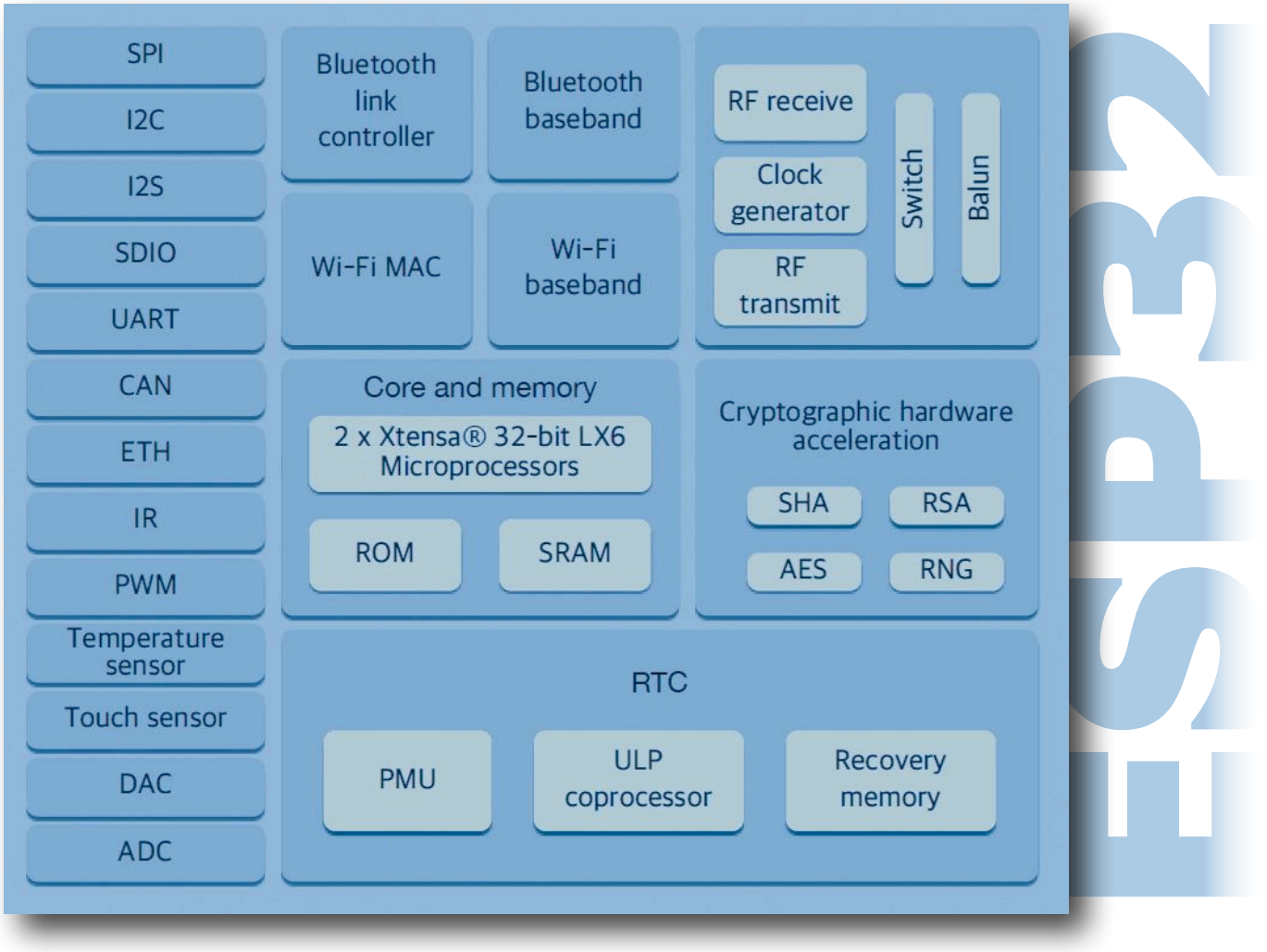
ESP32 for Power Users

Native programming

By **Tam Hanna** (Slovakia)

In the previous issue of Elektor we showed how easy it is to program the powerful ESP32 in the Arduino IDE [1]. However, if you want to utilize the full functionality of this microcontroller with integrated WLAN and Bluetooth, you have to use the native ESP IoT Development Framework (IDF). Working with the various command line tools can be intimidating for beginners or developers without a Linux background, so this article aims to show you how it's done.



Ubuntu is the preferred operating system at Espressif, and the actions described in this article are intended to be performed under Ubuntu 14.04 LTS. If you use Windows instead, you can obtain more information at [2]. And if you are an Apple user, you should consult [3].

The manufacturer provides the toolchain in the form of a binary package. Several auxiliary programs are necessary for using the toolchain; they can be downloaded and installed with the following command:

```
sudo apt-get install git wget make libncurses-dev
   flex bison gperf python python-serial
```

A helpful tip in this regard: `apt-get install` does not mind if some of the tools are already present on the target system

(see the parameters); they are simply skipped without any comment.
If you work with a 64-bit operating system, you should download the file at [4]. Those of you with a 32-bit system can find a suitable file at [5], although it is not very well supported.

If you download the file in Firefox, the content is automatically placed in the *Downloads* folder. Execute the following series of commands to extract the content to the *esp* subfolder and make it ready for use:

```
tamhan@TAMHAN14:~$ mkdir -p ~/esp

tamhan@TAMHAN14:~$ cd ~/esp

tamhan@TAMHAN14:~/esp$ tar -xzf ~/Downloads/xtensa-
   esp32-elf-linux64-1.22.0-61-gab8375a-5.2.0.tar.gz
```

For those of you with a Unix background, a brief explanation of the tilde character is appropriate here. It is a single symbol which represents the path to the home directory of the currently logged-in user, in order to avoid problems with typos.
The ESP32 toolchain expects the variable PATH to contain a particular directory. This can be achieved by entering the *export* command. You should bear in mind that this command is only effective as long as the current console window remains open:

```
tamhan@TAMHAN14:~$ export PATH=$PATH:$HOME/esp/
   xtensa-esp32-elf/bin
```

Last but not least, you have to download the support library from GitHub. Note that the following command must be issued in the home directory of the ESP toolchain:

```
tamhan@TAMHAN14:~/esp$ git clone --recursive https://
   github.com/espressif/esp-idf.git
```

Your first project

If you want to create a new project, you should first visit GitHub to see what is already available. The processor manufacturer provides a template which can be downloaded as follows:

```
tamhan@TAMHAN14:~/esp$ git clone https://github.com/
   espressif/esp-idf-template.git elektor1
```

The `clone` command takes as a parameter the name of the folder where the project structure should be set up.
For now we can ignore the various make files in that folder and concentrate on the file *main.c* located in a folder with the same name. Espressif equips it with a relatively complex skeleton, but we don't want to use that skeleton here. Instead, we want to output the sawtooth waveform described in the previous article, so we first have to delete the content of *main.c* (all code examples can be downloaded at [6]).

As usual, the first thing we have to do is to include several header files which provide the API. Here you can see that one of these header files points to the FreeRTOS real-time operating system, which Espressif uses in various places:

Make what?

The command line tool *make* has become more or less standard in the Unix world for automation of build processes (compilation, linking, etc.). The instruction files used to control the compilation process are called *make files*.

```
#include “freertos/FreeRTOS.h”
#include “esp_system.h”
#include “esp_event.h”
#include “esp_event_loop.h”
#include “nvs_flash.h”
#include “driver/gpio.h”
#include <driver/dac.h>
```

For the event loop of the RTOS we need an event handler, although in our case it always returns “OK” and does not affect execution of the code:

```
esp_err_t event_handler(void *ctx, system_event_t
   *event) {
    return ESP_OK;
}
```

That is followed directly by the function `app_main`, which is called when the microcontroller program starts. First it initializes the external flash memory and registers the event handler:

```
void app_main(void) {
    nvs_flash_init();
    ESP_ERROR_CHECK( esp_event_loop_init(event_
   handler, NULL) );
```

Next we can port the code used in the previous article of this series. The documentation at [7] shows that only one method is necessary:

```
    while (true) {
    for(int i=0;i<255;i++){
                dac_out_voltage(DAC_CHANNEL_1, i);
        }
    }
}
```

Find the board

In the Arduino IDE we would be finished at this point: just click and run, and let the digital storage scope do its job. But with the command line tools a bit of manual effort is necessary.
The first task is to find the ESP32 board. Unix puts the serial port of the ESP32 Thing module we are using (see [1]) somewhere under */dev*. To make it easier to find the target device, it is a good idea to check the content of the kernel log after plugging in the device. There is a sort of ring buffer which the Linux kernel populates with various information during system operation.

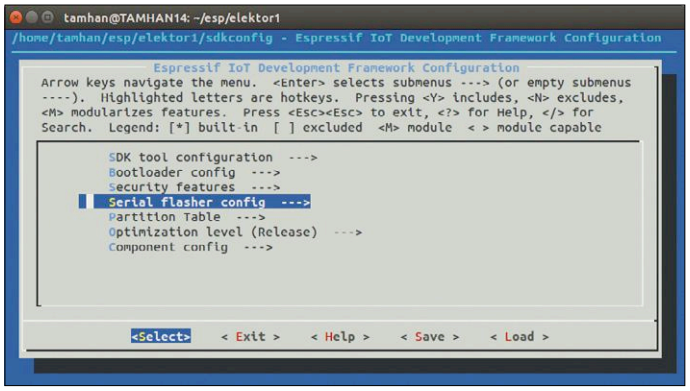


Figure 1. The start screen of *make menuconfig* is ready for entries.

The command shown here redirects the output of *dmesg* in the *grep* tool, which looks for the string located in the output of the FTDI driver and only displays the relevant lines:

```
tamhan@TAMHAN14:~/esp$ dmesg | grep 'FTDI USB Serial
   Device converter now attached'

[    4.817153] usb 1-1.6: FTDI USB Serial Device
   converter now attached to ttyUSB0
```

What matters here is the value which shows where the new device has been placed in the device tree. In our case this is *ttyUSB0*, so the path is */dev/ttyUSB0*.

Access to serial devices is normally limited to the superuser, so we need to give our Linux user account permission to access this port. For this we must first determine which user group the device belongs to. That can be done with a special variant of the `ls` command, which outputs additional information about a directory or a queried component:

```
tamhan@TAMHAN14:~/esp/nmgsample1$ ls -l /dev/ttyUSB0

crw-rw---- 1 root dialout 188, 0 feb 26 22:58 /dev/
   ttyUSB0
```

Under Unix this is usually called *dialout*. Next we have to add our user account to this group in order to obtain the access permissions:

```
root@TAMHAN14:~/esp/nmgsample1# sudo adduser tamhan
   dialout

Adding user `tamhan' to group `dialout' ...

Adding user tamhan to group dialout

Done.

root@TAMHAN14:~/esp/nmgsample1# sudo reboot
```

Unix updates the access permissions during a restart. If you want to avoid constantly prefixing your commands with `sudo`, you should restart your workstation at this point.

Menuconfig

Now let's look at the process of configuring the execution environment. For this we employ the frequently used *make* tool, which is responsible for processing compilation instructions. Since manual editing of make files is a tedious task, there is also a more or less standardized editing tool called *menuconfig*. *Menuconfig* is used not only in the ESP IDF, but also for compiling kernels and operating systems – for example, OpenWRT.
The *menuconfig* variant of the ESP IDF expects a variable named `IDF_PATH` which points to the directory containing the main part of the library. The call therefore looks like this:

```
tamhan@TAMHAN14:~/esp/elektor1$ export IDF_PATH=~/
   esp/esp-idf
```

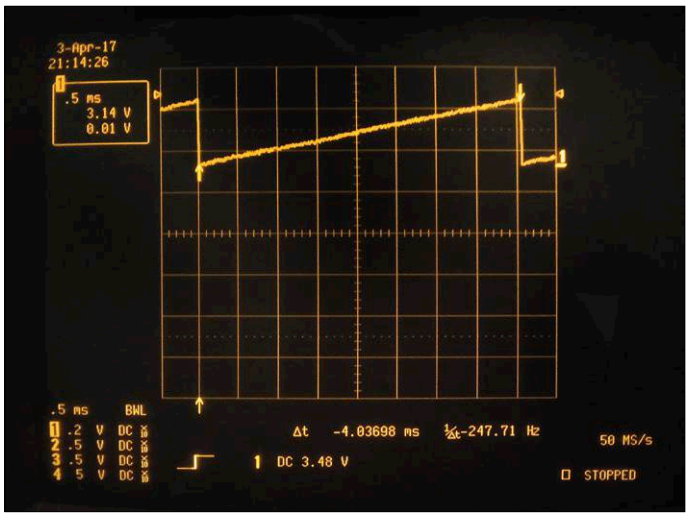


Figure 2. The ESP IDF is slower here than the Arduino IDE.

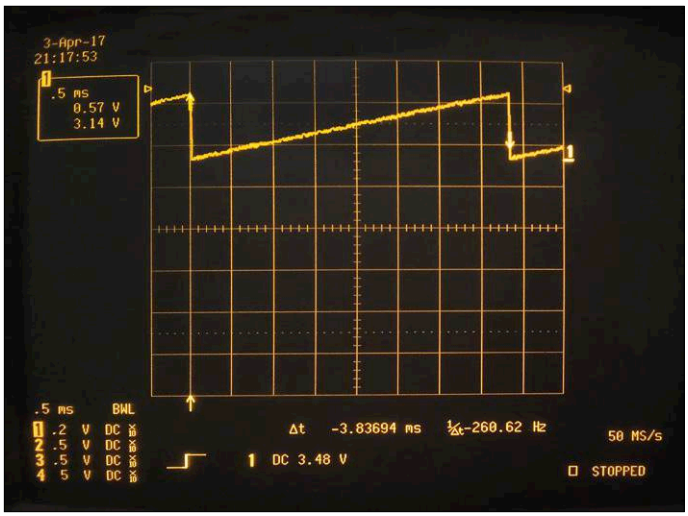


Figure 3. Changing to Release mode does not help very much.

```
tamhan@TAMHAN14:~/esp/elektor1$ make menuconfig
```

As in the previously mentioned case with the variable `PATH`, the *export* instruction is only valid as long as the terminal window to which it applies remains open.
Don't be surprised when compiler messages appear on the screen during the processing of `make menuconfig` – some parts of the tool are compiled directly before it is used. The actual user interaction is via the cursor keys, along with the Enter key to confirm the selected options (**Figure 1**).
The first important setting is located under *Serial flasher config → Default serial port*. There you should enter the previously determined port ID in order to link the tool chain to the ESP32 connected to the PC. Don't forget to save the configuration with the *Save* command after entering your settings.

After you are finished with the configuration, you have to download the firmware to the microcontroller. Here again you use `make` for this, but with different command parameters:

```
tamhan@TAMHAN14:~/esp/elektor1$ make flash

GENCONFIG

CC src/bootloader_flash.o

. . .

Wrote 16384 bytes at 0x00008000 in 1.4 seconds (91.0
   kbit/s)...

Hash of data verified.

Leaving...
Hard resetting...
```

Note that the output of the command depends on the specific operating state. The entire library must be fully processed during the first compilation, but not during subsequent compilations, so they run raster.

Figure 4. Disabling the FreeRTOS event loop also does not make the code significantly faster.

Running...

We connected a storage oscilloscope to pin 25 of the ESP32, yielding the result shown in **Figure 2** – which is not especially satisfactory. Apparently the native version of the code is significantly slower than the Arduino version described in the previous article.

We therefore went back to *menuconfig* to adjust various settings. First we set the *Optimization Level* parameter to *Release* and then deployed the program again after saving the change. The reward for this was approximately 10% more speed (**Figure 3**). This shows that a native API is not necessarily faster than the Arduino API.

The next thing we tried was to disable the event loop running in the background. To do so, we commented out a line in the `app_main` code:

```
void app_main(void) {
    nvs_flash_init();

    //ESP_ERROR_CHECK( esp_event_loop_init(event_
   handler, NULL) );
```

That also raised the speed, although the increase was only slight (see **Figure 4**). Despite all our efforts, we were not able to reach a speed level comparable to that attained with the Arduino sketch. The reason for this is the real-time operating system, which performs several time-consuming synchronization operations each time before it writes data to the registers responsible for the output.

Bluetooth

The next task is to establish a wireless link to a smartphone. The most important new feature of the ESP32 is the Bluetooth transceiver, which we now want to get up and running. First we had to activate the Bluetooth module of the microcontroller. To do so, we opened *make menuconfig* and went to the heading *Component Config*. We activated Bluetooth by pressing the Y key (an activated option is indicated by an asterisk between two square brackets ([*]). Then we saved the configuration. Several dozen additional files were compiled during the next compilation round.

Espressif relies on the Bluedroid stack. If you have a lot of previous experience with Android, you will probably be familiar with some of these methods. For the rest, we provide a brief introduction here. Before plunging into the details, we would like to make a general remark: When working with complex software systems (which definitely includes real-time operating systems such as FreeRTOS), it is very advisable to not write your own software completely from scratch (starting with a clean sheet). Instead, you should find a demo program or sample code which implements a similar function, examine its structure, and copy or adapt individual code segments or routines step by step.

The entry point of our program (without the error logging code) looks like this:

```
void app_main() {
    esp_err_t ret;
    esp_bt_controller_config_t bt_cfg =
   BT_CONTROLLER_INIT_CONFIG_DEFAULT();
    ret = esp_bt_controller_init(&bt_cfg);

    . . .
```

First we call the function `esp_bt_controller_init`, which is responsible for initializing the overall Bluetooth subsystem. The object to be configured is handled by a macro, which returns a standard implementation of the Bluetooth structure.

The next step is to define the operating mode of the Bluetooth microcontroller. Here we use "BTDM" in order to activate both Bluetooth LE and Bluetooth Classic. Next we call two housekeeping functions which allocate memory and processing time to the Bluetooth stack:

```
    ret = esp_bt_controller_enable(ESP_BT_MODE_BTDM);

    . . .

    ret = esp_bluedroid_init();

    . . .

    ret = esp_bluedroid_enable();

    . . .
```

The Bluetooth LE implementation of the ESP32 operates very asynchronously. We therefore register two event handlers which are responsible for the GATT and GAP events [8]. Finally, we register an additional application for the GATT protocol, which can be used later to hold attributes:

```
    esp_ble_gatts_register_callback(gatts_event_
   handler);
    esp_ble_gap_register_callback(gap_event_handler);

    esp_ble_gatts_app_register(0); //App-ID 0
    return;
}
```

In order to use the Bluetooth API, we must additionally include a group of headers. It's a mystery to the author why Espressif does not provide a catch-all file containing all of the necessary headers:

```
#include "esp_system.h"
#include "esp_log.h"
#include "nvs_flash.h"
#include "bt.h"
#include "bta_api.h"
#include "esp_gap_ble_api.h"
#include "esp_gatts_api.h"
#include "esp_bt_defs.h"
#include "esp_bt_main.h"
#include "esp_bt_main.h"
#include "sdkconfig.h"
```

Add some handlers

Bluetooth LE is intended to be a low-power communication system, but using synchronous methods and polling negates this advantage by increasing the load on the main processor. For this reason it is not surprising that the Bluetooth stack has a fully asynchronous structure and requires registration of event handlers.

The `gap_event_handler` looks after the events of the GAP protocol. Its main task is to send packets called "advertisements" which inform other hardware that the device concerned is present. Calling `esp_ble_gap_start_advertising` also instructs the stack to start a new advertising cycle:

```
static void gap_event_handler(esp_gap_ble_cb_event_t
   event, esp_ble_gap_cb_param_t *param)
{

    switch (event) {

    case ESP_GAP_BLE_ADV_DATA_SET_COMPLETE_EVT:
        esp_ble_gap_start_advertising(&test_adv_
   params);
        break;

    case ESP_GAP_BLE_ADV_DATA_RAW_SET_COMPLETE_EVT:
        esp_ble_gap_start_advertising(&test_adv_
   params);
        break;
```

The various parameters of the advertising command are supplied in the form of an `esp_ble_adv_params_t` structure, for which the settings can be copied directly from the template:

```
static esp_ble_adv_params_t test_adv_params = {
    .adv_int_min        = 0x20,
    .adv_int_max        = 0x40,
    .adv_type           = ADV_TYPE_IND,
    .own_addr_type      = BLE_ADDR_TYPE_PUBLIC,
    .channel_map        = ADV_CHNL_ALL,
    .adv_filter_policy =
   ADV_FILTER_ALLOW_SCAN_ANY_CON_ANY,
};
```

It's interesting to note that the stack requires prior inclusion of the file *string.h*, which is done as follows:

```
#include <string.h>
```

Next we have to declare memory areas where the Bluetooth stack can store temporary data. There is also a structure of type `esp_attr_value_t` which describes the attribute to be created:

```
#define GATTS_DEMO_CHAR_VAL_LEN_MAX 0x40

uint8_t char1_str[] = {0x11,0x22,0x33};

esp_attr_value_t gatts_demo_char1_val =
{
    .attr_max_len = GATTS_DEMO_CHAR_VAL_LEN_MAX,
    .attr_len     = sizeof(char1_str),
    .attr_value   = char1_str,
};
```

Bluetooth LE identifies devices and counterparts by their numeric IDs (UUIDs), which are globally unique and therefore typically very long. The Bluetooth API provides the type `esp_bt_uuid_t` to help developers cope with these UUIDs:

```
esp_gatt_srvc_id_t service_id;
uint16_t service_handle;
esp_bt_uuid_t descr_uuid;
esp_bt_uuid_t char_uuid;
```

If you have been paying close attention so far, you may be wondering why two characteristics are necessary for the implementation of our simple service. In addition to the actual characteristic, here we create a descriptor which provides information about the data contained in the characteristic. The actual event handler is then responsible for responding to the various events occurring in Bluetooth LE. We therefore restrict the complexity here to the implementation of a read-only characteristic, which leads to the following code:

```
static void gatts_event_handler(esp_gatts_cb_event_t
   event, esp_gatt_if_t gatts_if, esp_ble_gatts_cb_
   param_t *param)
{
    switch (event) {

    case ESP_GATTS_REG_EVT:
        ESP_LOGI(GATTS_TAG, “REGISTER_APP_EVT, status
   %d, app_id %d\n”, param->reg.status, param->reg.
   app_id);
        service_id.is_primary = true;
        service_id.id.inst_id = 0x00;
        service_id.id.uuid.len = ESP_UUID_LEN_16;
        service_id.id.uuid.uuid.uuid16 =
   GATTS_SERVICE_UUID_TEST_A;
        esp_ble_gap_set_device_name(“ElektorTest”);
```

Here `ESP_GATTS_REG_EVT` is responsible for registering a new characteristic with the stack; the Service ID parameter is populated during processing. `CREATE_EVT` is called when the body has been generated and is ready for configuration:

```
    case ESP_GATTS_CREATE_EVT:
        service_handle = param->create.
   service_handle;
        char_uuid.len = ESP_UUID_LEN_16;
        char_uuid.uuid.uuid16 = 0xFF01;
        esp_ble_gatts_start_service(service_
   handle);
        esp_ble_gatts_add_char(service_handle,
   &char_uuid,
                        ESP_GATT_PERM_READ |
   ESP_GATT_PERM_WRITE,
                        ESP_GATT_CHAR_PROP_BIT_
   READ |
                        ESP_GATT_CHAR_PROP_BIT_
   WRITE |
                        ESP_GATT_CHAR_PROP_BIT_
   NOTIFY,
                        &gatts_demo_char1_val,
   NULL);
       break;
```

Read requests addressed to the characteristic are answered by using the function `esp_ble_gatts_send_response`, which receives a bit string containing the data to be sent back to the requesting party.

```
    case ESP_GATTS_READ_EVT: {
        esp_gatt_rsp_t rsp;
        memset(&rsp, 0, sizeof(esp_gatt_rsp_t));
        rsp.attr_value.handle = param->read.
   handle;
        rsp.attr_value.len = 4;
        rsp.attr_value.value[0] = 0xde;
        rsp.attr_value.value[1] = 0xed;
        rsp.attr_value.value[2] = 0xbe;
        rsp.attr_value.value[3] = 0xef;
        esp_ble_gatts_send_response(gatts_if,
   param->read.conn_id, param->read.trans_id, ESP_
   GATT_OK, &rsp);
        }
        break;
```

`ESP_GATTS_ADD_CHAR_EVT` is responsible for the actual creation of

Is your Bluetooth library current?

Espressif has significantly revised their Bluetooth API several times in recent months. If the example described in this article does not work, you should update your IDF installation. The easiest way to do this is to delete the libraries and download them again. After downloading, don't forget to call `make menuconfig` again to update the configuration, and then save the new file.

More Bluetooth examples

Espressif offers numerous examples for the various Bluetooth operating modes. More information is available at [10].

the characteristic, which was previously instantiated by calling the method `esp_ble_gatts_add_char`:

```
        case ESP_GATTS_ADD_CHAR_EVT: {
            uint16_t length = 0;
            const uint8_t *prf_char;
            //gl_profile_tab[PROFILE_A_APP_ID].char_
   handle = param->add_char.attr_handle;
            descr_uuid.len = ESP_UUID_LEN_16;
            descr_uuid.uuid.uuid16 =
   ESP_GATT_UUID_CHAR_CLIENT_CONFIG;
            esp_ble_gatts_get_attr_value(param->add_
   char.attr_handle,  &length, &prf_char);
            esp_ble_gatts_add_char_descr(service_
   handle, &descr_uuid, ESP_GATT_PERM_READ | ESP_
   GATT_PERM_WRITE, NULL, NULL);
            }
            break;
```

Finally, we have to restart the advertising process after the link between the device and the data source is broken, in order to be visible to other potential clients:

```
        case ESP_GATTS_DISCONNECT_EVT:
            esp_ble_gap_start_advertising(&test_adv_
   params);
            break;

        default:
            break;
        }
}
```

There is not enough space here to go into the details of the interaction with the smartphone, but **Figure 5** shows how the created characteristic appears in the *Nordic BLE* app. The program, which is available for download in the Play Store at [9], acts as a sort of scanner which analyzes the contents of Bluetooth LE devices and enables interaction with them.

Summary

Even though the performance of the ESP IDF API cannot keep pace with the Arduino API in some areas, you have to work with the IDF if you want to take advantage of the full functionality of the ESP32. If you have previous experience with other 32-bit microcontrollers, you generally should not find it difficult to learn how to use the API. ⏮

(160457-I)

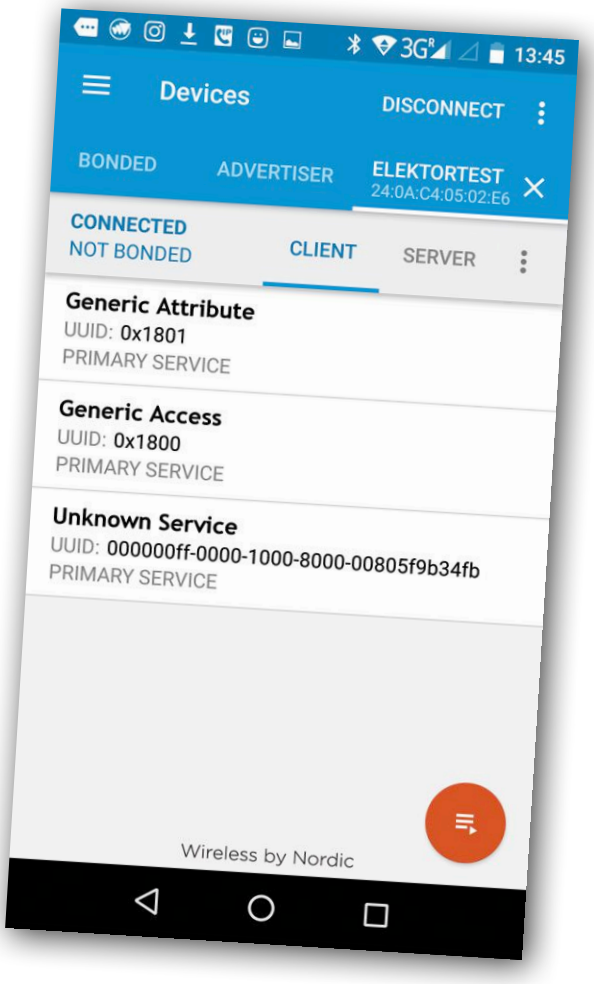


Figure 5. The Bluetooth characteristic created by the ESP32 is ready for access.

Web links

[1] www.elektormagazine.com/160454
[2] https://esp-idf.readthedocs.io/en/latest/get-started/windows-setup.html
[3] https://esp-idf.readthedocs.io/en/latest/get-started/macos-setup.html
[4] https://dl.espressif.com/dl/xtensa-esp32-elf-linux64-1.22.0-61-gab8375a-5.2.0.tar.gz
[5] https://dl.espressif.com/dl/xtensa-esp32-elf-linux32-1.22.0-61-gab8375a-5.2.0.tar.gz
[6] www.elektormagazine.com/160457
[7] http://esp-idf.readthedocs.io/en/latest/api-reference/peripherals/dac.html
[8] https://en.wikipedia.org/wiki/List_of_Bluetooth_profiles
[9] https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp&hl=en
[10] https://github.com/espressif/esp-idf/tree/07b61d5/examples/bluetooth