string str = RXChar->getValue();
    String str = RXChar->getValue();
```

Finally, in the *loop* function of Listing 5-10, include the two changes:

```
//  BLEScanResults res = BLEScan->start(scan, false);
    BLEScanResults *res = BLEScan->start(scan, false);
//  Serial.printf("%d Devices found\n", res.getCount());
    Serial.printf("%d Devices found\n", res->getCount());
```

ESP-NOW

Following the migration to Arduino ESP32 core 3.0, the `WiFi.mode(WIFI_STA)` instruction is required in the *setup* function. If the MAC address of the ESP32 microcontroller is required, then the `while(!WiFi.STA.started()) delay(100)` is also included in the *setup* function.

The `esp_now_register_recv_cb(receiveData)` instruction calls the *receiveData* function. The first parameter of the *receiveData* function of `const uint8_t * mac` , where *mac* is the MAC address of the transmitting ESP32 microcontroller, is replaced with `const esp_now_recv_info *info`. The info parameter includes the *src_addr*, *des_addr* and *rx_ctrl* structures, which are the MAC addresses of the transmitting or source and receiving or destination ESP32 microcontrollers. For example, the source MAC address is accessed with the instructions:

```
for (int i = 0; i < 6; i++)
{
  Serial.printf("%02x", info->src_addr[i])
  if (i < 5) Serial.print(":")
}
Serial.println("")
```

Alternatively, in the *receiveData* function, the source MAC address is copied to a byte array, defined by the `byte sendAdd[6]` instruction, with the `memcpy(sendAdd, info->src_addr, 6)` instruction and elements of the *sendAdd* array are subsequently accessed.

Information on the ESP-NOW packet is contained in the *rx_ctrl* structure with details of the *rx_ctrl* elements available at docs.espressif.com/projects/esp-idf/en/v4.4.2/esp32/api-reference/network/esp_wifi.html under the *struct wifi_pkt_rx_ctrl_t* heading. For example, the packet RSSI (Received Signal Strength Indicator) is obtained, within the *receiveData* function, with the instructions:

```
wifi_pkt_rx_ctrl_t * rx_ctrl = (wifi_pkt_rx_ctrl_t *)info
int RSSI = rx_ctrl->rssi
```

where *info* is defined in the `void receiveData(const esp_now_recv_info *info, const uint8_t * data, int len)` instruction.

The updated Listing 14-7, shown as Listing 1-1, incorporates the changes to ESP-NOW.

Listing 1-1 *Updated Listing 14-7*

```
#include <WiFi.h>                                     // include WiFi and
#include <esp_now.h>                                  //   ESP-NOW libraries
typedef struct
{
  int count;                                          // data structure with
  float value;                                        //   integer, real number
  char text[10];                                      //   and character array
} dataStruct;
dataStruct payload;
int rcv = 0;                                          // counter of received signals
int flag = 0;                                         // flag to display received data
byte sendAdd[6];                                      // array for source MAC address
int msglen, rssi, chan, cbw, noise;

void setup()
{
  Serial.begin(115200);                               // Serial Monitor baud rate
  WiFi.mode(WIFI_STA);
  if(esp_now_init() != 0)                             // initialise ESP-NOW
  {
    Serial.println("error initialising ESP-NOW");
    return;
  }
  esp_now_register_recv_cb(receiveData);              // call receiveData function
}
                                                // function to process received data
void receiveData(const esp_now_recv_info *info, const uint8_t * data, int len)
{
  memcpy(&payload, data, sizeof(payload));            // copy received data to payload
  msglen = len;
  wifi_pkt_rx_ctrl_t * rx_ctrl = (wifi_pkt_rx_ctrl_t *)info;
  rssi = rx_ctrl->rssi;                               // packet RSSI
  chan = rx_ctrl->channel;                            // channel number
  cbw = rx_ctrl->cwb;                                  // bandwidth 0 or 1
  noise = rx_ctrl->noise_floor;                        // packet noise floor
  memcpy(sendAdd, info->src_addr, 6);                  // copy source MAC address
  flag = 1;
}

void loop()
{
  if(flag > 0) displayData();
}

void displayData()                                     // function to display data
{
  rcv++;                                               // increment signal counter
  String str = cbw ? "40MHz" : "20MHz";                // define bandwidth
  Serial.printf("RXctrl RSSI %d chan %d bandw %d %s noise %d \n",
                rssi, chan, cbw, str, noise);          // display packet information
  for (int i = 0; i < 6; i++)
  {    Serial.printf("%02x", sendAdd[i]);              // display source MAC address
    if (i < 5) Serial.print(":");
  }                                                    // display payload content
  Serial.printf("\treceived %d \tbytes %d \tcount %d \tvalue %.2f \ttext %s \n",
                rcv, msglen, payload.count, payload.value, payload.text);
  flag = 0;
}
```

Hall sensor

The Hall sensor is no longer supported. Listing 13-4 must be compiled and loaded with Arduino ESP32 core version 2.0.17.

I2S

With Arduino ESP32 core version 3.0.4 installed, sketches including the *M5Core2* library or the *Audio* library by Wolle will not compile (see *ESP32 core versions* section). For an M5Stack Core2 module, the solution is to define the module with the *Board>M5Stack>M5Core2* option, which loads Arduino ESP32 core version 2.1.1. For an ESP32 microcontroller, the Arduino ESP32 core version 3.0.4 must be replaced with Arduino ESP32 core version 2.1.17 in the *Boards Manager* option.

In Chapter 2 *I2S Audio*, sketches including the *M5Core2* library are

2-1, 2-2 and 2-3 Display audio signal and FFT

2-5 Internet radio minimal sketch

2-6 and 2-7 Internet radio, parse text

2-9 and 2-10 Play MP3 files, SD card file structure

2-12 Bluetooth signal to M5Stack Core2

2-13 Bluetooth and M5Stack Core2 (colour squares)

2-15 Bluetooth signal from M5Stack Core2

Sketches for an ESP32 microcontroller, which include the *Audio* library, are:

2-4 Internet radio with an ESP32 module	accesses Audio library, use Arduino 2.0.17
2-8 MP3 files on an SD card and I2C decoder	accesses Audio library, use Arduino 2.0.17

Details of the I2S API are available at docs.espressif.com/projects/arduino-esp32/en/latest/api/i2s.html. Following development of the I2S API, Phil Schatzmann recommends use of the *AudioTools* library rather than the *ESP32-A2DP* library (see www.pschatzmann.ch/home/2024/04/07/esp32-a2dp-redesigning-the-i2s-output). The *ESP32-A2DP* library website (see github.com/pschatzmann/ESP32-A2DP) provides example sketches incorporating the *AudioTools* library, which is available at github.com/pschatzmann/arduino-audio-tools. The additional libraries *arduino-libhelix* (see github.com/pschatzmann/arduino-libhelix) and the *SdFat* library by Bill Greiman, available at github.com/greiman/SdFat, are also required. The *AudioTools* library must be installed from the downloaded *.zip* file and not through the Arduino IDE.Current library versions are: *ESP32-A2DP* (1.8.3) *AudioTools* (0.9.8), *libhelix* (0.8.5) and *SDFat* (2.2.3).

The *ESP32-A2DP* library example sketch of *bt_music_receiver_to_internal_dac,* for an ESP32 microcontroller, outputs the received audio data, as transmitted with the Bluetooth protocol, on an internal DAC (Digital to Analog Conversion) (see Listing 1-2). The *Huge APP (3MB No OTA/1MB SPIFFS)* Partition Scheme must be selected from the Arduino IDE *Tools* menu. The *BluetoothA2DPSink* library is a subset of the *ESP32-A2DP* library. A powered speaker is connected to GND and to one of the ESP32 microcontroller DAC output pins of GPIO 25 and 26.

***Listing 1-2** Bluetooth signal to an ESP32 microcontroller with DAC output*

```
#include <AudioTools.h>                   // include AudioTools and
#include <BluetoothA2DPSink.h>            //   BluetoothA2DPSink libraries
AnalogAudioStream BT;                     // DAC output
BluetoothA2DPSink a2dp_sink(BT);          // receive audio by Bluetooth

void setup()
{
  Serial.begin(115200);                   // Serial Monitor baud rate
  Serial.println("I2C and DAC");
  a2dp_sink.start("BTmusic");             // output audio on DAC
}

void loop()
{}
```

Sound quality is improved with a higher resolution DAC than the ESP32 microcontroller 8-bit DAC. The *ESP32-A2DP* library example sketch of *bt_music_receiver_simple*, for an ESP32 microcontroller, outputs received audio data to a PCM5102 I2S decoder module (see Listing 1-3). The *Huge APP (3MB No OTA/1MB SPIFFS)* Partition Scheme must be selected from the Arduino IDE *Tools* menu. Listing 1-3 replaces Listing 2-11 in *ESP32 Formats and Communication*. Listing 1-3 differs from Listing 1-2 by replacement of `AnalogAudioStream BT` with `I2SStream BT`.

Listing 1-3 *Bluetooth signal to an ESP32 microcontroller with PCM5102 decoder*

```
#include <AudioTools.h>
#include <BluetoothA2DPSink.h>
I2SStream BT;                                      // I2S output
BluetoothA2DPSink a2dp_sink(BT);

void setup()
{
  Serial.begin(115200);
  Serial.println("I2C and PCM5102");
  auto config = BT.defaultConfig();                // configure I2S output GPIO
  config.pin_bck = 27;                             // bit clock (SCK or BCLK)
  config.pin_ws = 25;                              // word select (WS or LRCK)
  config.pin_data = 26;                            // serial data (SD or SDOUT)
  BT.begin(config);
  a2dp_sink.start("BTmusic");
}

void loop()
{}
```

No change is required to Listing 2-14 in *ESP32 Formats and Communication* as the *AudioTools* library is already included in the sketch. Listing 2-14 is developed from the *examples-communication/a2dp/player-sdfat-a2dp* example sketch in the *AudioTools* library. Similarly, no change is required to Listing 2-16 in *ESP32 Formats and Communication* as the sketch scans for Bluetooth devices, which does not require the I2S API.

ledc

Details of the *ledc* API are available at docs.espressif.com/projects/arduino-esp32/en/latest/api/ledc.html.

Brightness of an LED was controlled by defining the GPIO pin connected to the LED, *LEDpin*, the square wave frequency, *freq*, the PWM resolution, *resol*, the scaled duty cycle, *scalduty*, and the output channel, *channel*, with the instructions:

```
int LEDpin = 4, freq = 5000, resol = 8, scalduty = 128, channel = 0;
ledcAttachPin(LEDpin, channel)
ledcSetup(channel, freq, resol)
ledcWrite(channel, scalduty)
```

where 8, 10 or 12-bit resolution corresponds to 2^8 = 256, 1024 or 4096 levels of LED brightness, the scaled duty cycle is the duty cycle percentage multiplied by 2^N-1, for resolution *N*, and a channel value between 0 and 15.

Following the migration to Arduino ESP32 core 3.0, the `ledcAttachPin` and `ledcSetup` instruction are combined to `ledcAttach(LEDpin, freq, resol)`, as a channel is automatically assigned. LED brightness is defined with the `ledcWrite(LEDpin, scalduty)` instruction, with the `ledcRead(LEDpin)` and `ledcReadFreq(LEDpin)` instructions returning the scaled duty cycle and frequency, respectively. The `ledcRead(LEDpin)` instruction has 10-bit resolution.

Similar to the Arduino `analogWrite` instruction, the brightness of an LED is controlled with the `analogWrite(LEDpin, scalduty)` instruction, which has 8-bit resolution and values between 0 and 255. The frequency must first be defined with the `analogWriteFrequency(LEDpin, freq)` instruction.

Square waves with 50% duty cycle are generated with the `ledcWriteTone(LEDpin, freq)` instruction. Square waves for set musical frequencies are generated with the `ledcWriteNote(LEDpin, note[i], octave)` instruction, with the *note* array defined as:

```
note_t note[] = {NOTE_C, NOTE_Cs, NOTE_D, NOTE_Eb, NOTE_E, NOTE_F,
                 NOTE_Fs, NOTE_G, NOTE_Gs, NOTE_A, NOTE_Bb, NOTE_B};
```

The default frequency of *note A* is 220Hz, which is the third octave. The underlying values of the *note_t* class for the *note* array are *{0, 1, 2, 3,...11}* with `NOTE_C` equal to *note_t* class value 0. The frequency of a note, which is *N* notes from a *baseline* note, is *baseline*Hz × $2^{N/12}$, such as `NOTE_C` = 220×$2^{3/12}$ = 262Hz. The maximum frequency of the `ledcWriteNote` instruction is 7902Hz, equal to `NOTE_B` and octave 8.

The sketch in Listing 1-4 illustrates the range of *ledc* instructions with separate functions for changing the duty cycle, changing the frequency and creating frequencies for musical notes. A delay is required between the `ledcWrite` and `ledcRead` instructions, with the `analogWriteFrequency` instruction preceding the `analogWrite` instruction. Note that the resolution of the `analogWrite`, `ledcWriteTone` and `ledcWriteNote` instructions is 8, 10 and 10-bit, respectively.

Listing 1-4 *ledc*

```
int LEDpin = 2;                                   // define LED GPIO
int freq = 5000, resol = 12;                      //   frequency and resolution
int steps = 20, val, bright, change = 10;
int lag = 1000;
float increm, duty;
note_t note[] = {NOTE_C, NOTE_Cs, NOTE_D, NOTE_Eb, NOTE_E, NOTE_F,
                 NOTE_Fs, NOTE_G, NOTE_Gs, NOTE_A, NOTE_Bb, NOTE_B};
int octave = 5;
```

```
void setup()
{
  Serial.begin(115200);                          // Serial Monitor baud rate
  ledcAttach(LEDpin, freq, resol);               // set frequency
  increm = 1.0 * (pow(2,resol)-1)/steps;         // incremental step size
  Serial.printf("increm %.1f \n", increm);
}

void loop()
{
  fn_ledcWrite();                                // call functions to
  fn_analogWrite();                              //   illustrate ledc
  fn_writeTone();                                //   instructions
  fn_writeNote();
}

 void fn_ledcWrite()                             // function to change duty cycle
 {                                               //   with ledcWrite
  Serial.println("\n     write read duty");
  for (int i=0; i<steps; i++)  // freq 5k
  {
    bright = i * increm;
    ledcWrite(LEDpin, bright);                   // scaled duty cycle
    delay(lag);
    val = ledcRead(LEDpin);                      // read LED scaled duty cycle
    duty = 100.0 * val/(pow(2, resol)-1);        // calculate duty cycle%
    Serial.printf("write %d %d %.1f%% \n", bright, val, duty);
  }
  delay(1000);
 }
void fn_analogWrite()                            // function to change duty cycle
{                                                //   with analogWrite
  analogWriteFrequency(LEDpin, freq/2);          // set frequency for analogWrite
  Serial.println("\n     write read duty");
  for (int i=255; i>0; i=i-change)
  {
    analogWrite(LEDpin, i);                      // 8-bit resolution
    delay(lag);
    val = ledcRead(LEDpin);
    duty = 100.0 * val/(pow(2, 8)-1);
    Serial.printf("analog %d %d %.1f%% \n", i, val, duty);
  }
  delay(1000);
 }

void fn_writeTone()                              // function to set frequency
{                                                //   ledcWriteTone
  ledcWriteTone(LEDpin, freq);                   // square wave 50% duty cycle
  delay(lag);
  val = ledcRead(LEDpin);
  duty = 100.0 * val/(pow(2, 10)-1);             // 10 bit resolution
  Serial.println("\n     duty");
  Serial.printf("tone %.1f%% \n", duty);
  delay(1000);
}

void fn_writeNote()                              // function for music frequencies
```

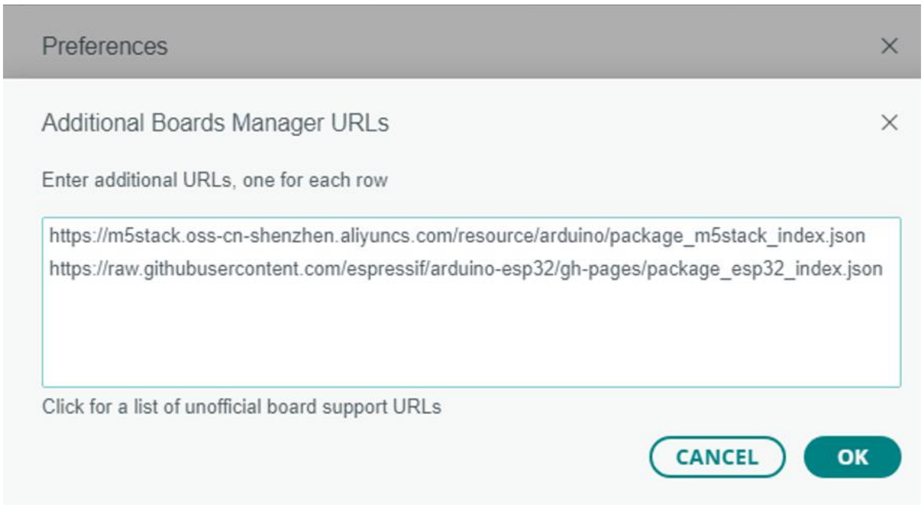
```
{                                                               //  with ledcWriteNote
  Serial.println("\n     duty freq");
  for (int i=0; i<12; i++)
  {
    ledcWriteNote(LEDpin, note[i], octave);
    delay(lag);
    duty = 100.0*ledcRead(LEDpin)/(pow(2, 10)-1);               // 10-bit resolution
    val = ledcReadFreq(LEDpin);
    Serial.printf("note %d %.1f%% %dHz \n", i, duty, val);
  }
  ledcChangeFrequency(LEDpin, freq, resol);                     // reset frequency
  delay(1000);
}
```

ESP32 core versions

In the Arduino IDE, select *File>Preferences* and in the *Additional Boards Manager URLs* box enter the URLs for the esp32 and M5Stack (see Figure 1-2):

https://m5stack.oss-cn-shenzhen.aliyuncs.com/resource/arduino/package_m5stack_index.json

https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json



Additional Boards Manager URLs

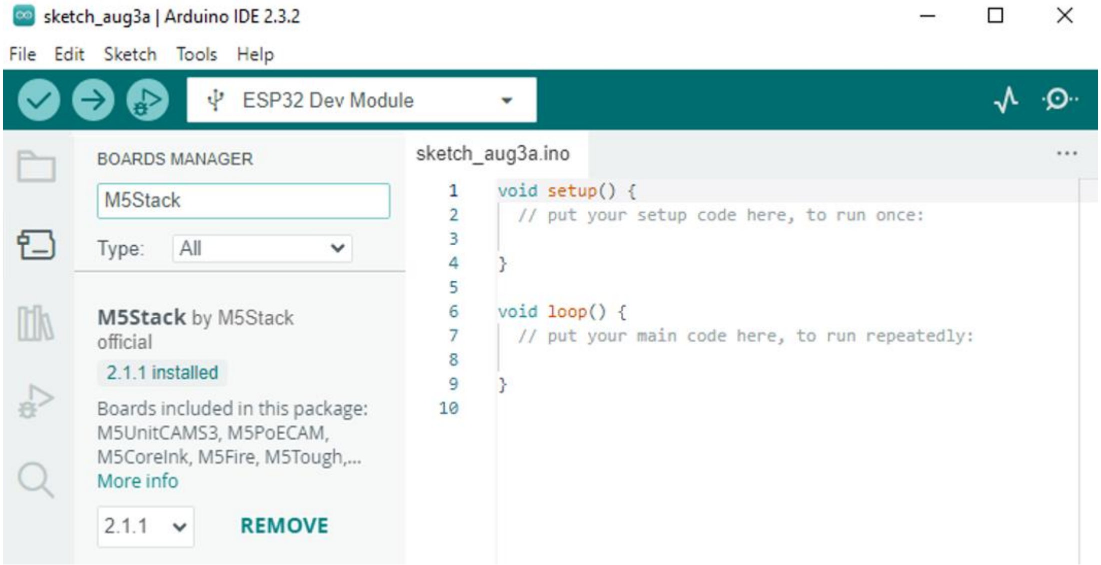
Enter additional URLs, one for each row

https://m5stack.oss-cn-shenzhen.aliyuncs.com/resource/arduino/package_m5stack_index.json
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json

Click for a list of unofficial board support URLs

Figure 1-2 *Arduino IDE Additional Boards Manager URLs*

The *esp32* and *M5Stack* libraries are installed in the Arduino IDE by selecting the *Boards Manager* icon in the left side panel or selecting *Board>Boards Manager*, from the *Tools* menu. In the *Filter* option, enter *esp32* and *M5Stack* to display *esp32 by Espressif Systems* and *M5Stack by M5Stack official*, respectively, then click *Install* or *Update* (see Figures 1-1 and 1-3). The *esp32* and *M5Stack* files are located at *User\AppData\Local\Arduino15\packages\esp32\hardware\esp32\3.0.4* and in *User \AppData\Local\Arduino15\packages\m5stack\hardware\esp32\2.1.1*. Note that the *Boards Manager* options of *esp32* and *M5Stack* correspond to the *esp32* library versions 3.0.4 and 2.1.1, respectively.



Search: M5Stack

Type: All

M5Stack by M5Stack official

2.1.1 installed

Boards included in this package: M5UnitCAMS3, M5PoECAM, M5CoreInk, M5Fire, M5Tough,...

Version: 2.1.1

```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

Figure 1-3 *Install M5Stack Boards Manager*

Prior to compiling and loading a sketch, an M5Stack Core2 module is defined in the Arduino IDE *Tools* menu by selecting *Board>esp32>M5Core2* or *Board>M5Stack>M5Core2*, which map the *m5stack_core2* board to either Arduino ESP32 core version 3.0.4 or 2.1.1, respectively. When the *Board>esp32>M5Core2* option is selected, the Arduino IDE output panel displays :

```
FQBN: esp32:esp32:m5stack_core2
Using board 'm5stack_core2' from platform in folder:
User\AppData\Local\Arduino15\packages\esp32\hardware\esp32\3.0.4
```

defining the FQBN (Fully Qualified Board Name) of an *m5stack_core2* board with an *esp32* core, which is Arduino ESP32 core 3.0.4. In contrast, when the *Board>M5Stack>M5Core2* option is selected, the Arduino ESP32 core 2.1.1 is loaded, as indicated in the Arduino IDE output panel:

```
FQBN: m5stack:esp32:m5stack_core2
Using board 'm5stack_core2' from platform in folder:
User\AppData\Local\Arduino15\packages\m5stack\hardware\esp32\2.1.1
```

Map to Arduino ESP32 core

Sketches including the *M5Core2* library will not compile with the Arduino ESP32 core version 3.0.4, so an M5Stack Core2 module must currently (August 2024) be defined with the *Board>M5Stack>M5Core2* option to load Arduino ESP32 core version 2.1.1. The compilation error message is:

```
User \Documents\Arduino\libraries\M5Core2\src/M5Display.h: In member function
'void M5Display::startWrite()':
User \Documents\Arduino\libraries\M5Core2\src/utility/In_eSPI.h:231:5: error:
'GPIO' was not declared in this scope
  231 |     GPIO.out_w1tc = (1 << TFT_CS); \
```

Audio library

Sketches including the *Audio* library by Wolle (schreibfaul1) will not compile with the Arduino ESP32 core version 3.0.4, which is based on ESP-IDF 5.1, as and internal DAC is not recognised with respect to I2S. Sketches with an ESP32 microcontroller must be compiled and loaded with Arduino ESP32 core version 2.0.17.