elektorMAG
design > share > earn
WK 01
* SINCE 1961 *
DEC. 2023 & JAN. 2024
ELEKTORMAGAZINE.COM

Special edition
guest-edited by

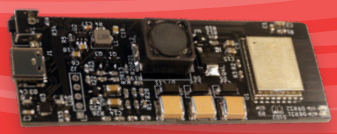
ESPRESSIF

Declassified Bonus Edition!

Home Automation With **Espressif Chips**

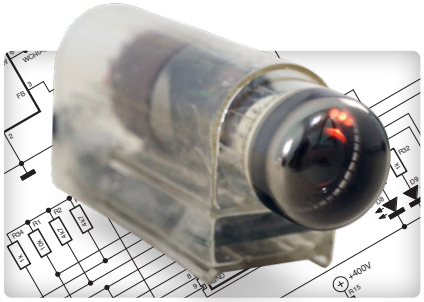
Try it with ESP Launchpad

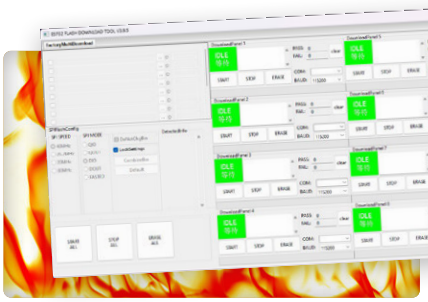
ADF, IDF, and Other **SDKs**



ESP32 and **ChatGPT**

Bonus articles for Pros, Makers, and Students!





INELTEK

Design-In Expertise & Service

Your Partner for ESPRESSIF

- **Ineltek** — Leading Espressif franchise distribution partner
- **Experience** — 36 years of experience in the semiconductor industry
- **Latest News** — Ineltek is always up to date regarding Espressif's innovations – Contact us!
- **Innovation** — Features are missing? We address your requests for next product generations
- **Application Support** — Dedicated and focused in-house support offering
- **Time to Market** — Espressif & Ineltek's FAE team are in close & regular contact to speed up your designs
- **Information** — We inform you on Espressif's products & wireless trends in trainings & webinars
- **Supply** — Popular Espressif SoCs, modules and EVKs available from stock

Contact us for Parts & Support

www.ineltek.com

Ineltek is the worldwide independent distributor with a **Passion for Innovation** and a **Commitment to Service**.

Founded in 1987, Ineltek gained the trust of thousands industrial customers as a technical semiconductor and design-in service provider. We work with your team to ensure our mutual success by providing the highest level of technical and commercial services for your projects.

Start your career with us!

Apply now at personal@ineltek.com

- Field Sales Engineer (m/f/x)
- Field Application Engineer (m/f/x)
- Line Management / Marketing (m/f/x)

Follow us!

INELTEK GmbH

Heidenheim, Wien, Castelfranco, London
Hamburg, München, Frankfurt, Dresden

COLOPHON

Volume 49, BONUS EDITION
December 2023 & January 2024
ISSN 0932-5468

Elektor Magazine is published 8 times a year by
Elektor International Media b.v.
PO Box 11, 6114 ZG Susteren, The Netherlands
Phone: +31 46 4389444
www.elektor.com | www.elektormagazine.com

For all your questions
service@elektor.com

Become a Member
www.elektormagazine.com/membership

Advertising & Sponsoring
Büsra Kas
Tel. +49 (0)241 95509178
busra.kas@elektor.com
www.elektormagazine.com/advertising

Print
Senefelder Misset, Mercuriusstraat 35,
7006 RK Doetinchem, The Netherlands

Distribution
IPS Group, Carl-Zeiss-Straße 5
53340 Meckenheim, Germany
Phone: +49 2225 88010

International Editor-in-Chief
Jens Nickel

Content Director
C. J. Abate

Publisher
Erik Jansen

EDITORIAL

Declassified Bonus Edition

Over the course of several months, Elektor's content team worked closely with Espressif to create in-depth articles about a variety of exciting topics. All the hard work resulted in an Espressif guest-edited edition of Elektor Mag, which we published in early December 2023. But, we're not finished collaborating. This bonus edition of Elektor Mag is packed with additional projects and tutorials that will keep you inspired for months to come.

In typical Elektor fashion, this bonus edition has something for everyone, from professional engineers interested in developing AIoT products to makers looking for creative weekend projects. Elektor and Espressif offer tips for implementing ChatGPT with Espressif SOCs, a fun Dekatron project, advice for building IoT apps without software expertise, and much more. Enjoy!

When you start your next Espressif-related project, please share your experience with the community on the Elektor Labs online platform at elektormagazine.com/labs — we look forward to seeing what you create!

C. J. Abate (Content Director, Elektor)

ESPRESSIF × elektor

THIS EDITION

Unleashing the Power of OpenAI and ESP-BOX

A Guide to Fusing ChatGPT with Espressif SOCs

By Ali Hassan Shah, Espressif

Let's explore the potential of ChatGPT with ESP-BOX, a development platform consisting of microphone-equipped boards and a vast software ecosystem for speech recognition and more. This forms a powerful combination that can take IoT devices to the next level!

The world is witnessing a technological revolution, and OpenAI is at the forefront of this change. One of its most exciting innovations is ChatGPT, which utilizes natural language processing to create more engaging and intuitive user experiences. The integration of OpenAI APIs with IoT devices has opened up a world of possibilities.

This article is structured into three primary sections, each covering essential aspects of the project. The first section delves into the ESP-BOX development platform, providing details on its features and functionalities. The second section is a case study that outlines the steps involved in building the project from scratch. The final section provides a list of additional sources of information that readers can explore to deepen their knowledge and understanding of the project.

ESP-BOX

The ESP-BOX [1] is a next-generation AIoT development platform that includes the ESP32-S3-BOX and ESP32-S3-BOX-Lite development boards. These boards are based on the ESP32-S3 [2] Wi-Fi + Bluetooth 5 (LE) SoC and provide a flexible and customizable solution for developing AIoT applications that integrate with various sensors, controllers, and gateways.

Figure 1: ESP32-S3-BOX.

The ESP-BOX (**Figure 1**) is packed with a wide range of features that make it an ideal AIoT development platform. Let's take a closer look at some of the key features.

- **Far-field voice interaction with 2 mics:** The ESP-BOX supports far-field voice interaction with two microphones, allowing users to interact with their devices from a distance.
- **Offline speech commands recognition in Chinese and English languages with high recognition rate:** The ESP-BOX offers offline speech commands recognition in both Chinese and English languages with a high recognition rate, making it easy to develop voice-enabled devices.
- **Reconfigurable 200+ speech commands in Chinese and English languages:** Developers can easily reconfigure over 200 speech commands in Chinese and English languages according to their needs.
- **Continuous Identification and Wakeup Interrupt:** The ESP-BOX supports continuous identification and wakeup interrupt, ensuring that devices are always ready to receive voice commands.
- **Flexible and reusable GUI framework:** The ESP-BOX comes with a flexible and reusable GUI framework, allowing developers to create custom user interfaces for their applications.
- **End-to-end AIoT development framework ESP-RainMaker:** The ESP-BOX is built on Espressif's end-to-end AIoT/IoT development framework, ESP-RainMaker, providing developers with all the tools they need to create powerful and intelligent devices.
- **Pmod compatible headers support expand peripheral modules:** The ESP-BOX features Pmod compatible headers, making it easy to expand its capabilities with a wide range of peripheral modules.

Case Study

This case study outlines the development process for a voice-controlled chatbot that utilizes the combination of ESP-BOX and OpenAI API. The system is capable of receiving speech commands from users, displaying them on the screen, and processing them through the OpenAI APIs to generate a response. The response is then displayed on the screen and played through the ESP-BOX. The step-by-step workflow provides a detailed explanation of how to integrate these technologies to create an efficient and effective voice-controlled chatbot (see **Figure 2** and **Figure 3**).

Environment Setup

Setting up a suitable environment and installing the correct version is critical to avoid errors. In this demonstration, we'll be utilizing ESP-IDF version 5.0 (master branch). If you need guidance on how to set up ESP-IDF, please refer to the official IDF Programming guide [3] for detailed information. As of writing of this article, the current IDF commit head is *df9310ada2*.

To utilize ChatGPT, a powerful language model based on the GPT-3.5 architecture, you must first obtain a secure API key. This can be achieved by creating an account on the OpenAI platform [4] and obtaining tokens through creation or purchase. With an API key, you gain access to a wide range of features and capabilities that can be

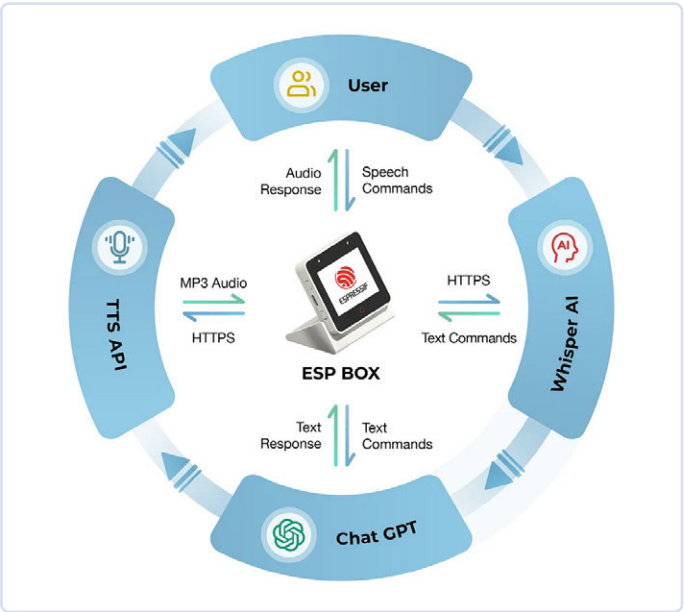


Figure 2: Workflow of the case study.

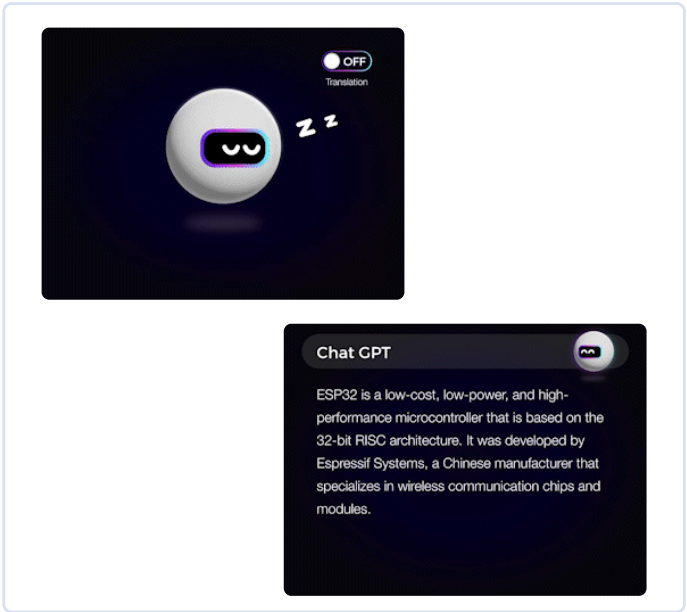


Figure 3: Demo of ESP-BOX as a chatbot.

customized to meet your specific needs, such as natural language processing and generation, text completion, and conversation modeling. Follow the official API reference link [5]. It goes without saying that maintaining the confidentiality and security of the API key is crucial to prevent unauthorized access to the user's account and data.

Adding Offline Speech Recognition

Espressif Systems has developed an innovative speech recognition framework called ESP-SR [6]. This framework is designed to enable devices to recognize spoken words and phrases without relying on external cloud-based services, making it an ideal solution for offline speech recognition applications.

ESP-SR framework consists of various modules, including the Audio Front-end (AFE), Wake Word Engine (WakeNet), Speech Command Word Recognition (MultiNet), and Speech Synthesis (which currently only supports the Chinese language). Follow the official Documentation [7] for more information.

Integrating OpenAI API

The OpenAI API provides numerous functions that developers can leverage to enhance their applications. In our project, we utilized the Audio-to-Text and Completion APIs and implemented them using C-language code based on ESP-IDF. The following section provides a brief overview of the code we employed.

Audio to Text

To extract text from audio, we utilize HTTPS and the OpenAI Audio API. The following code is used for this task. **(Listing 1)**

This code is a function named `create_whisper_request_from_record()`, which takes in a pointer to a buffer containing the audio data and an integer `audio_len` that represents the length of the audio data. This function sends a POST request to the OpenAI API endpoint to transcribe the given audio data.

The function starts by initializing the URL of the OpenAI API and setting the authorization headers with the bearer token `OPENAI_API_KEY`. Then, an HTTP client is configured and initialized with the provided configuration, including the URL, HTTP method, event handler, buffer size, timeout, and SSL certificate.

After that, the content type and the boundary string for the multipart form-data request are set as headers to the HTTP client. The file data and its size are also set, and a multipart/form-data request is built. The `form_data` buffer is allocated with a `malloc()` function, and the necessary information is added to it. This includes the filename and Content-Type of the audio file, the file contents, and the name of the model that will be used for transcription.

Once the `form_data` is built, it is set as the post field in the HTTP client, and the client sends the POST request to the OpenAI API endpoint. If there is an error during the request, the function logs an error message. Finally, the HTTP client is cleaned up, and the resources allocated for `form_data` are freed.

The function returns an `esp_err_t` error code, which indicates whether the HTTP request was successful or not.

Listing 1: Extracting Text from Audio.

```
esp_err_t create_whisper_request_from_record(uint8_t *audio, int audio_len)
{
      // Set the authorization headers
      char url[128] = "https://api.openai.com/v1/audio/transcriptions";
      char headers[256];
      snprintf(headers, sizeof(headers), "Bearer %s", OPENAI_API_KEY);
      // Configure the HTTP client
      esp_http_client_config_t config = {
         .url = url,
         .method = HTTP_METHOD_POST,
         .event_handler = response_handler,
         .buffer_size = MAX_HTTP_RECV_BUFFER,
         .timeout_ms = 60000,
         .crt_bundle_attach = esp_crt_bundle_attach,
      };
      // Initialize the HTTP client
      esp_http_client_handle_t client = esp_http_client_init(&config);
```

```
    // Set the headers
    esp_http_client_set_header(client, "Authorization", headers);
    // Set the content type and the boundary string
    char boundary[] = "boundary1234567890";
    char content_type[64];
    snprintf(content_type, sizeof(content_type), "multipart/form-data; boundary=%s", boundary);
    esp_http_client_set_header(client, "Content-Type", content_type);
    // Set the file data and size
    char *file_data = NULL;
    size_t file_size;
    file_data = (char *)audio;
    file_size = audio_len;
    // Build the multipart/form-data request
    char *form_data = (char *)malloc(MAX_HTTP_RECV_BUFFER);
    assert(form_data);
    ESP_LOGI(TAG, "Size of form_data buffer: %zu bytes", sizeof(*form_data) * MAX_HTTP_RECV_BUFFER);
    int form_data_len = 0;
    form_data_len += snprintf(form_data + form_data_len, MAX_HTTP_RECV_BUFFER - form_data_len,
                              "--%s\r\n"
                              "Content-Disposition: form-data; name=\"file\"; filename=\"%s\"\r\n"
                              "Content-Type: application/octet-stream\r\n"
                              "\r\n", boundary, get_file_format(file_type));
    ESP_LOGI(TAG, "form_data_len %d", form_data_len);
    ESP_LOGI(TAG, "form_data %s\n", form_data);
    // Append the audio file contents
    memcpy(form_data + form_data_len, file_data, file_size);
    form_data_len += file_size;
    ESP_LOGI(TAG, "Size of form_data: %zu", form_data_len);
    // Append the rest of the form-data
    form_data_len += snprintf(form_data + form_data_len, MAX_HTTP_RECV_BUFFER - form_data_len,
                              "\r\n"
                              "--%s\r\n"
                              "Content-Disposition: form-data; name=\"model\"\r\n"
                              "\r\n"
                              "whisper-1\r\n"
                              "--%s--\r\n", boundary, boundary);
    // Set the headers and post field
    esp_http_client_set_post_field(client, form_data, form_data_len);
    // Send the request
    esp_err_t err = esp_http_client_perform(client);
    if (err != ESP_OK) {
        ESP_LOGW(TAG, "HTTP POST request failed: %s\n", esp_err_to_name(err));
    }
    // Clean up client
    esp_http_client_cleanup(client);
    // Return error code
    return err;
}
```

The integration of OpenAI's ChatGPT with Espressif's ESP-BOX has opened up new possibilities for creating powerful and intelligent IoT devices.

Listing 2: HTTPS request for chat completion.

```
esp_err_t create_chatgpt_request(const char *content)
{
        char url[128] = "https://api.openai.com/v1/chat/completions";
        char model[16] = "gpt-3.5-turbo";
        char headers[256];
        snprintf(headers, sizeof(headers), "Bearer %s", OPENAI_API_KEY);
        esp_http_client_config_t config = {
         .url = url,
         .method = HTTP_METHOD_POST,
         .event_handler = response_handler,
         .buffer_size = MAX_HTTP_RECV_BUFFER,
         .timeout_ms = 30000,
         .cert_pem = esp_crt_bundle_attach,
        };
        // Set the headers
        esp_http_client_handle_t client = esp_http_client_init(&config);
        esp_http_client_set_header(client, "Content-Type", "application/json");
        esp_http_client_set_header(client, "Authorization", headers);
        // Create JSON payload with model, max tokens, and content
        snprintf(json_payload, sizeof(json_payload), json_fmt, model, MAX_RESPONSE_TOKEN, content);
        esp_http_client_set_post_field(client, json_payload, strlen(json_payload));
        // Send the request
        esp_err_t err = esp_http_client_perform(client);
        if (err != ESP_OK) {
          ESP_LOGW(TAG, "HTTP POST request failed: %s\n", esp_err_to_name(err));
        }
        // Clean up client
        esp_http_client_cleanup(client);
        // Return error code
        return err;
}
```

Chat Completion

The OpenAI Chat Completion API [8] is utilized to send HTTPS requests for chat completion. This process involves utilizing the `create_chatgpt_request()` function, which takes in a content parameter representing the input text to the GPT-3.5 model. **(Listing 2)**

The function first sets up the URL, model, and headers needed for the HTTP POST request, and then creates a JSON payload with the model, max tokens, and content. Next, the function sets the headers for the HTTP request and sets the JSON payload as the post field for the request. The HTTP POST request is then sent using `esp_http_client_perform()`, and if the request fails, an error message is logged. Finally, the HTTP client is cleaned up, and the error code is returned.

Handling Response

The callback function `response_handler()` is used by the ESP-IDF HTTP client library to handle events that occur during an HTTP request/response exchange. **(Listing 3)**

In case of `HTTP_EVENT_ON_DATA`, the function allocates memory for the incoming data, copies the data into the buffer and increments the `data_len` variable accordingly. This is done to accumulate the response data.

In case of `HTTP_EVENT_ON_FINISH`, the function prints a message indicating that the HTTP exchange has finished, and then calls the `parsing_data()` function to process the accumulated/raw data. It then frees the memory and resets the `data` and `data_len` variables to zero. Finally, the function returns `ESP_OK` indicating that the operation was successful.

Parsing Raw Data

The JSON `parser` component [9] is utilized to parse the raw response obtained from ChatGPT API and Whisper AI API over HTTPS. To perform this task, a function is used, which employs the parser component. Further details about this tool can be found on GitHub [10]. **(Listing 4)**

Integrating TTS API

At the moment, OpenAI doesn't offer public access to their Text-to-Speech (TTS) API. However, there are various other TTS APIs available, including Voicerss [11], TTSmaker [12], and TalkingGenie [13]. These APIs can generate speech from text input, and you can find more information about them on their respective websites.

For the purposes of this tutorial, we will be using the TalkingGenie API, which is one of the best options available for generating high-quality,

Listing 3: Handle events during an HTTP request/response exchange.

```
esp_err_t response_handler(esp_http_client_event_t *evt)
{
      static char *data = NULL; // Initialize data to NULL
      static int data_len = 0; // Initialize data to NULL
      switch (evt->event_id) {
      case HTTP_EVENT_ERROR:
          ESP_LOGI(TAG, "HTTP_EVENT_ERROR");
          break;
      case HTTP_EVENT_ON_CONNECTED:
          ESP_LOGI(TAG, "HTTP_EVENT_ON_CONNECTED");
          break;
      case HTTP_EVENT_HEADER_SENT:
          ESP_LOGI(TAG, "HTTP_EVENT_HEADER_SENT");
          break;
      case HTTP_EVENT_ON_HEADER:
          if (evt->data_len) {
              ESP_LOGI(TAG, "HTTP_EVENT_ON_HEADER");
              ESP_LOGI(TAG, "%.*s", evt->data_len, (char *)evt->data);
          }
          break;
      case HTTP_EVENT_ON_DATA:
          ESP_LOGI(TAG, "HTTP_EVENT_ON_DATA (%d +)%d\n", data_len, evt->data_len);
          ESP_LOGI(TAG, "Raw Response: data length: (%d +)%d: %.*s\n", data_len,
                  evt->data_len, evt->data_len, (char *)evt->data);

          // Allocate memory for the incoming data
          data = heap_caps_realloc(data, data_len + evt->data_len + 1,
                                   MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT);
          if (data == NULL) {
              ESP_LOGE(TAG, "data realloc failed");
              free(data);
              data = NULL;
              break;
          }
          memcpy(data + data_len, (char *)evt->data, evt->data_len);
          data_len += evt->data_len;
          data[data_len] = '\0';
          break;
      case HTTP_EVENT_ON_FINISH:
          ESP_LOGI(TAG, "HTTP_EVENT_ON_FINISH");
          if (data != NULL) {
              // Process the raw data
              parsing_data(data, strlen(data));
              // Free memory
              free(data);
              data = NULL;
              data_len = 0;
          }
          break;
      case HTTP_EVENT_DISCONNECTED:
          ESP_LOGI(TAG, "HTTP_EVENT_DISCONNECTED");
          break;
      default:
          break;
      }
      return ESP_OK;
}
```

Listing 4: Parsing the raw response obtained from ChatGPT API and Whisper AI API.

```
void parse_response (const char *data, int len)
{
      jparse_ctx_t jctx;
      int ret = json_parse_start(&jctx, data, len);
      if (ret != OS_SUCCESS) {
          ESP_LOGE(TAG, "Parser failed");
          return;
      }
      printf("\n");
      int num_choices;
      /* Parsing Chat GPT response*/
      if (json_obj_get_array(&jctx, "choices", &num_choices) == OS_SUCCESS) {
          for (int i = 0; i < num_choices; i++) {
              if (json_arr_get_object(&jctx, i) == OS_SUCCESS &&
                  json_obj_get_object(&jctx, "message") == OS_SUCCESS &&
                  json_obj_get_string(&jctx, "content", message_content,
                 sizeof(message_content)) == OS_SUCCESS) {
                      ESP_LOGI(TAG, "ChatGPT message_content: %s\n", message_content);
                  }
                  json_arr_leave_object(&jctx);
          }
          json_obj_leave_array(&jctx);
      }
      /* Parsing Whisper AI response*/
      else if (json_obj_get_string(&jctx, "text", message_content,
              sizeof(message_content)) == OS_SUCCESS) {
          ESP_LOGI(TAG, "Whisper message_content: %s\n", message_content);
      } else if (json_obj_get_object(&jctx, "error") == OS_SUCCESS) {
          if (json_obj_get_string(&jctx, "type", message_content,
              sizeof(message_content)) == OS_SUCCESS) {
                  ESP_LOGE(TAG, "API returns an error: %s", message_content);
          }
      }
}
```

natural-sounding speech both in English and Chinese. One of the unique features of TalkingGenie is its ability to translate mixed language text, such as Chinese and English, into speech seamlessly. This can be a valuable tool for creating content that appeals to a global audience. The following code sends a text response generated by ChatGPT to the TalkingGenie API using HTTPS, and then plays the resulting speech through an ESP-BOX. **(Listing 5)**

The function `text_to_speech()` takes a message string and an `AUDIO_CODECS_FORMAT` parameter as input. The message string is the text that will be synthesized into speech, while the `AUDIO_CODECS_FORMAT` parameter specifies whether the speech should be encoded in MP3 or WAV format.

The function first encodes the message string using `url_encode()` function that replaces some non-valid characters to its ASCII code, and then converts that code to a two-digit hexadecimal representation. Next, it allocates memory for the resulting encoded string. It then checks the `AUDIO_CODECS_FORMAT` parameter and sets the appropriate codec format string to be used in the `url`.

Next, the function determines the size of the `url` buffer needed to make a GET request to the TalkingGenie API, and allocates memory for the `url` buffer accordingly. It then formats the `url` string with the appropriate parameters, including the `voiceId` (which specifies the voice to be used), the encoded text, the speed and volume of the speech, and the audiotype (either MP3 or WAV).

The function then sets up an `esp_http_client_config_t` struct with the `url` and other configuration parameters, initializes an `esp_http_client_handle_t` with the struct, and performs a GET request to the TalkingGenie API using `esp_http_client_perform()`. If the request is successful, the function returns `ESP_OK`, otherwise it returns an error code. Finally, the function frees the memory allocated for the `url` buffer and the encoded message, cleans up the `esp_http_client_handle_t`, and returns the error code.

Handling TTS Response

In the similar fashion, callback function `http_event_handler()` is defined to handle events that occur during an HTTP request/response exchange. **(Listing 6)**

Listing 5: Text to Speech.

```
esp_err_t text_to_speech_request(const char *message, AUDIO_CODECS_FORMAT code_format)
{
      int j = 0;
      size_t message_len = strlen(message);
      char *encoded_message;
      char *language_format_str, *voice_format_str, *codec_format_str;
      // Encode the message for URL transmission
      encoded_message = heap_caps_malloc((3 * message_len + 1), MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT);
      url_encode(message, encoded_message);
      // Determine the audio codec format
      if (AUDIO_CODECS_MP3 == code_format) {
          codec_format_str = "MP3";
      } else {
          codec_format_str = "WAV";
      }
      // Determine the required size of the URL bu
      int url_size = snprintf(NULL, 0,
          "https://dds.dui.ai/runtime/v1/synthesize?voiceId=%s&text=%s&speed=1&volume=%d&audiotype=%s", \
                               VOICE_ID, \
                               encoded_message, \
                               VOLUME, \
                               codec_format_str);
      // Allocate memory for the URL buffer
      char *url = heap_caps_malloc((url_size + 1), MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT);
      if (url == NULL) {
          ESP_LOGE(TAG, "Failed to allocate memory for URL");
          return ESP_ERR_NO_MEM;
      }
      // Format the URL string
      snprintf(url, url_size + 1,
          "https://dds.dui.ai/runtime/v1/synthesize?voiceId=%s&text=%s&speed=1&volume=%d&audiotype=%s", \
              VOICE_ID, \
              encoded_message, \
              VOLUME, \
              codec_format_str);
      // Configure the HTTP client
      esp_http_client_config_t config = {
          .url = url,
          .method = HTTP_METHOD_GET,
          .event_handler = http_event_handler,
          .buffer_size = MAX_FILE_SIZE,
          .buffer_size_tx = 4000,
          .timeout_ms = 30000,
          .crt_bundle_attach = esp_crt_bundle_attach,
      };
      // Initialize and perform the HTTP request
      esp_http_client_handle_t client = esp_http_client_init(&config);
      esp_err_t err = esp_http_client_perform(client);
      if (err != ESP_OK) {
          ESP_LOGE(TAG, "HTTP GET request failed: %s", esp_err_to_name(err));
      }
      // Free allocated memory and clean up the HT
      heap_caps_free(url);
      heap_caps_free(encoded_message);
      esp_http_client_cleanup(client);
      // Return the result of the function call
      return err;
}
```

`HTTP_EVENT_ON_DATA` event is used to handle the audio data received from the server. The audio data is stored in a buffer called `record_audio_buffer` and the total length of the audio data received is stored in a variable called `file_total_len`. If the total length of the audio data received is less than a predefined `MAX_FILE_SIZE`, the data is copied into the `record_audio_buffer`.

Finally, the `HTTP_EVENT_ON_FINISH` event is used to handle the end of the HTTP response. In this case, the `record_audio_buffer` is passed to a function called `audio_player_play()`, which plays the audio.

Display

For display, we use LVGL, an open-source embedded graphics library that is gaining popularity for its powerful and visually appealing features and low memory footprints. LVGL has also released a visual drag-and-drop UI editor called SquareLine Studio [14]. It's a powerful tool that makes it easy to create beautiful GUIs for your applications.

To integrate LVGL with your project, Espressif Systems provides an official package manager tool [15]. This tool allows you to directly add LVGL and related porting components to your project, saving you time and effort. For more information follow the official blogs [16] and documentations [17].

Create Intelligent IoT Devices

The integration of OpenAI's ChatGPT with Espressif's ESP-BOX has opened up new possibilities for creating powerful and intelligent IoT devices. The ESP-BOX provides a flexible and customizable AIoT development platform with features like far-field voice interaction, offline speech commands recognition, and a reusable GUI framework. By combining these capabilities with the OpenAI API, developers can create voice-controlled chatbots and enhance user experiences in IoT applications.

Don't forget to check out Espressif [18] Systems' GitHub repository [19] for more open-source demos on ESP-IoT-Solution [20], ESP-SR, and ESP-BOX. The source code for this project will be found at GitHub [21]. As part of our future plans, we aim to introduce a component for the OpenAI API that will offer user-friendly functions.

230462-01

Questions or Comments?

If you have technical questions or comments about this article, feel free to contact the author at ali.shah@espressif.com or the Elektor editorial team by email at editor@elektor.com.

About the Author

Ali Hassan Shah is an embedded software engineer, driven by a deep passion for IoT technology. As an esteemed member of Espressif Systems's Application Engineering team, he channels his expertise into the mission of rendering technology effortlessly accessible and user-friendly for all, guided by the motto, "Simplifying Technology for All."

Related Products

- **ESP32-S3-BOX**
  www.elektor.com/20627

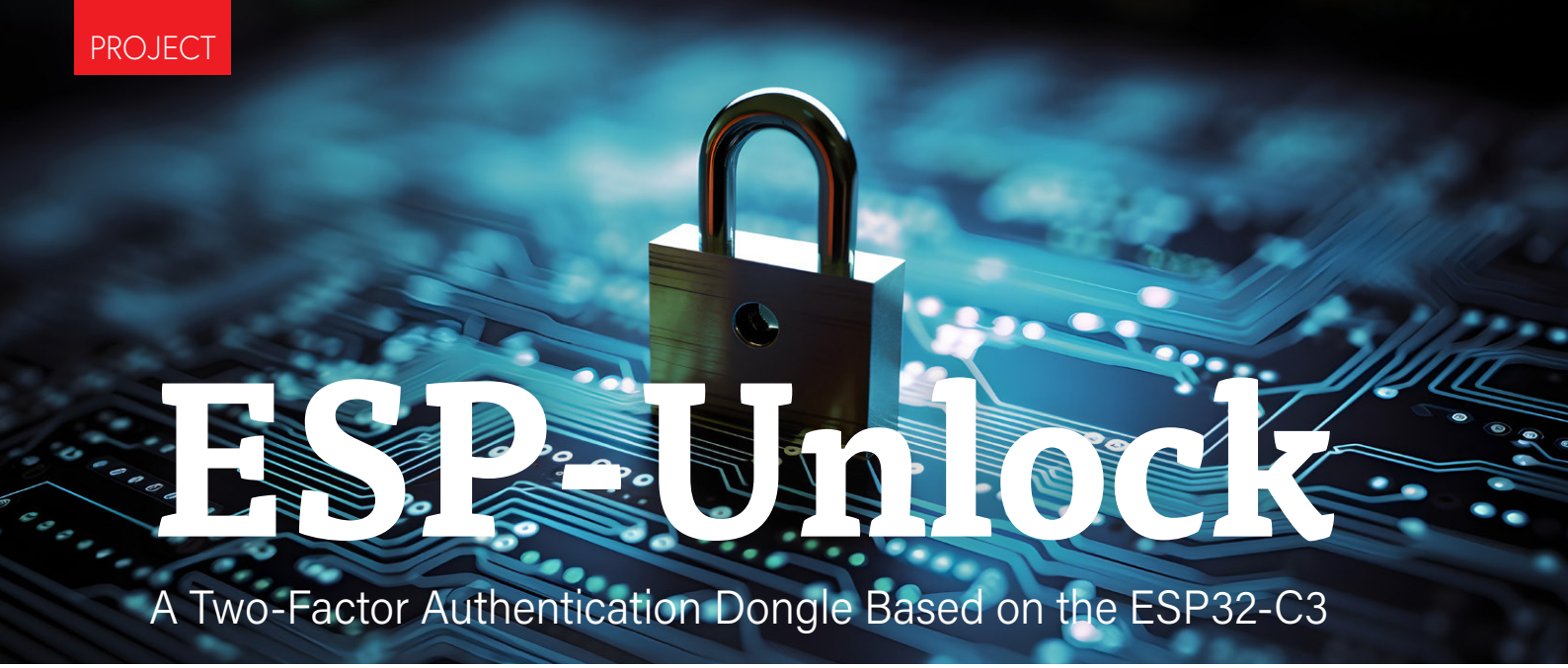
Listing 6: Handling TTS Response.

```
static esp_err_t http_event_handler(esp_http_client_event_t *evt)
{
      switch (evt->event_id) {
          // Handle errors that occur during the HTTP request
          case HTTP_EVENT_ERROR:
              ESP_LOGE(TAG, "HTTP_EVENT_ERROR");
              break;
      // Handle when the HTTP client is connected
      case HTTP_EVENT_ON_CONNECTED:
          ESP_LOGI(TAG, "HTTP_EVENT_ON_CONNECTED");
          break;
      // Handle when the header of the HTTP request is sent
      case HTTP_EVENT_HEADER_SENT:
          ESP_LOGI(TAG, "HTTP_EVENT_HEADER_SENT");
          break;
```

```
        // Handle when the header of the HTTP response is received
        case HTTP_EVENT_ON_HEADER:
            ESP_LOGI(TAG, "HTTP_EVENT_ON_HEADER");
            file_total_len = 0;
            break;
        // Handle when data is received in the HTTP response
        case HTTP_EVENT_ON_DATA:
            ESP_LOGI(TAG, "HTTP_EVENT_ON_DATA, len=%d", evt->data_len);
            if ((file_total_len + evt->data_len) < MAX_FILE_SIZE) {
                memcpy(record_audio_buffer + file_total_len, (char *)evt->data, evt->data_len);
                file_total_len += evt->data_len;
            }
            break;
        // Handle when the HTTP request finishes
        case HTTP_EVENT_ON_FINISH:
            ESP_LOGI(TAG, "HTTP_EVENT_ON_FINISH:%d, %d K", file_total_len, file_total_len / 1024);
            audio_player_play(record_audio_buffer, file_total_len);
            break;
        // Handle when the HTTP client is disconnected
        case HTTP_EVENT_DISCONNECTED:
            ESP_LOGI(TAG, "HTTP_EVENT_DISCONNECTED");
            break;
        // Handle when a redirection occurs in the HTTP request
        case HTTP_EVENT_REDIRECT:
            ESP_LOGI(TAG, "HTTP_EVENT_REDIRECT");
            break;
        }
        return ESP_OK;
}
```

WEB LINKS

[1] ESP-Box: https://github.com/espressif/esp-box
[2] ESP32-S3 product selector: https://tinyurl.com/esp32s3prodsel
[3] IDF Programming guide: https://docs.espressif.com/projects/esp-idf/en/release-v5.0/esp32/index.html
[4] OpenAI platform: https://openai.com
[5] Official API reference link: https://platform.openai.com/docs/api-reference
[6] ESP-SR: https://github.com/espressif/esp-sr
[7] ESP-SR User Guide: https://docs.espressif.com/projects/esp-sr/en/latest/esp32/index.html
[8] Chat Completion API: https://platform.openai.com/docs/api-reference/chat/create
[9] JSON parser: https://components.espressif.com/components/espressif/json_parser
[10] GitHub parser: https://github.com/espressif/json_parser
[11] Voicerss: https://voicerss.org/api
[12] TTSmaker: https://ttsmaker.com/zh-cn
[13] TalkingGenie: https://talkinggenie.com
[14] SquareLine Studio: https://squareline.io
[15] Official package manager tool for LVGL: https://components.espressif.com/components/lvgl/lvgl
[16] Blog about LVGL: https://tinyurl.com/espfancyui
[17] Documentation of LVGL: https://docs.lvgl.io/master/index.html
[18] Espressif Systems: https://espressif.com
[19] Espressif Systems' GitHub repository: https://github.com/orgs/espressif/repositories
[20] ESP-IoT-Solution: https://github.com/espressif/esp-iot-solution
[21] Source code for this project: https://github.com/espressif/esp-box/tree/master/examples



ESP-Unlock

A Two-Factor Authentication Dongle Based on the ESP32-C3

By Jakob Hasse, Espressif

Many online services nowadays provide two-factor authentication. The smartphone authenticator app is probably the most popular choice, but a smartphone can be stolen, and often private and business matters are mixed. This is where USB-stickshaped hardware tokens are useful. In this project, such a token is realized based on a cheap ESP32-C3 board.

Imagine you are a cybercriminal, and you have gained access to your victim's username and password for a random online service. Now you try to log in. It works! But then the online service asks you for a "verification code". Why? Because that online service provides two-factor authentication, and your victim has enabled it.

Two-factor authentication — common acronyms are 2FA, TFA, or MFA — basically means that you need an additional step to authenticate yourself against an online service. You not only authenticate yourself by something you *know*, but also by something you *possess*. In practice, this additional step often involves a hardware device that you have access to. A good example is withdrawing money from an ATM, where you need a PIN (knowledge) and your debit card (possession). Only knowing the PIN or possessing the card is not enough to access a victim's bank account.

But it is not only banks that use two-factor authentication. Many online services nowadays provide two-factor authentication, too. Unlike banks, they do not give out bank cards. Instead, you can use a smartphone authenticator app or an off-the-shelf electronic device, which we will call "hardware token" from now on. You would still need to remember your normal credentials to authenticate yourself (knowledge), but you would also be required to use your smartphone authenticator app or hardware token (possession).

The smartphone authenticator app is probably the most popular choice, since virtually everyone has a smartphone. But what about a backup if your phone gets stolen? How about separation of work accounts from your private phone? What if the phone ran out of power? Or simply, what if fishing your bathroom-tile-size flagship smartphone out of your pocket is too tedious?

This is where hardware tokens come in handy. They not only eliminate your dependency on a smartphone, but they can also be made small and cheap. A small hardware token can always be carried around in various situations. A cheap hardware token enables more people with a low budget to use it, and it also allows owning multiple hardware tokens for different purposes or to back each other up without the owner getting poor.

How Hardware Tokens Work

You might wonder: how can the online service distinguish your hardware token from any other token? The answer is that the hardware token is actually "smart" as it contains a small microprocessor and some storage. While setting up the hardware token as your online service authenticator, you obtain a cryptographic key from the online service, which is written to the storage (more details on that later). The key is then used to prove the hardware token's authenticity. You can imagine it as another password that the hardware token remembers to log into the online service. However, the way you use this kind of hardware token is very similar to using a physical key. If you use a smartphone authenticator app instead, it will save the key somewhere on your smartphone, but the principles explained here will be the same.

One standard for two-factor authentication is the widely used time-based one-time password (TOTP) standard [1]. TOTP is based on one-time Passwords (OTP), created from a cryptographic key and the current wall clock time, hence the name *time-based* one-time password. The key is shared between the online service and the hardware token. A hash function creates a numeric 6-digit OTP from the key and the current time. The online service does the same calculation with its own copy of the key and the current time. The OTP calculated by the hardware token is sent to the online service, which compares this OTP with

the OTP calculated from its own copy of the key. Only if the two OTPs match is the user is authenticated successfully. During the OTP calculation, a hash function is used to ensure that the original key is not compromised when transferring it from the hardware token to the online service. For more information on the TOTP standard, please refer to [1].

ESP-Unlock Hardware Token

Since two-factor authentication hardware tokens are so practical and the TOTP standard is widely used and easy to implement, I created the "ESP-Unlock:" An open-source TOTP-compatible two-factor authentication hardware token.

I chose the fairly inexpensive ESP32-C3 chip for the hardware token because I am familiar with the chip and the Espressif IoT Development Framework (ESP-IDF) [2], which can be used to program an ESP32-C3. To be more precise, I chose an ESP32-C3-WROOM-02U module that integrates the ESP32-C3 and other useful components.

The ESP32-C3 offers a few very useful features:

- An integrated USB-serial converter that enables communication over USB, an interface available on virtually every computer.
- A CPU with cryptographic accelerators to calculate the OTP.
- Hardware that enables flash encryption to encrypt the secret key and secure boot to prevent running unauthorized code on the hardware token.
- Flash memory to store program code and key is included on the ESP32-C3-WROOM-02U module.

The ESP-IDF development framework implements flash encryption and secure boot on top of the hardware as an easily accessible feature. The only missing parts were a PCB to put all the hardware together and software to connect the bits and pieces. The ESP-Unlock project combines all the parts, creates a simple and cheap open-source TOTP hardware token and an authenticator application for a host computer, which is discussed later.

We will now go over the architecture of the entire project, over the hardware and software, then discuss some considerations about security and finally see how to use the ESP-Unlock.

Project Architecture

To calculate and display the OTP, you need the hardware token itself, called ESP-Unlock, as well as a host computer that has a USB-A port. The ESP-Unlock stores the keys to generate OTPs for various online services. The host provides the current wall clock time via its real-time clock. Communication between the two is realized as serial communication over USB, using the USB-serial peripheral of the ESP32-C3.

Calculating an OTP needs four steps (see **Figure 1**):

1. The host computer acquires the current wall clock time, from its own real-time clock.
2. The host computer requests OTP calculation towards the ESP-Unlock.
3. The ESP-Unlock uses the appropriate key to calculate the OTP.
4. The ESP-Unlock sends the OTP back in a result message.

The communication (see steps 2 and 4 in Figure 1) between the host and ESP-Unlock consists of a fairly simple request-response protocol with the host always acting as initiator, while the ESP-Unlock only ever answers requests. All messages are sent as plaintext to ease debugging. A service name in the request message is necessary since the hardware token works with multiple online services, requiring one key per service. The example request in Figure 1 is:

```
TOTP:github,1690975870
```

where `TOTP:` and the newline character at the end are delimiters for parsing, `github` is the service name of the key to choose and `1690975870` is the current UNIX time, formatted as a decimal number. The corresponding response message in Figure 1 is:

```
TOTP:123456
```

where `TOTP:` and the newline are delimiters again and `123456` is the OTP. For more information about this protocol and the additional messages to list service names and add new sets of key and service names, visit the project's repository [3].

Hardware

As a small size was a paramount requirement for the hardware, the printed circuit board (PCB) for the ESP-Unlock only contains necessary components: an ESP32-C3-WROOM module (without antenna), corresponding bootstrap circuitry, power supply (3.3 V from the USB 5 V supply), two LEDs, and a USB-A connector to plug it into a computer. There is also a footprint for an

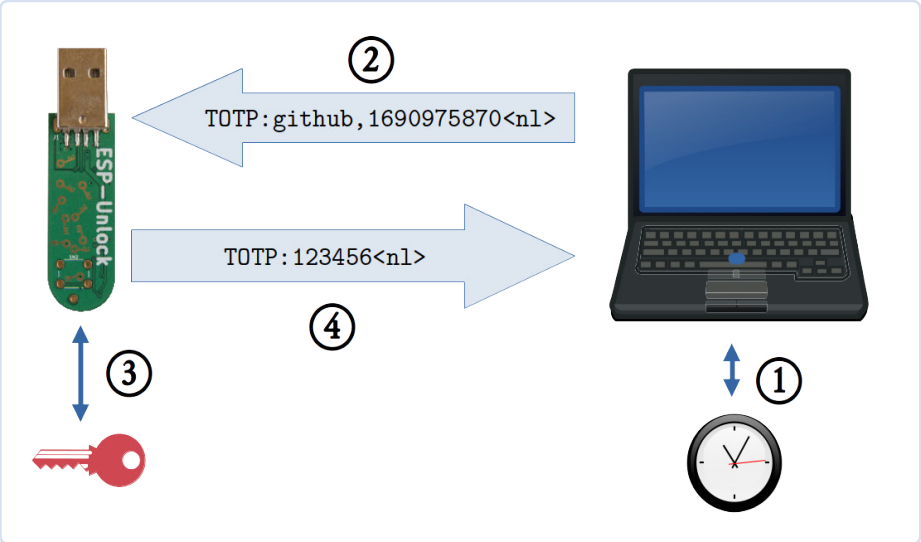


Figure 1: Communication between an ESP-Unlock hardware token and the host computer.

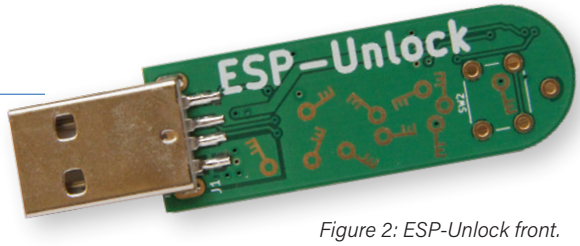


Figure 2: ESP-Unlock front.

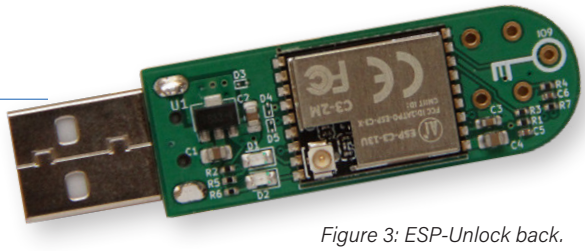


Figure 3: ESP-Unlock back.

optional button that can be used by future software versions to allow OTP calculations only if the button is pressed, thus increasing security. Including the USB connector, the resulting hardware measures 6×1.6 cm, roughly the size of a normal door key (**Figure 2** and **Figure 3**).

The hardware has been designed with KiCad, an open-source software capable of creating circuits and designing PCB layouts. The ESP-Unlock PCB layout has two-layers (**Figure 4**), because this is usually the cheapest option. All components except the through-hole components are located on one side, thus making all SMD components easy to solder using reflow-soldering.

Software

Two software applications are relevant for this project: a firmware running on the ESP-Unlock, making it "smart," and an application running on the host computer, called ESP-Unlock authenticator application.

The firmware running on the ESP-Unlock is a C++ application using ESP-IDF [2] and ESP-IDF-C++ [4]. As soon as the ESP-Unlock firmware boots, the application begins to listen for OTP requests. The two LEDs will blink once in an alternating pattern when the device is ready for requests. Once an OTP request has been received, the ESP-Unlock firmware checks if it has a key corresponding to the name in the request. If yes, it looks up the corresponding key. Then, it will calculate the OTP, using the just-read key and the time from the OTP request. A resulting response message containing the OTP is sent back to the host (compare Figure 1). Note that the entire OTP calculation is done on the ESP-Unlock — the key never leaves it.

The ESP-Unlock authenticator application is responsible for providing a simple user interface and for coordinating communication with the ESP-Unlock. The user interface lists all services to which keys are available on the currently connected ESP-Unlock, displays OTPs for corresponding services, and allows adding new keys. It is written for Linux only, but the ESP-Unlock firmware does not care about the host's operating system, so the application could be re-implemented for or ported to any other operating system.

Security Considerations

Please note that this is not a formal security analysis, which would be too much for the project at its current stage. It is meant to hint at dangers that exist and make these dangers comprehensible for the general reader.

The critical data on the ESP-Unlock is the keys, which should be kept secret all the time. Direct access to the key can be prevented by using flash encryption. But the software running on the ESP-Unlock has access to the keys, too. Hence, only authorized software should be running to prevent unauthorized software from leaking the keys. Secure Boot makes sure that only authorized code runs on the ESP32-C3. If Secure Boot and flash encryption are enabled, even an attacker who has physical access to the ESP-Unlock cannot read the key data.

Another factor is that the current software allows anyone with physical access to the ESP-Unlock to generate and read out OTPs. If you lose it and someone else finds it, that person can generate OTPs the same way as you do.

The firmware running on the ESP-Unlock might have exploits. In the worst case, such an exploit would allow an attacker to execute their own code on the ESP-Unlock. A mitigation that reduces some classes of exploits is enabling the stack protector in the firmware, which can be set in the firmware's configuration. Another mitigation would be a security-focused code review, which has not been done since the code is an early prototype. Note that secure boot does not protect against code exploits. Secure boot only protects against running unauthorized code on the hardware token. It does not protect from authorized code misbehaving and allowing random code execution. Exploiting the code is most likely with physical access to the ESP-Unlock. Attacks using software on the host computer as a proxy should also be considered, but are less likely due to the additional steps.

Considering these aspects, if an attacker gains physical access to an ESP-Unlock, the owner should set up a new ESP-Unlock and change the secret keys on all related online services as soon as possible. Note, however, that as long as the attacker does not obtain the user's credentials (username and password), they still will not be able to log in. After all, the OTP is merely the second factor, and you should still be as secure as without any two-factor authentication.

Using the ESP-Unlock

Before the ESP-Unlock can generate OTPs, the ESP-Unlock authenticator application has to be installed and the key for the two-factor authentication has to be added. For more information about the installation of the ESP-Unlock authenticator app, please refer to the instructions in its repository [5]. To add your key, you first need to retrieve it from the compatible online service you want to use. How that works depends on the specific online service, but they normally guide you through

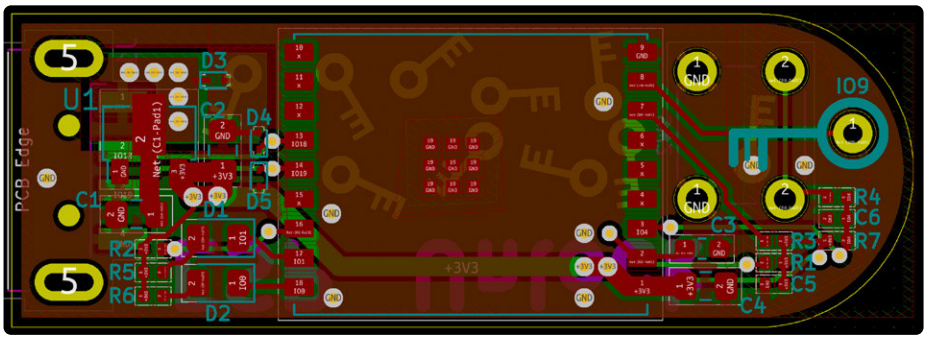


Figure 4: Top, bottom, and silkscreen layers of the PCB.

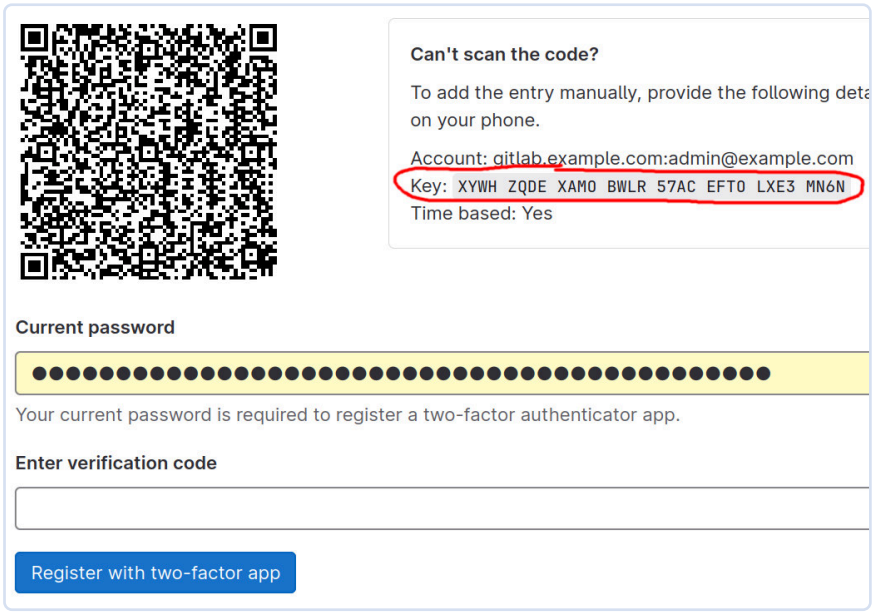


Figure 5: Base32-encoded key in GitLab (marked red).

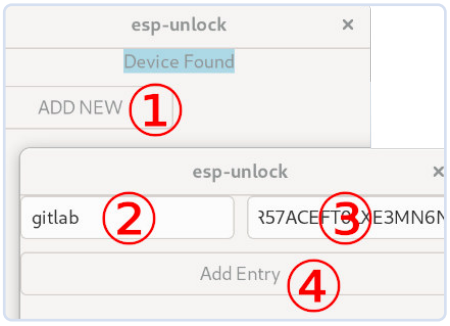


Figure 6: Add a new key.

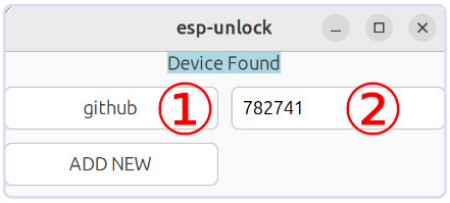


Figure 7: Request OTP.

the process of setting up your two-factor authentication device or application. They usually display the key as a QR code, which is convenient when using a smartphone authenticator app, but for the ESP-Unlock, the text form, formatted as base32, is required. For example, GitLab provides the base32-encoded key on the setup page for two-factor authentication besides the QR code (**Figure 5**).

Once the ESP-Unlock is plugged into the host computer, two-factor authentication for a new service can be set up in the authenticator application by following these steps (compare **Figure 6**):

1. Click on *ADD NEW*.
2. In the window that pops up, choose a service name and enter it.
3. Enter the base32-formatted key itself. Note that you need to remove any spaces or the key on the ESP-Unlock will be incorrect.
4. Click on *Add Entry*.

Whenever you need an OTP from now on, you insert the ESP-Unlock into the host's USB port and start the ESP-Unlock authenticator application, which will display the names of all keys saved on the ESP-Unlock and on a button click (step 1 in **Figure 7**) request and display the corresponding OTP (step 2 in Figure 7).

If you think this project is interesting, feel free to take a look or try it and visit the ESP-Unlock repository [3] and its corresponding ESP-Unlock authenticator application repository [5]. Schematics and PCB layout can be found in the hardware repository [6], but the ESP-Unlock hardware is not necessary! You can indeed use any ESP32-series development board, given a slightly changed configuration (in this case, please also refer to the ESP-Unlock repository's *README.md* [3]). Any contributions to the project, such as suggestions, hints, improvements, and bug reports, are very welcome!

230559-01

Questions or Comments?

If you have technical questions or comments about this article, feel free to contact the author via jakob.hasse@mailbox.org or the Elektor editorial team at editor@elektor.com.

About the Author

Jakob Hasse has a degree in Computer Science, and has various different interests. While studying, he worked at the German Aerospace Center in the Robotics field. Later, he developed a passion for embedded systems and Open-Source software. After starting as an embedded Linux systems engineer, Jakob joined Espressif to work on their FreeRTOS-based Espressif IoT Development Framework. Besides that, he is interested in IT security, which is the main driver for the ESP-Unlock project.

Related Products

- **ESP32-C3-WROOM-02U**
  www.elektor.com/20695
- **Peter Dalmaris, *KiCad 6 Like A Pro* (Bundle of two books)**
  www.elektor.com/20180

WEB LINKS

[1] Time-based One Time Password (TOTP) standard: https://ietf.org/rfc/rfc6238.txt
[2] Espressif IoT Development Framework (ESP-IDF): https://github.com/espressif/esp-idf
[3] The project's repository: https://github.com/0xjakob/esp-unlock
[4] ESP-IDF-C++: https://github.com/espressif/esp-idf-cxx
[5] The ESP-Unlock authenticator app: https://github.com/0xjakob/esp-unlock-host-gui
[6] Hardware repository: https://github.com/0xjakob/esp-unlock-hardware

Dekatron

A Piece of History Comes Back to Life!

By Jeroen Domburg, Espressif

Exactly as is the case with music, where ancient and modern styles often blend to achieve extraordinary results, the same can be verified in electronics: In this article, a very modern ESP32-C3 module effectively drives a Dekatron, a decadic vacuum tube counter from the 1950s — bringing it back to life after 70 years!

Unlike some venerable institutes like Elektor, Espressif doesn't really have a history that goes back very far. This is probably for the best: We doubt that there would have been a large market for IoT WiFi chips back in 1961. Instead, Espressif was set up in 2008, to release their first chips a good five years later: the ESP8089 and the ESP8266. While the ESP8089 was mostly an OEM chip, designed to give Wi-Fi capability to Android tablets, for example, the ESP8266 is an actual IoT WiFi chip. The fact that even nowadays, people are still starting new projects with the ESP8266 illustrates how little of this all can be considered "retro." (Although, if you're thinking of starting a project with an ESP8266 chip yourself, we'd ask you to consider using e.g. an ESP32-C3 instead; as it turns out, half a decade of innovation makes that a much nicer chip for about the same price.)

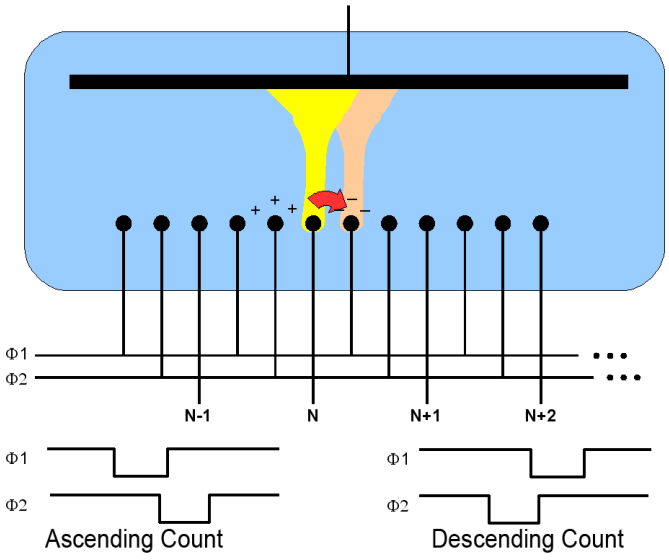


Figure 1: Basic working principles of a Dekatron. (Source: Wikimedia Commons.)

As such, I decided that it might make for a more interesting story to see if I could interface between the old and the new. In this particular case, I'm going to marry a more than 50-year-old Dekatron to the relative newcomer that is the ESP32-C3 and use that as an indicator of how much of our internet bandwidth we're using. Surely, that wasn't a use anyone had in mind when the Dekatron was built, but it is a good use nevertheless.

The Dekatron Tube

In case you're unfamiliar with antique electronic goods, a Dekatron is a gas-filled glass tube. (Note that this is unlike a vacuum tube, which is typically filled with a marked absence of any gas.) The gas is usually an inert gas such as neon, but gases like hydrogen were also used. If the mention of "neon" reminds you of the small neon lights you used to see in on/off buttons of old devices, you're not far off: They make use of similar principles.

A Dekatron is effectively a counter with a visual indication of a count. It works as such: The top of a Dekatron contains a round anode. It has 30 cathodes in the form of pins evenly spaced around it, as you can see in the explanatory drawing of **Figure 1** and in the real-life snapshots of **Figure 2**. Ten of these are so-called output cathodes, the other twenty are guide cathodes, which are divided into two groups: G1 and G2 (Φ1 and Φ2, respectively, in Figure 1). Each output cathode will have a G1 guide cathode, G1 on one side and a G2 on the other. All G1 cathodes are paralleled, as are all G2 cathodes.

When started, a current-limited 400 V is applied between the anode and the output cathodes. As this is more than the ionization voltage of the gas, the gas will ionize and glow at one of these cathodes. As this lowers the anode voltage to below the ionization voltage, no other

electrode will glow. This glow can be "transferred" to the adjacent output cathodes by grounding G1 and G2 in sequence — for instance, if G1 is grounded and then G2 is grounded, the ionization will travel clockwise, while reversing the G1/G2 sequence makes the ionization travel counter-clockwise.

What's the use of all these ionization shenanigans? Well, it makes these bits of glass, metal and gas capable of counting the G1/G2 pulses; specifically, it can count up to ten of those before wrapping around (hence the "deka" in "Dekatron"). Usually, some or all of the output cathodes are led to individual pins on the base of the Dekatron, so certain counts can be detected. This way, they can be used as pulse counters and frequency dividers, and also as memory elements: Famously, a 1951 computer uses Dekatrons as such [1]. (Note the present tense in the previous sentence: The computer is still running as a museum exhibit!)

Obviously, Dekatrons have been obsoleted multiple times over, initially by counters built out of discrete transistors. Then, as ICs took over the more complicated tasks, chips such as the venerable CD4017 decade counter picked up the baton. Finally, nowadays in the time of cheap microcontrollers and FPGAs, the function of the Dekatron is usually achieved by but a few lines of code or HDL.

One downside of all these newfangled counting techniques, however, is that unlike the Dekatron, they won't provide any visual output by themselves. Sure, you can connect LEDs to your CD4017 (like I suspect many readers have done in their past), or put a spinner on that LCD that you connected to your fancy new microcontroller, but there's something to the neon glow of the Dekatron that cannot really be replaced with pixels or light-emitting semiconductors.

Supply Design Challenges

Therein also lies a bit of a problem: the ESP32-C3 is ideally suited to lighting up those pixels or LEDs, but not so much to interface with 70-year-old glassware: For all the good it does to the Dekatron, the 3.3 V the ESP32-C3 can send to its IO pins might as well not exist. In other words, aside from finding a way to generate the 400 V, we also need to switch several high-ish voltages.

Let's start with the 400 V. The easiest way to get this is to use a voltage doubler on the voltage from the mains. However, for this product, I didn't really want to use mains voltage; the safety precautions needed when dealing with that would be a bit too much. I decided instead that a modern USB-C connector was the way to go; between laptop power bricks, cellphone adapters, and that random charger you got with some electronic device (but you can't for the life of you remember which one), there're enough power supplies with that connector to go around. Additionally, nowadays USB-PD is a common standard, and this allows you to request higher voltages than the 5 V normally available on a USB port. This happened to come in handy in this case.

So how to generate that 400 V? There are multiple ways to do this. The most straightforward of them involve transformers: For instance, you can convert your low voltage to AC, dump it into the low-voltage winding of a transformer, and get high-voltage out of the other end. Another way is to use it as a flyback converter: Charge the magnetic field by applying a voltage to the low-voltage side, remove the voltage and have the collapsing magnetic field be picked up by the high-voltage side, and there's your output. The issue with both these solutions is that miniature transformers tend to be a fairly niche product, and this runs the risk of the design not being replicable when the chosen one goes out of stock. While I don't expect this design will be made by the millions, I'd like it to work and be repairable for a long time, so non-standard components aren't really something I want here.

The Solution

So instead, I'm going with another option: a boost converter. A boost converter is generally used to convert voltages to slightly higher voltages. You might, for example, use one if you want to run some 5 V logic from two AA batteries. They have the downside that boosting voltages by a lot is somewhat problematic — specifically, the switching element that switches the low voltage also needs to stand up to the high output voltage, and the maximum factor you can boost your input voltage by is limited by things like the DC resistance of the inductor you use.

A trick to get around this is to use a voltage doubler after the boost converter; this way, we only need to boost to half the 400 V required. (Note that this trick is taken from a tried-and-true Dekatron boost converter circuit [2])

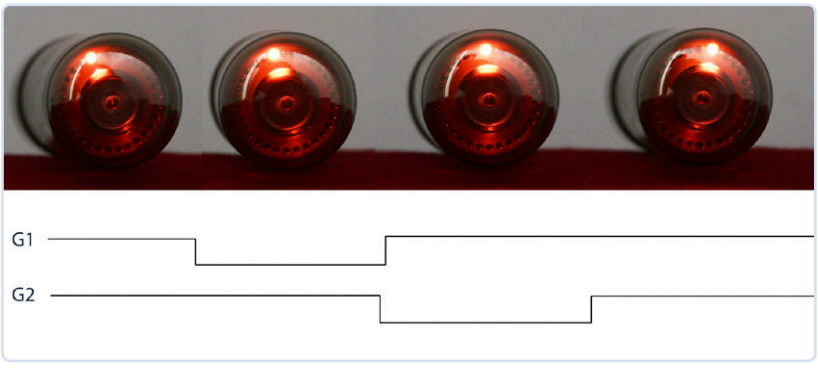


Figure 2: The Dekatron in action in a FWD count, with the related signals on G1 and G2.

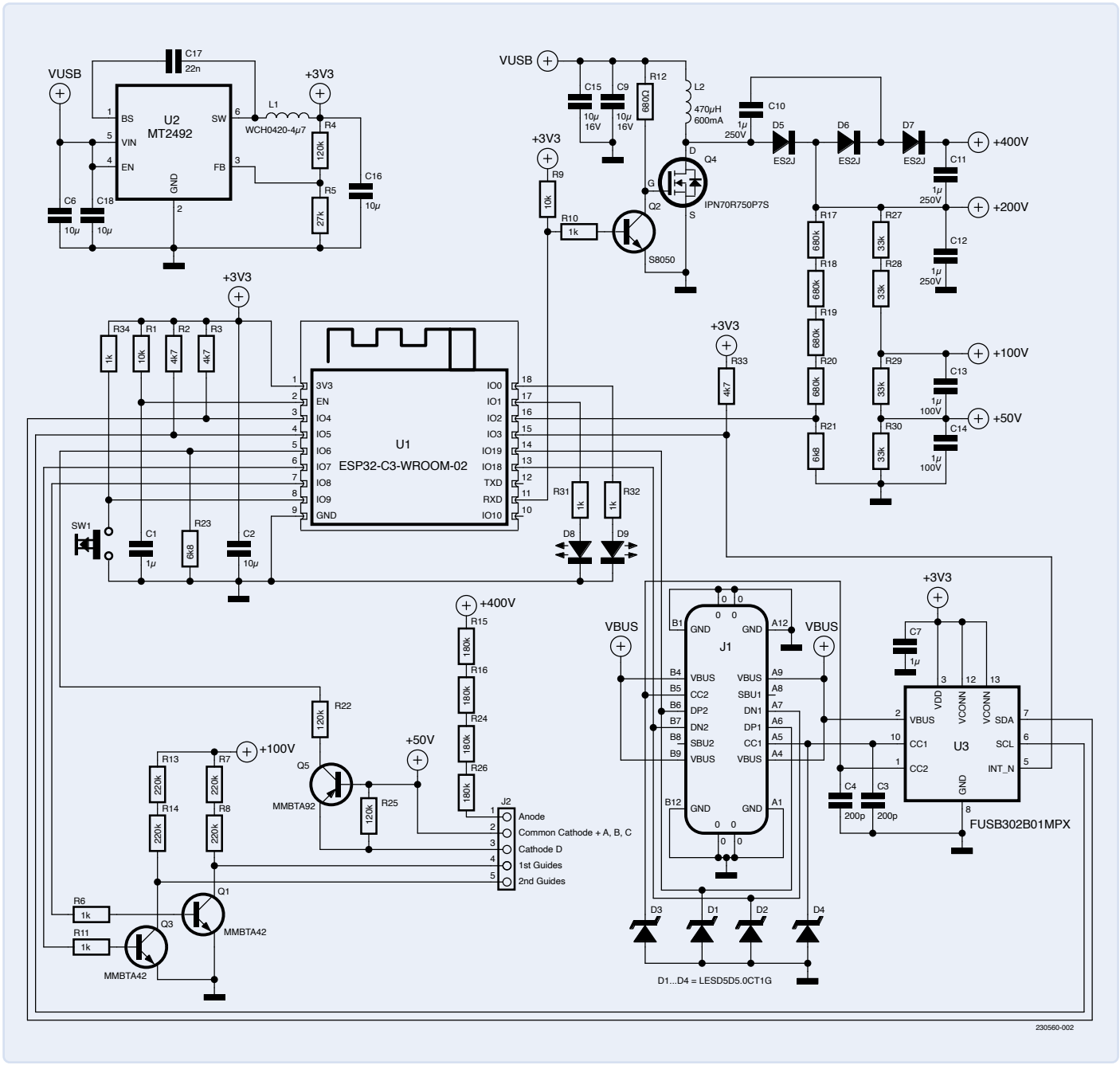


Figure 3: Schematics of this project.

The other trick is that we can start with a higher voltage: Generally, PD-compatible USB-C chargers can deliver 5 V but also 9 V, 15 V, or 20 V, if you request it. This means that the boost converter doesn't have to work so hard to get the input voltage all the way up. (Another solution would be to use a second boost converter to get the supply voltage raised a bit before increasing it all the way to 400 V. Full disclosure: I picked the USB-PD route as I wanted to play with USB-PD anyway.)

The implementation details of this are in **Figure 3**. The voltage from the USB port, J1, is stabilized using two ceramic caps first. The main boost logic is formed by L2 and Q4: When Q4 is in conduction, it'll build up a magnetic field, and when it's open, the current from the collapsing field will be dumped into C11 and C12 via the rectification and voltage doubler formed by D5 to D7. This results in two DC voltages, one of 200 V and one of 400 V.

Q4 is special; as stated before, it specifically needs to be able to stand up to the high voltages L2 generates. Unfortunately, MOSFETS that can do this do not tend to be logic-level ones, so Q2 is there to convert the incoming 3.3 V PWM signal into something that makes it conduct. That PWM signal is generated by the ESP32-C3, and the duty cycle is regulated by measuring the 200 V rail and adjusting accordingly. This is done by dividing it down into something the internal ADC can digest, using R17 to R21.

Aside from 400 V, we also need some lower voltages, and these are generated by dividing the 200 V down using some resistors. Note that the schematic tends to use

multiple resistors in series where only one would suffice: This is because I wanted to use 0603 parts, and these only stand up to 75 V or so. Putting them in series means a lower voltage drop on each one, meaning no sudden sparks: While HV arcing would be an epic indicator of a top-rate download, it lacks in replicability and would smell up the room.

On the Dekatron side, as illustrated in the schematics in Figure 3, the 400 V is fed into the anode via a 720 kΩ resistance. According to the datasheet [3], the anode current needs to be limited to about 310 µA, and this series of resistors takes care of that. The G1 and G2 signals are level-shifted by two high-voltage NPN transistors, and one of the non-shared cathodes is brought to a PNP transistor level shifter in order to make the ESP32-C3 aware if the glow passes this cathode.

The ESP32-C3 Module

The ESP32-C3 itself is visible in the middle of the schematics of Figure 3. As I'm using an ESP32-C3-WROOM02 module, most of the needed support hardware is included: flash, RF matching networks, and the antenna are all there. The only things on the outside are two indicator LEDs, a push button and an RC network to generate a power-on reset. If you're using KiCad for your PCB design, like I am, note that Espressif maintains a free-to-use repository of symbols and footprints for all the modules and chips [4].

Finally, In the top and right part of the diagram, the power section is shown. The USB-C connector has two purposes: If a USB-PD supply is attached, it powers everything, but you can also connect it to your computer. In that case, the Dekatron will not power up, as the 5 V supply that is available is not enough for that, but it does allow reprogramming the code in the ESP32-C3 flash. In order to do the USB-PD negotiation, a FUSB302 USB-C controller is used; the ESP32-C3 can talk to this over I²C to have it communicate with the power supply. In order to convert whatever voltage comes in to something the ESP32-C3 can run on, I used a synchronous buck converter, namely the Aerosemi MT2492. This little chip works at up to 16 V, and the fact that it's a switcher means it stays nice and cool while generating the 3.3 V rail that the ESP32-C3 works on.

PCB Layout

Figure 4 shows the PCB design. I chose a form factor that is as wide as the Dekatron itself, so it can "hide" underneath. Routing this PCB actually is a bit trickier than it would appear: The high voltages meant I needed to keep creepage in mind, or traces might arc over, and, generally, the high voltage requires physically larger components. I still managed to squirrel everything away on a two-layer board, even with a ground plane that is mostly intact. I decided to have the PCB fabbed in a nice black solder mask so that it wouldn't stand out and steal the show from the Dekatron, as you can see in **Figure 5.**

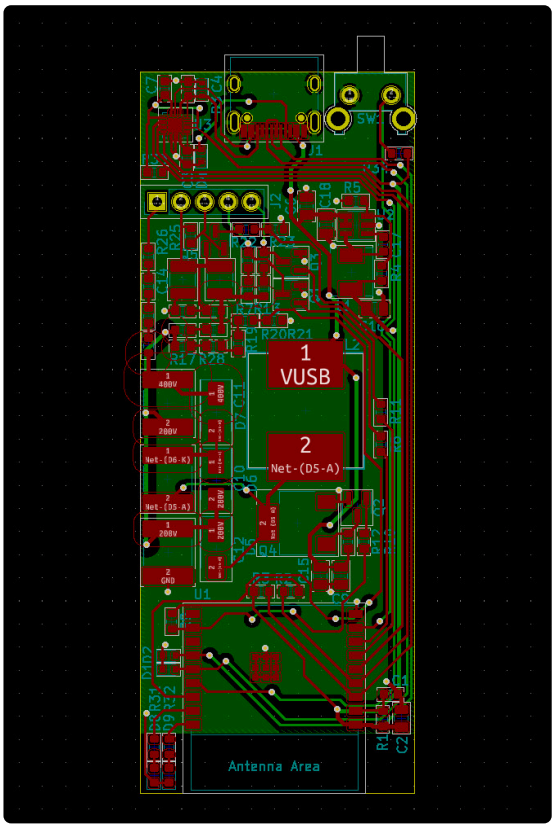


Figure 4: PCB layout of the ESP32-C3-based Dekatron interface.

Software Implementation

With the hardware built, I started coding. The software needs to do a few things. First of all, it needs to make sure the rest of the hardware gets the proper voltage, and, to do that, it needs to implement a USB-PD stack to talk to the attached power supply. I've used an existing library for that that works well [5]. The hardware itself can use a fairly wide voltage range, so the software will try to request 12 V from the power supply, but if that fails it'll accept either 9 V or 15 V as well. To also get the Dekatron its high voltage, it measures what the current voltage on the 200 V rail is and adjusts the PWM duty cycle to the boost converter accordingly.

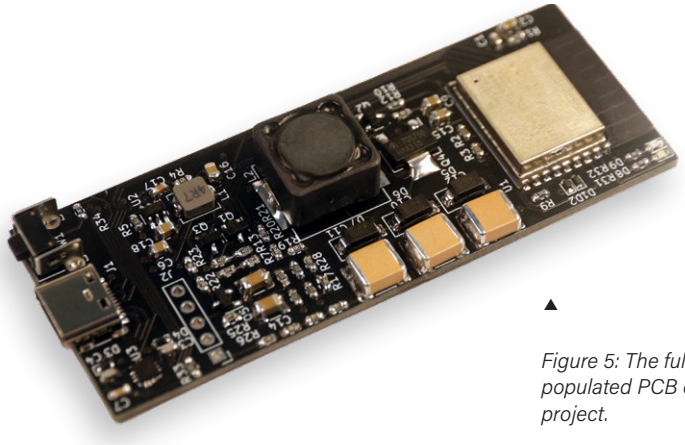


Figure 5: The fully populated PCB of this project.

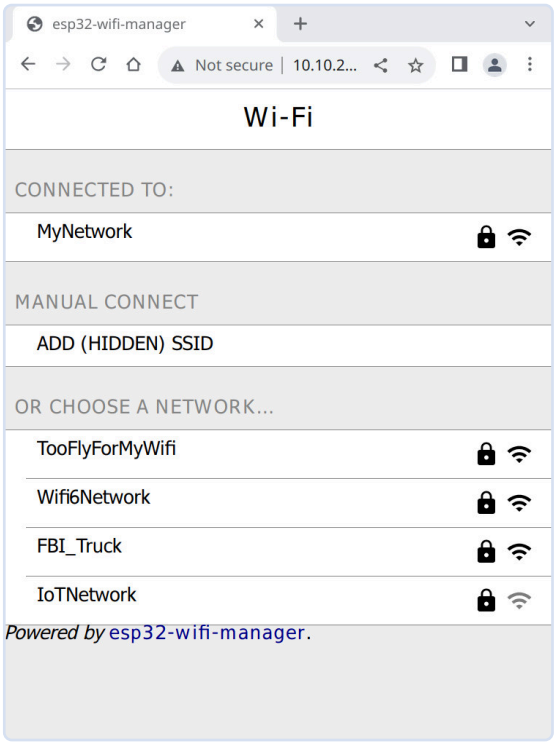


Figure 6: Selecting the correct access point on the ESP32 WiFi Manager page.

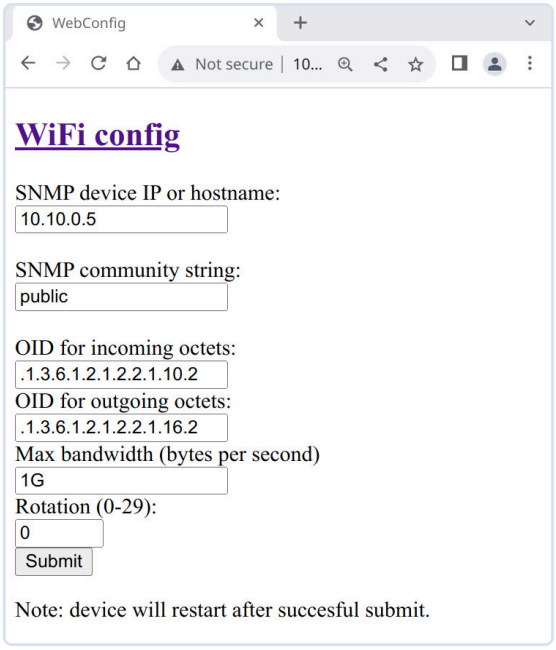


Figure 7: Entering the configuration parameters for the connection.

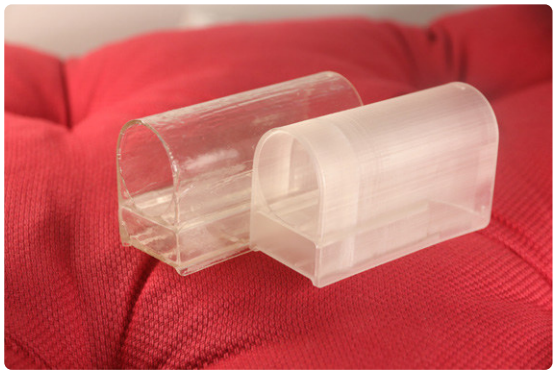


Figure 8: 3D-printed housings for the project: raw (left) and painted with resin and cured under UV light (right).

Secondly, it needs to figure out the currently used internet bandwidth. My internet connection is actually distributed through an Ethernet switch, and this switch keeps statistics about all the data flowing through it. This data is easily obtained using a protocol called the Simple Network Management Protocol (SNMP). This protocol is generally used to monitor IT infrastructure such as routers, switches, servers, etc. It is low-overhead as it only requires a few UDP packets to request the statistics for a network port, so I could easily do this multiple times per second to get a real-time idea of the traffic used. This data is then used to set the speed of the G1 and G2 pulses, meaning faster download results in quicker clockwise spinning. If someone is uploading data instead, the Dekatron will also spin, but in the counter-clockwise direction.

The final feature the firmware needs to have is a sort of configuration interface. I don't like hard-coding configuration into my firmware, so I needed some way to enter the Wi-Fi credentials as well as the switch's IP and SNMP information. Entering Wi-Fi information is called *provisioning*, and ESP-IDF has a perfectly good solution for that, which uses a smartphone app to streamline the process. However, in this case, I went for what's called a *WiFi manager*. When unconfigured, this creates a Wi-Fi access point where you can point a laptop or phone. Once connected, it opens a webpage that allows you to select the Wi-Fi access point and enter the password (**Figure 6**). As the Wi-Fi manager needs a web server anyway, it's easy to expand it to add a configuration page for the SNMP parameters as well (**Figure 7**). To help debugging, there's also some statistics on there, such as the actual voltage negotiated with the power supply.

Designing the Case

With the software and hardware done, I still needed one last thing: a case. In this situation, a case wasn't only needed for decorative reasons: The PCB contains voltages that, while not likely to be deadly, can still bite you pretty bad. I outsourced this work: I asked one of our resident industrial designers if he could make me a case and print it on our SLA 3D-printer. He was nice enough to oblige (Thanks, Kaijie!) and printed a case in transparent resin, so you could still see the Dekatron's internals of.

At least, that was the idea. As it turns out, the way the case is printed makes it opaque rather than transparent. Luckily, there's a pretty easy way to fix this: You simply coat the object with a thin layer of resin, then cure that resin by shining a UV light on it. The results of this are in **Figure 8**, next to an untreated case. I probably could have done a better job making the covering more even, but, honestly, I don't mind the "handmade" aesthetic the current case has: When the Dekatron is fitted, as seen in

Figure 9, it automatically pulls attention to the front of the unit. This obviously looks even better when the unit is turned on: Even if you aren't aware of what exactly it visualizes, the glowing point that goes round and round is really eye-catching, as you can see in **Figure 10**.

Before We Leave

All in all, I made something that I hope, at least partially, qualifies as Retronics. Perhaps, some day in the future, the entire project will qualify as such. Perhaps, by that time, Elektor and Espressif will cooperate on a magazine take-over again, and maybe — just maybe — I'll be allowed to take this unit off its shelf and wax lyrically about all the things that led to its existence. But, for now, it'll be useful as something to look at during my next large downloads.

If you have a Dekatron and want to recreate this design (or simply reuse hardware or software bits for your own purposes), the entire project is open-source. You can find the firmware as well as the PCB artwork and the 3D design files for the enclosure on GitHub [6].

230560-01

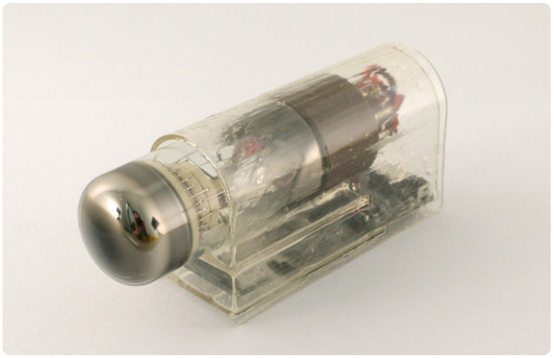


Figure 9: The complete project in its final housing.

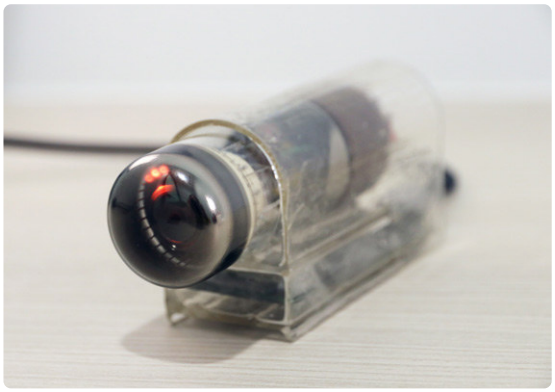


Figure 10: The prototype at work: irresistibly eye-catching!

Questions or Comments?

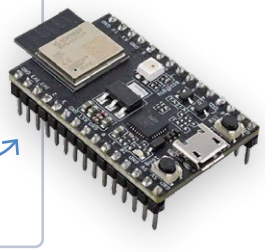
If you have technical questions or comments about this article, feel free to contact the author at jeroen@spritesmods.com or the Elektor editorial team at editor@elektor.com.

About the Author

Jeroen Domburg is a Senior Software and Technical Marketing Manager at Espressif Systems. With more than 20 years of embedded experience, he is involved with both the software as well as the hardware design process of Espressifs SoCs. In his private time, he likes to tinker with electronics as well to make devices that are practically useful as well as devices that are less so.

Related Products

- **Anycubic Photon Mono X – UV Resin SLA 3D Printer**
  www.elektor.com/19831
- **ESP32-C3-DevKitM-1**
  www.elektor.com/20324
- **Dogan Ibrahim, The *Complete ESP32 Projects Guide* (Elektor, 2019)**
  www.elektor.com/18860



WEB LINKS

[1] The Harwell Dekatron computer (WITCH): https://tnmoc.org/witch
[2] Dekatron boost converter circuit: https://threeneurons.wordpress.com/dekatron-stuff/dekatron-do-hickie-kit
[3] GC10/4B Datasheet: http://r-type.org/pdfs/gc10-4b.pdf
[4] KiCad libraries for Espressif chips and modules: https://github.com/espressif/kicad-libraries
[5] USB-PD driver stack for the FUSB302 : https://github.com/Ralim/usb-pd
[6] The project's repository: https://github.com/Spritetm/elek_dekatron

Q&A With Paulus Schoutsen

The Smart Home Revolution

Paulus Schoutsen on Home Assistant, ESPHome, and More

Paulus Schoutsen. (Source: P. Schoutsen / Midjourney)

Questions by Saad Imtiaz (Elektor)

Paulus Schoutsen, the mind behind Home Assistant and ESPHome, talks about his journey into the world of home automation, IoT, and more. Discover how ESPHome's easy-to-use firmware is changing the game for DIY smart devices.

Elektor: Let's begin with your background. Have you always been interested in electronics and engineering? What led you to study for degrees in computer science at the University of Twente?

Schoutsen: I studied Business Information Technology at the University of Twente. It has programming and database courses, but the program focuses on information systems: What information flows through business processes and how you can capture, process, and analyze it. In a way, it's a precursor to my home automation journey. I was always very interested in programming, but not so much in electronics. That came by accident. When the Philips Hue connected light was released in 2012, I immediately bought it. I realized it had a local API and wrote a script to control the light from my computer. This script grew, and on September 17, 2013, I pushed the first version to GitHub; Home Assistant was born. This open-source smart home platform focusing on local control and privacy has become the second-most-active open-source project on GitHub [1].

As Home Assistant grew, I played around with electronics, but there was a big problem: connecting electronics to the internet was too expensive. Adding an Arduino Wi-Fi shield costs $70. But, then, the chip that would change everything arrived: the ESP8266 by Espressif.

The ESP8266 was a $3.50 chip that eventually ran Arduino-compatible code and could connect to a wireless network. This affordable chip finally made it financially feasible to tinker with electronics that could integrate with your smart home.

While the ESP8266 slowly got a foothold in the DIY space, I founded Nabu Casa. Nabu Casa exists to make the development of Home Assistant sustainable. Home Assistant is a big open-source project, and running a project such as this requires a lot of administration, processes, structure, and maintenance — more than anyone can do in their spare time.

About Paulus Schoutsen

Paulus Schoutsen is the founder of Home Assistant, one of the largest open-source projects on GitHub, and Nabu Casa, the company that ensures the long-term viability of Home Assistant. His work revolves around building the "Open Home": his vision of a smart home that offers privacy, choice, and sustainability.

Full-time Nabu Casa employees provide that, funded by subscriptions with extra services we offer users. Nabu Casa has no investors and only exists to satisfy its users.

As Home Assistant grew, we defined our Open Home vision [2]. It's a vision for the smart home based on three values: privacy, choice, and sustainability. From this vision, we realized that if you want your smart home to be local and private, this should apply to both the smart home platform (i.e., Home Assistant) and the devices connected to it.

One of the core developers of Home Assistant was Otto Winter. He was interested in electronics, but found that early ESP8266 projects had too much boilerplate and needed easier integration with Home Assistant. He set out to solve that, and, on January 21, 2018, he released Esphomelib [3]. ESPHome got increasingly popular, but Otto didn't have the time to manage it anymore. Because ESPHome is an essential foundation for people to make devices that match our Open Home values, we wanted to support it. On March 21, 2018, Nabu Casa acquired ESPHome [4]. Today, we have two dedicated full-time developers working on ESPHome, all funded by the people subscribing to Home Assistant Cloud by Nabu Casa.

Elektor: Can you briefly describe ESPHome? How does it differ from other IoT firmware and home automation solutions?

Schoutsen: ESPHome is firmware for ESP8266, ESP32, and Raspberry Pi Pico boards that allows users to create and maintain smart home devices easily. ESPHome takes away all the boilerplate of writing software and enables you to focus on making your hardware.

To give you an example of how easy it is to make something with ESPHome: Just take a temperature sensor and connect it to your ESP32, tell ESPHome, in a configuration file, which pin the sensor is connected to, and hit *Install*. ESPHome will then generate all the necessary firmware and install it on your ESP32 device. The device will boot up, connect to your Wi-Fi network, and your temperature will be available in Home Assistant. That's it.

The first time you install ESPHome, you must connect the board to your computer. Subsequent updates happen over the air. Connect a second sensor to the same board? Just update the configuration file and hit *Install*. No matter where in the house your device resides, updates are seamless.

A bonus feature of ESPHome is that devices can also act as a Bluetooth proxy for Home Assistant. This allows Home Assistant to expand its Bluetooth reach by using ESP32s running ESPHome to listen for BLE packets and establish direct connections for control. To make this extra useful for tinkerers, we've also introduced BTHome [5], a BLE protocol to send data that these Bluetooth proxies pick up (**Figure 1**).

Elektor: ESPHome has a sizable following in the DIY smart home community. What do you believe has led to its popularity?

Schoutsen: Ease of use. We've been relentlessly focusing on making it easier to play with hardware. With ESPHome, you'll never be fighting with C++ compilation errors or misconfigured MQTT topics. And, it's easy to share your configuration files with other users, making it possible for beginners to build working devices immediately.

We've also developed a tool to make sharing configurations easier: ESP Web Tools. This is a web-based installer for ESP8266 and ESP32 boards that allows users to one-click-install firmware on their devices directly from their browser. Now, users can buy ready-made devices and install ESPHome on them. For example, we allow users to turn their M5Stack Atom Echo Dev Kit into a voice assistant for Home Assistant directly from their browser [6].

ESP Web Tools was a great help in sharing our ESPHome-based projects, but we believe this kind of technology should be shared with everyone. So, nowadays, it also powers the installers of Tasmota, WLED, and a lot of other firmware.

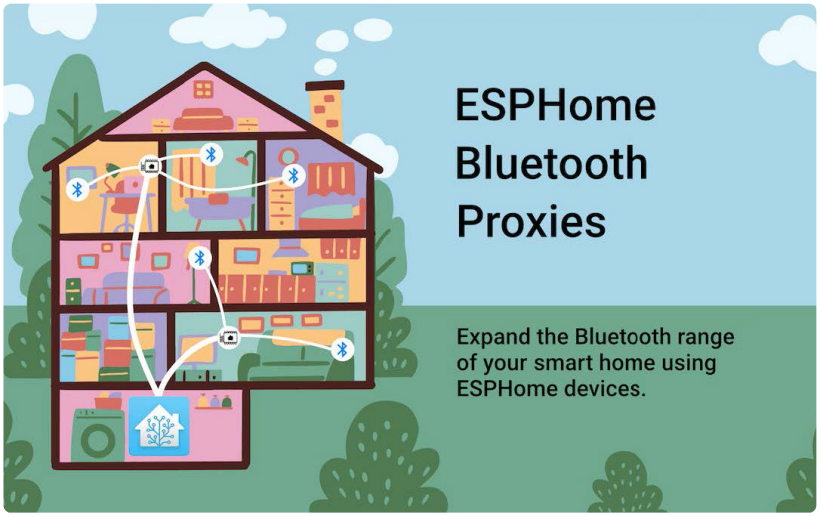
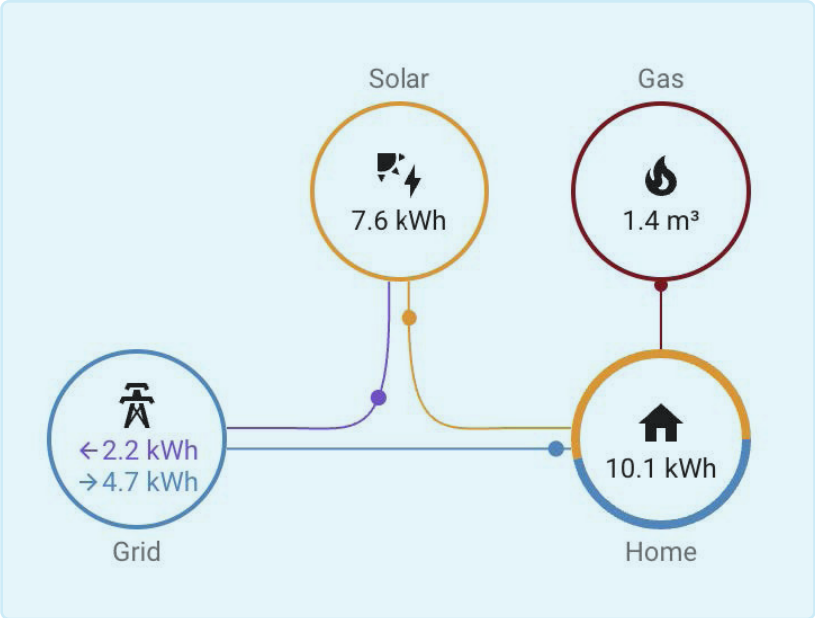


Figure 1: Home Assistant can extend its Bluetooth range by using ESPHome-powered Bluetooth proxy devices. (Source: Paulus Schoutsen)



▲

Figure 2: Energy dashboard in Home Assistant for easy access to energy consumption. Additionally, it provides indicators to assess grid reliance. (Source: Paulus Schoutsen)

Elektor: How can ESPHome make it easier to integrate numerous sensors and devices into a home automation environment, particularly for people without extensive coding experience?

Schoutsen: The great thing is that, to use ESPHome, you don't need to know how to program. The problem with electronics is that you're learning three things simultaneously: How do I build hardware, interact with that hardware, and get that interaction into another system? With ESPHome, a user can focus on just the first step: experimenting with hardware. The rest, we take care of for you. We allow users to concentrate on the fun part of building their hardware.

Elektor: YAML-based setup is one of ESPHome's standout features. Can you explain why you picked this method and how it benefits users?

Schoutsen: When ESPHome was founded, it mimicked Home Assistant's architecture and configuration format. It fits the ESPHome format very well. An integration represents each connected sensor. You can configure that integration to tune the sensors, and it will be included the next time you update your device.

Elektor: The ESPHome community has helped to expand the list of compatible devices and components. How crucial is community involvement to the project's success, and how do you effectively manage this collaboration?

Schoutsen: For open-source home automation, the community is indispensable. It takes time to dive into how each sensor works, figure out the drivers, and hook it into ESPHome. Our full-time maintainers are working with the community to review their contributions and get things merged.

Topics of interest this year for ESPHome have been voice assistant, Bluetooth proxy, and e-ink displays:

- Voice assistant lets you turn an ESPHome device into a voice assistant that controls Home Assistant. ESPHome is responsible for capturing audio, while Home Assistant processes the speech and acts on it. You can install a voice assistant from your browser [6].
- Bluetooth proxy allows turning any ESP32 device into a Bluetooth proxy for Home Assistant. Home Assistant uses the ESP32's BLE to listen for packets and connect to devices to control them. You can install a Bluetooth proxy from your browser [7].
- E-ink displays are hot. We've been adding support for more, so building the most fantastic dashboards for your smart home is more accessible.

Elektor: IoT and home automation security is a major worry. What security and privacy measures does ESPHome take to protect user data and connected devices?

Schoutsen: By default, each new ESPHome device is encrypted using the Noise protocol. It's a fast and efficient protocol that Wireguard and WhatsApp also use. It means that no one can snoop on ESPHome data on the network.

If you use ESPHome devices together with Home Assistant, your smart home will be fully open source and work fully locally and encrypted. No data will be shared with anyone, and your smart home can even function when it's not connected to the internet.

When ESPHome contains a security update, it can recompile your configuration file and update your device wirelessly to ensure it is running the latest version. To help you keep your smart home secure, updating your ESPHome devices from within Home Assistant is possible.

Elektor: Could you offer some real-world use cases or projects where Home Assistant and ESPHome have made a significant or unique difference?

Schoutsen: Humans emit CO_2 into the atmosphere, causing the climate to change and the world to heat up. One way we can all help with this is to ensure that our individual energy footprint is as small as possible. Our homes represent a substantial part of our energy use.

ESPHome is a firmware for ESP8266, ESP32, and Raspberry Pi Pico boards that allows users to create and maintain smart home devices easily.

In 2021, we introduced home energy management in Home Assistant [8]. It allows users to monitor their energy usage, transition to sustainable energy, and save money (**Figure 2**).

The most crucial information is knowing how much energy flows between the electricity grid and your home. Real-time access to this information allows users to decide when to adjust their energy usage by charging their e-bikes, doing the laundry, or reducing their heating.

Getting this information from electricity meters is a somewhat standardized task, but adoption of different standards is usually on a per-country or per-region basis. Together with our community, we built open-source devices based on ESPHome to support various standards. For example, the SlimmeLezer by Marcel Zuidwijk [9] connects directly to P1 ports (used in the Netherlands and some other EU countries) to access real-time energy usage and cumulative energy and gas consumption. Other electricity meters have a pulsing LED instead of a port. Each pulse represents a unit of energy used. With the Home Assistant Glow by Klaas Schoute [10], these LED pulses can be counted for accurate energy consumption, and Home Assistant can track your usage over time (**Figure 3**).

Elektor: What further advancements or enhancements may ESPHome users expect as the IoT ecosystem evolves? Are you thinking about joining ESPHome and Home Assistant together?

Schoutsen: One of our values for Open Home is choice. So, while we ensure that the integration between Home Assistant and ESPHome is the best, we're not interested in locking people out of controlling ESPHome devices via other systems.

There are some more opportunities for tighter integration. For example, when a user buys a device with ESPHome firmware, they need to use the ESPHome dashboard to adopt the device and be able to install updates. We want to streamline that, so users do not need to learn all that to stay up-to-date. It's part of our ongoing efforts to improve product experience in our *Works with ESPHome* program.

The *Works with ESPHome* program allows creators to carry our badge if their devices fulfill a set of requirements by, for example, enabling features for easier onboarding and allowing users to customize the device's configuration.

Elektor: What tips would you give to beginners interested in getting started with ESPHome and starting on their own home automation projects? Do you have any resources or best practices to share?

Schoutsen: Start with a simple ESP32 development board and a temperature sensor. Get it working, set it up in Home Assistant, and start playing with the ESPHome device configuration. You will see the changes instantly reflected in Home Assistant as you update your device.

The easiest way to get started with ESPHome is by installing it from the add-on store in Home Assistant [11]. It's a 1-click installation. Click *Add device* and a wizard will guide you in installing your first device. Happy automating!

Figure 3: Home Assistant Glow by Klaas Schoute. (Source: Paulus Schoutsen)

Elektor: ESPHome is known for its user-friendly approach, but what actions or resources would you recommend for total novices to assist them get started with their first ESPHome project?

Schoutsen: I would go to YouTube. There is a wide variety of getting started videos.

Elektor: Is there any idea or continuing attempt to provide more detailed documentation or courses designed primarily for beginners in the spirit of making home automation more accessible?

Schoutsen: We're looking into that for 2024 — we'd love to have an easy starter kit for ESPHome.

Elektor: Can you offer any information on planned features or upgrades that will make ESPHome even more user-friendly? Are there any plans to improve the user interface?

Schoutsen: When you create an ESPHome device, you embed your Wi-Fi network and password into the firmware. That way, it will automatically find your network and connect to it. Great, except when you want to share it with friends or sell your creations. To make this work, we've developed Improv Wi-Fi [12]. It's a standard that enables setting up devices using BLE or Serial.

One improvement that we're planning is to add support for Improv Wi-Fi over BLE to Home Assistant. So, when you plug in a purchased device, Home Assistant can guide you through the onboarding experience and get the device connected and configured.

Elektor: ESPHome already works with a wide variety of sensors and devices. Are there plans to add more sensors and components in the future, and if so, what types of sensors can users anticipate seeing?

Schoutsen: We will never stop! We've seen a recent surge in mmWave sensors added. They offer a new way of presence-sensing in a room that works even when you sit very still.

We also want to explore cheaper air-quality sensors. If the air in your room is bad, it can impact your thinking and performance. It would be cool to unlock cheap sensors based on ESPHome to help people live healthier lives.

Elektor: Users may encounter difficulties while integrating custom sensors or devices. Is there any plan to simplify this process even more and make it easier for makers to connect their own DIY sensors to ESPHome?

Schoutsen: Any ESPHome configuration generates a C++ project that is compiled and installed on your device. If a developer of custom sensors is creating their own protocol, they need to write C++ to integrate this into ESPHome. We have protocols such as BTHome [5] to send data, but that is meant to get the data into Home Assistant.

With ESPHome, a user can focus on just the first step: experimenting with hardware. The rest we take care of for you. We allow users to concentrate on the fun part of building their hardware.

Elektor: Are there any plans for Home Assistant to become a cloud-based solution as wel, where you can use an ESP32 sensor setup to send data and visualize data, and not set up and run Home Assistant on the local server?

Schoutsen: No. Your smart home should run locally and keep your data there. We've released the Home Assistant Green [13] to make it easier for users to run Home Assistant. It runs Home Assistant out of the box and is the easiest way for new users to get started with Home Assistant. It retails for $99.

Elektor: Are ESPHome and Home Assistant involved in any partnerships or collaborations with hardware makers, or in any other efforts to improve device compatibility and user experience?

Schoutsen: Yes. We have the *Made for ESPHome* [14] program to work with creators selling ESPHome-based devices to get the best experience. It has a set of requirements to ensure the devices are customizable and open. We also have the *Works with Home Assistant* [15] program. This program ensures that we test devices for compatibility and work with the vendor to fix any issues.

Elektor: How does Home Assistant manage introducing new features while supporting existing ones to ensure a reliable and user-friendly experience for all users as it evolves?

Schoutsen: It takes a lot of people. Last year, Home Assistant was the second-most-active open-source project on all of GitHub. We had 13,200 people contribute. All these people are working on keeping Home Assistant compatible as the smart home space evolves by supporting new devices or new features in devices. The power of Home Assistant is the community behind it. The community makes it the best platform and offers us the best ideas to make our homes smart.

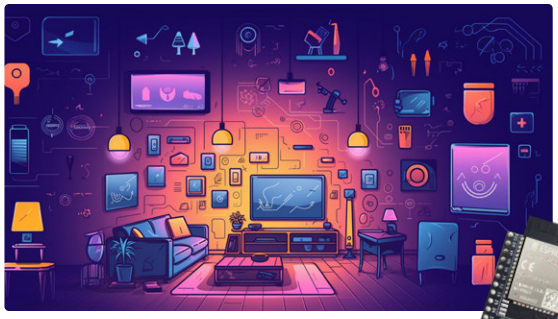
Elektor: Many newcomers to home automation and IoT may be concerned about the learning curve. What tips would you give to new users to assist them overcome obstacles and frustrations?

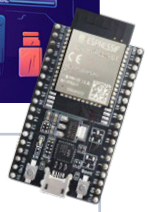
Schoutsen: Don't start with anything fancy; stick to the default path for now. Some alternative ways exist to install Home Assistant but stick to Home Assistant Operating System. The same goes for ESPHome; stick to standard ESP32 boards at first, and don't try to go fancy with less common boards like ESP32-S3. It only makes things more complicated.

Elektor: How are firmware and bug patches handled? Are there plans to automate or streamline this process to ensure that users always have the most recent and stable versions?

Schoutsen: Security is a top priority in our vision for the Open Home, which makes it a top priority for our projects like Home Assistant and ESPHome. We've made it very easy to keep everything up to date: Home Assistant allows users to update all parts of Home Assistant with a single click from the interface. This includes updating ESPHome devices over-the-air when a new ESPHome version comes out.

230594-01





WEB LINKS

[1] Top open-source projects: https://octoverse.github.com/2022/state-of-open-source
[2] The Open Home: https://home-assistant.io/blog/2021/12/23/the-open-home
[3] Esphomelib: https://community.home-assistant.io/t/esphomelib-library-to-greatly-simplify-home-assistant-integration-with-esp32/40245
[4] ESPHome acquisition: https://home-assistant.io/blog/2021/03/18/nabu-casa-has-acquired-esphome
[5] BTHome protocol: https://bthome.io
[6] Voice assistant for Home Assistant: https://home-assistant.io/voice_control/thirteen-usd-voice-remote
[7] Bluetooth proxy installation: https://esphome.github.io/bluetooth-proxies
[8] Energy Management in Home Assistant: https://home-assistant.io/blog/2021/08/04/home-energy-management
[9] SlimmeLezer by Marcel Zuidwijk: https://slimmelezer.nl
[10] Home Assistant Glow by Klaas Schoute: https://github.com/klaasnicolaas/home-assistant-glow
[11] Add-on store in Home Assistant: https://my.home-assistant.io/redirect/_change/?redirect=supervisor_addon%2F%3Faddon%3D5c53de3b_esphome
[12] Improv Wi-Fi: https://improv-wifi.com
[13] Home Assistant Green: https://home-assistant.io/green
[14] Made for ESPHome: https://esphome.io/guides/made_for_esphome.html
[15] Works with Home Assistant: https://partner.home-assistant.io

Burn, Firmware, Burn!

Flashing your ESP32

By Pedro Minatel, Espressif

If you're new to embedded development, you'll get familiar with the terms burning, flashing, and writing the firmware into some microcontroller or flash memory. This is a basic step for programming the application on the device. In this article, we take a deeper look how this can be done for ESP32 controllers during the development and production phase in an efficient and secure way.

The concept of "burning the firmware" was often used on one-time programmable (OTP) memory, such as the programmable read-only memory (PROM), where you can only write data to the memory once, so, the term "burn" implies irreversibility. The explanation for this is the fact that a physical connection path is burned, like a fuse, between two points, interrupting the current flow, and changing this "bit" from a 1 to 0.

In contrast to this, an ESP32 application is stored in an external flash memory, and this allows you to flash the firmware or rewrite data thousands of times.

Flashing the ESP32

On the ESP32, the flashing process is easier than on most other microcontrollers, mainly because the ESP32 can be flashed over UART. There is no need for a special programmer or JTAG (although the JTAG port is present on the ESP32 for debugging) — just an external USB-to-serial adapter. If you are using a recent ESP32, you will have both USB serial for flashing and USB JTAG for debugging the SoC internally through software implementation.

ESP32 Download Mode

To flash the firmware on any ESP32, you will need to be in "download mode," handled by the ROM boot stage (1st-stage bootloader); otherwise, the boot process will continue and will try to read the flash bootloader (2nd-stage bootloader) and then the application.

The download mode is activated by pulling down the BOOT GPIO pin (usually GPIO0 or GPIO9), holding it there and then resetting the SoC. Once download mode is active, the ESP32 can receive the firmware via the UART and store it in the flash.

Internal USB Serial and JTAG

In some of the most recent SoCs, two new and extremely useful functionalities have been introduced: USB serial and JTAG. These improve development and reduce your bill-of-materials by removing the external USB-to-Serial, and will also save some pins (especially for the JTAG).

When using the UART for flashing, you will need to use two GPIO pins for UART (TX and RX) and one GPIO pin to enable download mode. You'll also need to connect the reset pin to the reset circuit with the download mode pin. On the other side, by using the internal USB serial, only the D+ and D- GPIOs are needed because the automatic download mode is handled internally.

Supported SoCs are:

- ESP32-C3
- ESP32-S3 (also includes USB host and device)
- ESP32-C6
- ESP32-H2
- ESP32-P4

To use the USB Serial and JTAG on the supported SoCs, you need to add a USB connector to the D+ and D- or a test jig connected directly via USB cable to the computer. Please see the datasheet for the USB pins. This means that on each ESP32 you have, internally (on the aforementioned SoCs), the USB serial and JTAG for flashing and debugging, with no extra cost or external hardware.

The ESP32-S2 has no USB serial or JTAG, but it has USB-OTG (host and device), so you can flash using DFU (device firmware upgrade) instead of USB serial.

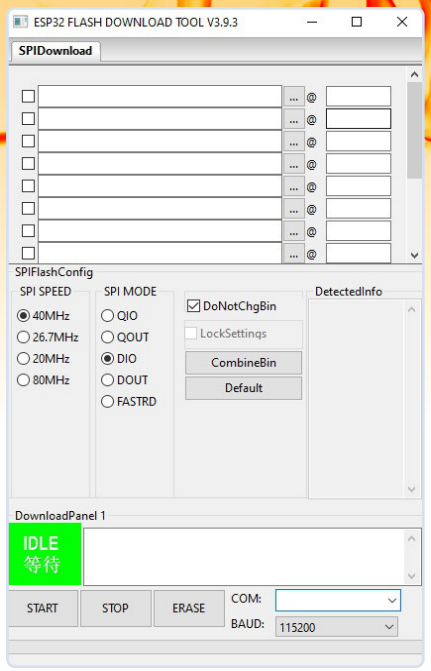


Figure 1: Flash Download Tools is Espressif's official GUI-based flashing application, which you can use without the need to install ESP-IDF or run ESPTool from the CLI.

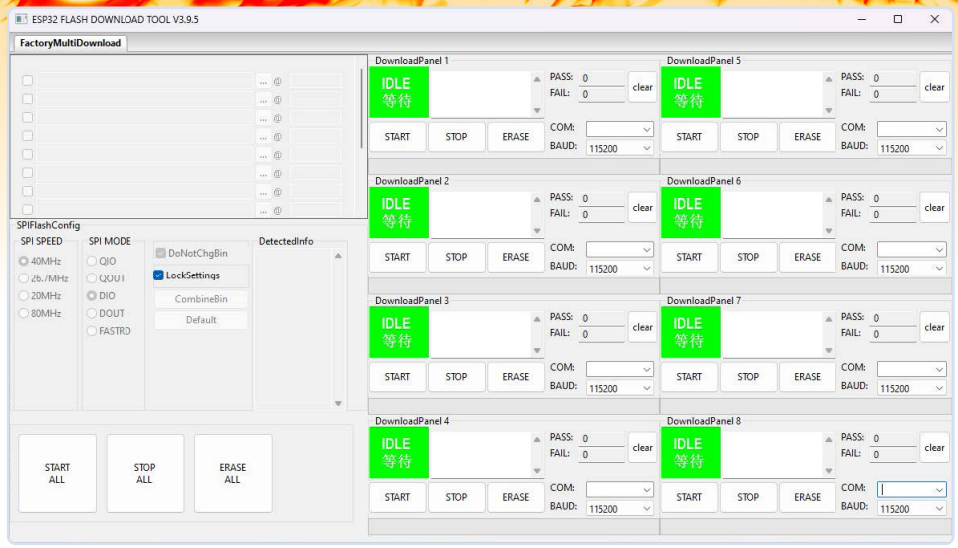


Figure 2: Flash Download Tools in factory mode.

Is That Secure?

If you have security concerns about shipping your product with a serial and JTAG interface built-in, both features can be disabled using the *DIS_USB_JTAG* and *DIS_USB_SERIAL_JTAG* eFuses during the mass production phase. This action is not reversible, so don't play with it during development.

Another interesting eFuse is the *USB_EXCHG_PINS*. This eFuse will exchange USB pins D+ and D- in case you accidentally swapped pins in your hardware design.

It Is Time to Flash the Device!

There are different ways to flash the ESP32. You must decide which way is more convenient, depending on the project's characteristics or production stage. The following is an overview of all the possibilities available for flashing the ESP32.

ESPTool

The first and most-used tool is ESPTool [1]. ESPTool is a Python-based tool and the main tool for flashing the firmware. It's integrated into the ESP-IDF, so this is the tool used when you flash the firmware via command-line-interface (CLI) or via the IDE.

To use the ESPTool directly from the CLI, you need to set some parameters, like this:

esptool.py -p -b --before default_reset --after hard_reset --chip write_flash --flash_mode --flash_size --flash_freq

If you are skeptical about this command, good news: You can also flash firmware built with ESP-IDF by using the *idf.py -p flash* command.

Here are some of the ESPTool features you can use beyond just flashing firmware:

- Perform full chip flash erase or erase just a region of the flash
- Read and write from flash memory and RAM
- Run application code in flash
- Read MAC addresses from OTP ROM
- Read the Chip ID from OTP ROM
- Dump headers from a binary file (bootloader or application)
- Merge multiple raw binary files into a single file for later flashing
- Get some security-related data

If you need to flash the firmware during manufacturing for mass production, you can use ESPTool to automate the flashing procedure through a script.

Flash Download Tools

Flash Download Tools is a graphical Windows-only application that you can use for flashing without the need to install the ESP-IDF or run ESPTool at the command line. The GUI is quite simple and easy to use and does not require installation (**Figure 1**).

To use Flash Download Tools, you need to select all the binaries you want to flash, and the address in flash memory for each binary.

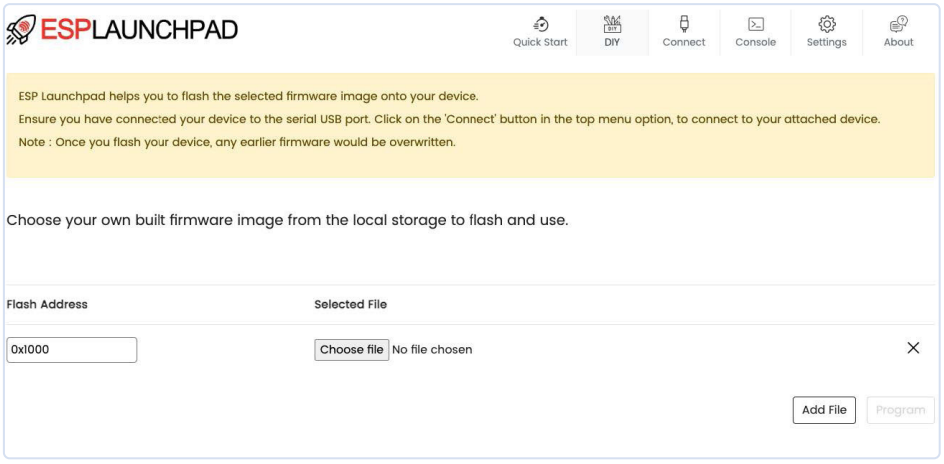


Figure 3: ESP Launchpad accessed directly from the Chrome browser.

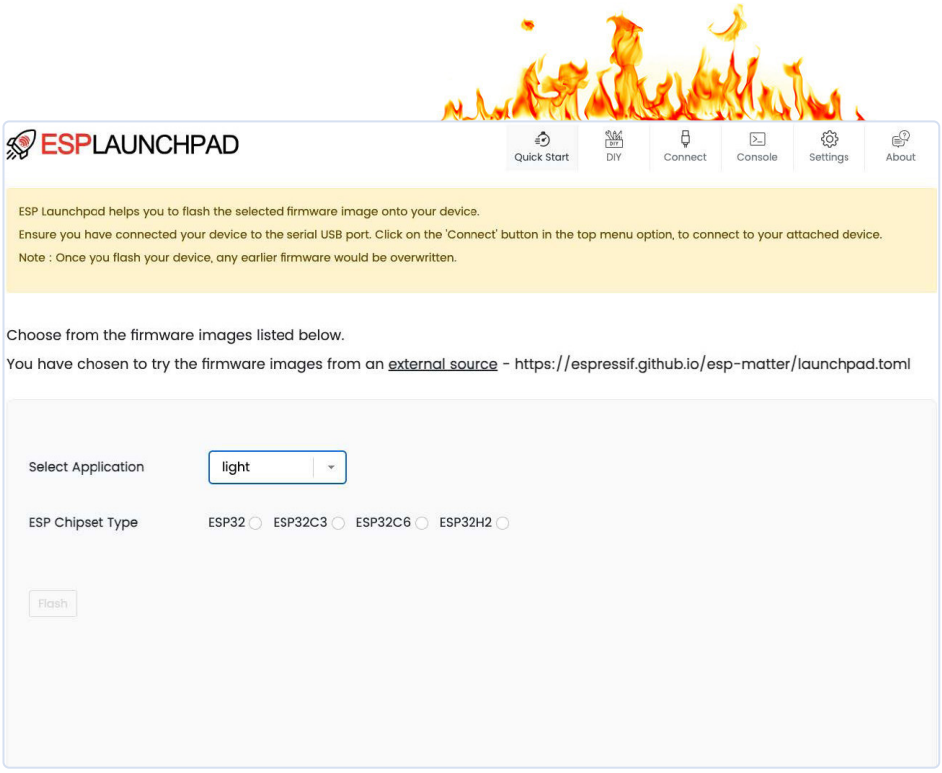


Figure 4: ESP Launchpad can be customized using some additional configuration files.

Flash Download Tools also works in factory mode. In this mode, you can select multiple serial interfaces and flash the firmware simultaneously in just one click (**Figure 2**).

ESP Launchpad

Sometimes, you don't want to install or run an application on your computer, and you prefer to use only web-based applications. In this case, you can try ESP Launchpad directly from your Chrome browser.

ESP Launchpad [2] is a web application based on *esptool-js*, which allows you to flash the firmware directly from the browser without any installation, thanks to the WebUSB API. You can also monitor the console output and erase the flash (wiping all data from flash).

ESP Launchpad is open-source software and can be customized (**Figure 3**) with some additional files to tell it where the firmware file is stored and the supported target devices for that file (**Figure 4**). Thus, you just need to send the link to your customer and the flashing process will be simple and intuitive.

Example of the TOML File

In the TOML configuration file, you can add the URL for the binaries and specify the targets for each binary. It's important to note that the binary file must be a single file, meaning the merged version including all required binaries. You can merge them using ESPTool or Flash Download Tools. Here's an example TOML:

```
esp_toml_version = 1.0
firmware_images_url = "https://espressif.github.io/
esp-matter/"
supported_apps = ["light","light_switch"]
[light]
chipsets = ["ESP32","ESP32C3","ESP32C6","ESP32H2"]
image.esp32 = "esp32_light.bin"
image.esp32c3 = "esp32c3_light.bin"
image.esp32c6 = "esp32c6_light.bin"
image.esp32h2 = "esp32h2_light.bin"
ios_app_url = "https://apps.apple.com/app/esp-rainmaker/
id1497491540"
android_app_url = ""

[light_switch]
chipsets = ["ESP32","ESP32C3","ESP32C6","ESP32H2"]
image.esp32 = "esp32_light_switch.bin"
image.esp32c3 = "esp32c3_light_switch.bin"
image.esp32c6 = "esp32c6_light_switch.bin"
image.esp32h2 = "esp32h2_light_switch.bin"
ios_app_url = "https://apps.apple.com/app/esp-rainmaker/
id1497491540"
android_app_url = ""
```

After hosting this file on a public server, you can use the URL combined with the ESP Launchpad URL [3].

A useful case for ESP Launchpad is as a firmware update tool to be used by the final customer, after sale, when no over-the-air updates are available.

ESP Serial Flasher

Imagine you need to flash your ESP32 device through another device, or host microcontroller. You might look at the ESP Serial Flasher project. In this project, you can use a host microcontroller to flash a target ESP32, or you can use one ESP32 to flash another ESP32 (**Figure 5**).

One common application for this feature is when the ESP32 is being used as a radio coprocessor, and you want to flash a new firmware version via the host device.

Currently, we support the following host controllers:

- ESP32
- STM32
- Raspberry Pi
- Any MCU running Zephyr OS

The target microcontrollers that support update via UART are:

- ESP32
- ESP8266
- ESP32-S2
- ESP32-S3
- ESP32-C3
- ESP32-C2
- ESP32-H2
- ESP32-C6
- ESP32-P4

Over-the-Air Update

Finally, the over-the-air (OTA) update is another great solution for firmware updating, especially in the field. This technique is often used to update firmware via the internet or any local network without dealing with the individual firmware updates physically.

The main advantage of the OTA update comes in when you have any issues with your current firmware version or need to add a new functionality to it. You can do it remotely and in bulk, updating thousands of devices at the same time. This will save you time and money.

To get started with OTA, you can check the examples under the ESP-IDF project on GitHub or in your local installation.

Bonus: ESP USB Bridge

This is another ESP-IDF project that can help you during development, especially if you are using an ESP32 without USB serial or JTAG, such as the ESP32, ESP32-C2, and ESP32-S2.

You can turn any ESP32-S2 or ESP32-S3 into a serial-to-USB and JTAG device and use it for flashing and debugging another ESP32. You just need to flash any ESP32-S2 or ESP32-S3 with ESP USB Bridge and connect the GPIOs to the target microcontroller (**Figure 6**).

You can use this project alongside all the flashing tools, such as ESPTool, Flash Download Tools, and ESP Launchpad. You can also use the USB-MSC (Mass Storage Class) by dropping the binary in UF2 format onto the USB mass storage that appears when you connect the ESP-USB-Bridge to your computer.

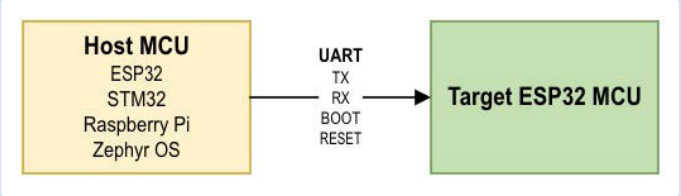


Figure 5: With the ESP Serial Flasher project, you can use a host microcontroller to flash a target ESP32, or even use another ESP32 as the host for flashing.

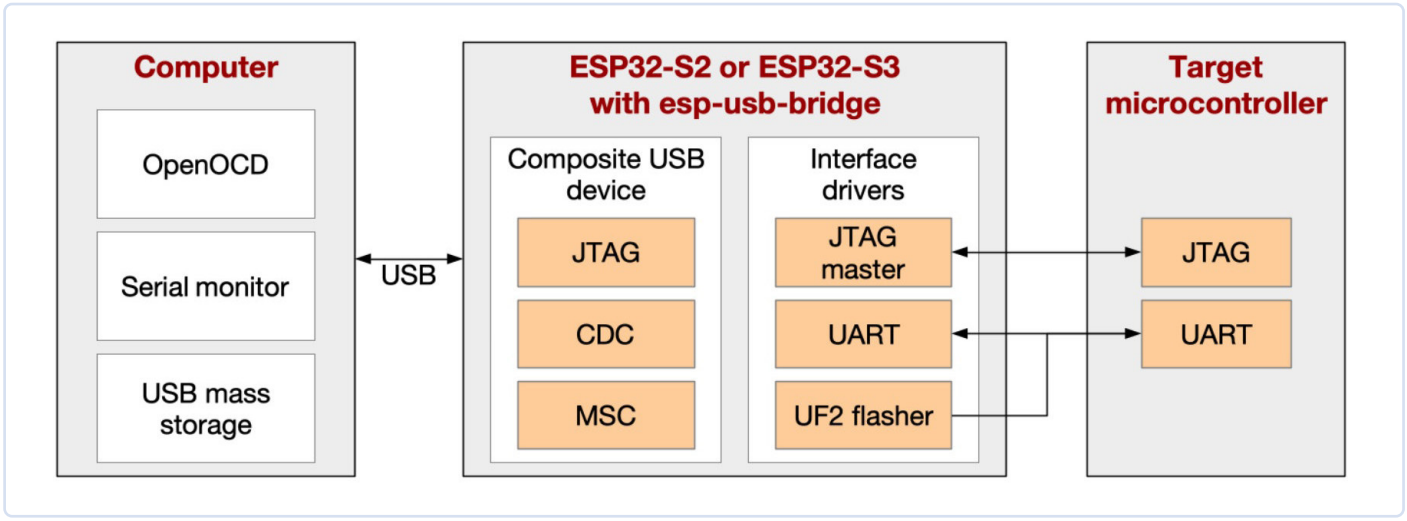


Figure 6: ESP USB Bridge is an ESP-IDF project utilizing an ESP32-S2 or an ESP32-S3 to create a link between a computer (PC) and a target microcontroller (MCU).

Many Ways

In the world of ESP32 microcontrollers, there are many ways to upload new firmware to your ESP32. While ESPTool may seem like the go-to solution, it is important to keep in mind that certain scenarios call for more advanced tools featuring graphical user interfaces (GUIs) or simplified flashing processes that cater to users across various platforms without requiring any software installation.

This article aims to give ideas for improving product development and production. It also aims to empower individuals with the ability to easily flash or update firmware to the latest versions, regardless of their technical background or the platform they are using.

230625-01

Questions or Comments?

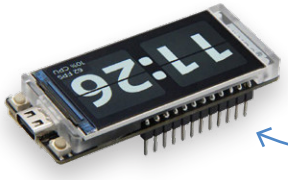
If you have questions about this interview, feel free to e-mail the author at pedro.minatel@espressif.com or the Elektor editorial team at editor@elektor.com.

About the Author

Pedro Minatel is a Developer Advocate at Espressif. He holds an electronics technical degree and an associate's degree in information technology. Pedro started working for Espressif in 2021, but he has been an active community member since 2014, publishing articles and presenting talks at conferences about IoT and the maker community. Currently, he is responsible for the annual Espressif Developers Conference, known as the DevCon.

Related Products

- **ESP32-C3-DevKitM-1**
  www.elektor.com/20324
- **LILYGO T-Display-S3 ESP32-S3 Development Board (with Headers)**
  www.elektor.com/20299



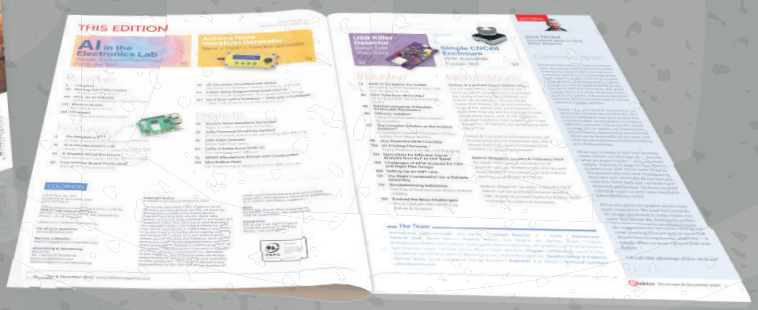
WEB LINKS

[1] ESPTool: https://github.com/espressif/esptool
[2] ESP Launchpad: https://espressif.github.io/esp-launchpad/
[3] ESP Launchpad URL: https://espressif.github.io/esp-launchpad/?flashConfigURL=https://espressif.github.io/esp-matter/launchpad.toml

20% discount
on the first year of your membership

Join the Elektor Community

Take out a elektor GOLD membership!



- The Elektor web archive from 1974!
- 8x Elektor Magazine (print)
- 8x Elektor Magazine (digital)
- 10% discount in our web shop and exclusive offers
- Access to more than 5000 Gerber files
- Free shipping within US, UK & Ireland

www.elektormagazine.com/gold-member

Use the coupon code:

ESPRESSIF20

ESPRESSIF × elektor

CASE STUDY

From Rockets to Cellos

Practical Applications and Considerations When Creating Wirelessly Enabled Solutions

By Sorin Jayaweera, GXB Ventures

What do you get when you mix a push button and an antenna? A rocket computer, obviously!

The Dream

In this article, we will try to provide you with the essential tools for initiating wireless projects, focusing on core hardware, communication network structures, and key considerations.

We started our journey trying to build a network of inexpensive, small, and wireless "help" buttons, in the hopes that we could scatter them around our extended family members' homes, to allow easy contact with the rest of the family in the event of an emergency. On the most abstract level, each button requires the ability to record data — whether the button's been pressed — and the ability to transmit that data in some way to something that can make use it. We also needed to develop a gateway to the internet that could simultaneously monitor the local network while relaying alerts to remote family members.

This is a seemingly simple task, but it's built on the same foundations of a lot of other cool real-world projects. For instance, using the same systems as the button for a short-range communication network, but adding sensors for recording environmental data, could be the basis of an efficient local weather monitoring station. The home use we created would answer questions such as, "what's the current and high/low temperatures in the attic, the basement, and the chicken coop outside?" From that data, we could decide whether to automatically activate fans to cool those spaces, or close their vents to keep them insulated. Several of these simple and inexpensive weather monitoring stations working in tandem across a farm could provide invaluable information to better care for crops.

Increasing the wireless range and adding in relevant sensors such as accelerometers and GPS yields a hobbyist model rocket tracker and flight data logger. Drastically increasing the range could yield a high-altitude balloon telemetry and tracking system. Conversely, drastically decreasing latency at low range is useful for electronic music system design (i.e., making small wireless pedals that communicate with whatever microcontroller is at the center of music production). Really, a lot of interesting things become possible with just a few common methods for communication, data logging, and signal processing. So, what are the fundamental requirements at the center of these types of projects?

The Technicals

To address the requirements mentioned earlier, the circuits must be compact, lightweight, and have efficient power management. Further, every project should have the ability to communicate wirelessly via a network — for example, ad hoc "mesh" networks or "hub and spoke" networks, etc. (we'll get to what these are shortly). These networks

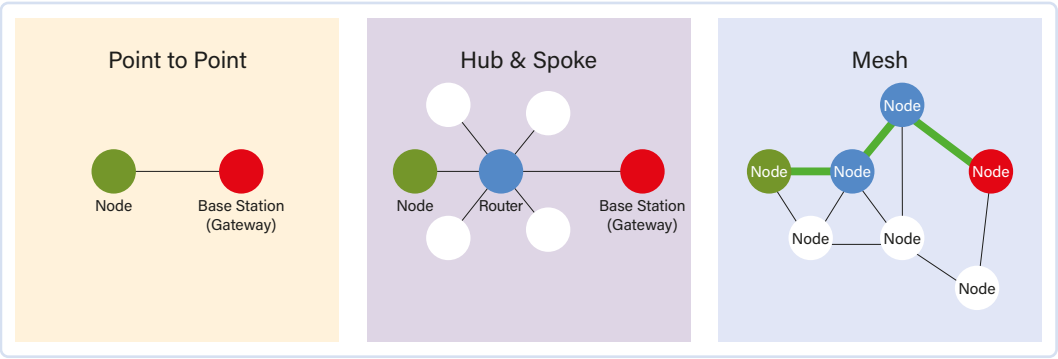


Figure 1: Network topologies.

can be at a relatively small scale — for example, the help button — or long-range for the rockets, but the overall structure of these networks is similar.

For the help buttons and subsequent projects, we ruled out powerful mini-computers such as the Raspberry Pi — they can't tolerate hard shutdowns and have a relatively large energy requirement. Initially, we used Teensy controllers, which are really powerful Arduino-compatible microcontrollers for signal processing and are a great solution for many problems. While they are very adaptable, they require additional peripherals for everything they need to do, especially wireless communications, and that gets clunky as projects expand. Because long-range wireless communications are so important, we started looking for pre-made integrated solutions to minimize the complexity of our setup.

We tried the Heltec LoRa boards, as those have Wi-Fi, Bluetooth, and LoRa built in. While they seemed promising, we ran into numerous struggles trying to get the boards to work, and found their forums and technical support to be unhelpful. Example code they provided wouldn't compile properly without figuring out how to edit the source and linked libraries ourselves, and even the shipped antennas were mistuned for the LoRa frequency bands for which they were supposed to be optimized. We kept looking, and discovered the various Espressif chips and the development boards that use them.

Espressif chips have integrated Wi-Fi and Bluetooth, reducing the complexity of our overall system. Using the boards' built-in antennas works for many relatively short-distance local IoT projects, as range can be extended by using several chips and forwarding messages. To be fair, not all commercial boards worked well — doing an ESP-NOW (Wi-Fi) range test using an ESP-WROOM32 board vs an ESP32-C3 board yielded ranges of 3 meters and 80 meters, respectively. But, after narrowing down the alternatives, we really enjoyed integrating the ESP32-C3 into our wireless projects. This was largely due to the chip's miniscule size, specific support for our types of projects, low power consumption, and integrated communication with a well-designed printed antenna.

Now that we had our main "computer," all that was left was to design the PCB and additional sensors and I/O for each project. For the short-range call buttons, we decided to make each button have its own C3 module operating on coin cell batteries, turning on only when the button was pressed. In order to receive the help signals, we also had an always-listening gateway, which then connected to the general internet and sent a text message to us. This is a small version of a "hub-and-spoke" network (**Figure 1**), with always-listening gateways connected to sensors that only turn on and transmit when required. Because each button only has to turn on when broadcasting its own signal, this greatly optimizes battery life. We implemented this system

Requirement Comparison

Board	Cost (USD)	Power (mW)	Pros	Cons
Arduino 	$15+	~400	+ Easy to learn + Rapid to prototype	- No multitasking - No communications - (Relatively) large
Teensy 	$15–40+	~800	+ Small + Very Fast + Extensive audio support + Active community	- No integrated communication - Large energy requirement
Raspberry Pi 	$15–80+	~1200	+ A true computer + Multi-threading + Any coding language + Integrated wireless (Bluetooth, WiFi)	- (Relatively) expensive - Can't tolerate hard shutdowns - Higher energy requirements than microcontrollers - Larger than microcontrollers - Large power requirement
ESP32 	$2–$5+	6–300	+ Arduino-compatible + Small (dev boards) & smaller (modules) + Integrated wireless (WiFi, Bluetooth 5 / BLE) + Inexpensive + Optimized for low energy	- Only connects to 2.4 GHz Wi-Fi - ESPNow & ESP-WiFi-Mesh is proprielary

by using only ESP-NOW from each end node to the always-listening gateway, which forwards messages to the internet (via Ethernet).

Another highly useful network topology is called a mesh network. In a mesh network, every node is always on. Messages get forwarded through the most direct route to an end destination. Meshes have more redundancy; if a single node fails, the rest of the system can keep working by changing the delivery route. The short-range version of a mesh network is relatively easy to put together using the *painlessMesh* library or an ESP-WIFI-MESH network with Espressif boards, all of whose source codes are easily found online.

For long-range projects, however, Wi-Fi is not a viable solution. Our favorite approach uses LoRa — a communication protocol with a low data rate and low power consumption, but the potential to have several kilometers of reach. LoRa has a very low data rate and is only half-duplex (most transmitters can't send and receive at the same time), which makes mesh networks challenging.

There are several other potential communication methods, as detailed in the **Wireless Communication Comparison** text box. Please note that we did not test every method, mainly getting information from the internet before choosing what to work with. We did compile our own data for LoRa, ESP-WIFI-MESH, and ESP-NOW. Other solutions are harder to work with, but are applicable to more formal projects.

For the rocket-tracking/data-logging system, we used LoRa- instead of Wi-Fi-based communication, as we needed stablity and long range. For LoRa peripherals, we recommend either the Hope RFM95W or the Reyax RYLR406. The Reyax (UART-based) is very user-friendly, coming with a good antenna and sending data with relatively simple AT commands. It doesn't have as much pre-made network structure support, but works perfectly for simple point-to-point communication. Alternatively, the HopeRF board (using SPI) has a lot more support with *radiolib* or the *radiohead* library, which have implementations for (slightly shaky) ad hoc mesh networks.

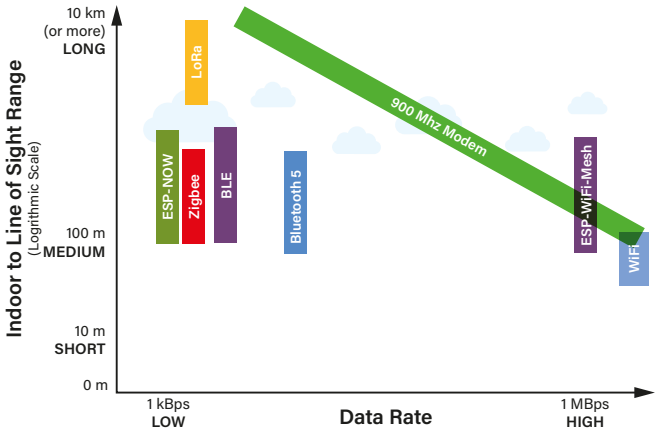
We used a mesh topology for the rocket systems, as, that way, we could launch multiple rockets and have those in the air getting broadcasts from the grounded ones, and then transmitting the locations of every rocket to our base station. Because we weren't launching more than three rockets at a time, LoRa's slow speed and inability to transmit while listening wasn't detrimental to our goals. However, for larger-scale projects, it would be good to look into using LoRaWAN or other large hub-and-spoke networks that support full-duplex.

We tried to get as much relevant beginner information into this admittedly brief article as possible. For more detailed information on network structures, the basics of using Espressif systems, etc., watch the video of our talk, *DevCon23 — From Rockets to Cellos: Real-world Applications of ESP32 Series and Dev Board Variants*, on the Espressif website or on YouTube [1].

230627-01

Wireless Communication Comparison

Technology	Data rate (Kbps)	Indoor (line-of-sight) range
Wi-Fi 2.4 GHz (≠ 5.2 GHz)	9,000	25 m (100 m)
ESP-WiFi-MESH	1,000	50 m (200 m)
Bluetooth 5 (audio)	260	50 m (240 m)
Bluetooth LE (low bit rate)	120	70 m (500 m)
Zigbee 2.4 Ghz	30	70 m (250 m)
LoRa 915 MHz	30	1.2 km (20 km)
ESP-NOW	1	70 m (500 m)
900 MHz modem	25 to 7,000	50 m (45 km)



WEB LINK

[1] "DevCon23 - From Rockets to Cellos: Real-world Applications of ESP32 Series and Dev Board Variants": https://youtu.be/jxPUkmaYp2c

ARDUINO CLOUD

What Arduino Cloud is

Develop from anywhere

- **NO CODE**
  With ready-to-use templates
- **LOW-CODE**
  Automatically generated sketches
- **FULL ARDUINO EXPERIENCE**
  Either offline with the UDE2 or online with the Cloud Editor
- **STORE YOUR SKETCHES ONLINE**
  Use your code in your favourite Arduino development environment

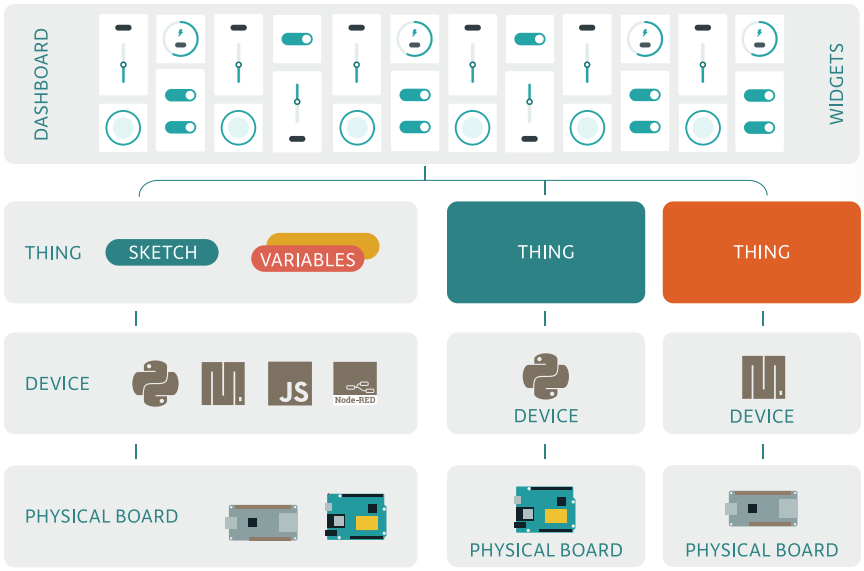
Program/Deploy

- **CABLE**
  Traditional USB programming
- **OVER-THE-AIR (OTA) UPDATES**
  Deploy your firmware wirelessly to your devices
- **MASS SCALE & AUTOMATION**
  With the Arduino Cloud CLI

Monitor & Control

- **CUSTOM DASHBOARDS**
  Using drag and drop widget
- **INSIGHTFUL WIDGETS**
  Interact with the devices and get real-time and historical data with dozens of widgets
- **MOBILE APP**
  Visualise your data in real-time from your phone with the IoT remote app

How does it work?



Compatible hardware

WITHIN ARDUINO DEVELOPMENT ENVIRONMENTS

ARDUINO
Cloud Applications can be developed using the Arduino Cloud Editor or Arduino IDE 2.

ESP32/ESP8266
+70% Of Arduino Cloud active users use ESP-based boards.

OUTSIDE ARDUINO DEVELOPMENT ENVIRONMENTS

Use your favourite programming environment and language to connect your devices to the Cloud.

Third party platform integration

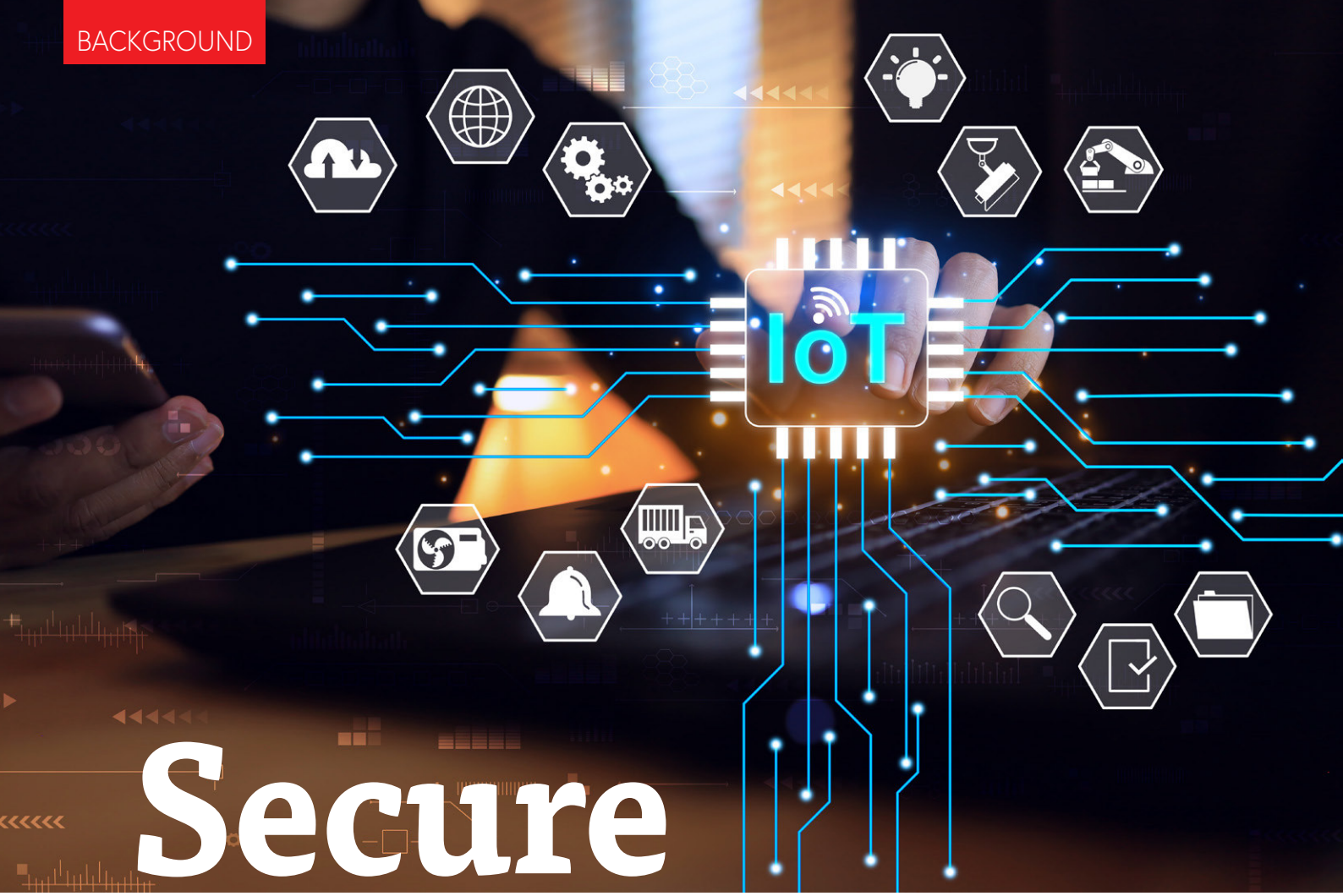
TRIGGER ACTIONS ON THIRD PARTY PLATFORMS

Connect your Arduino Cloud devices to external platforms such as IFTTT, Zapier and Google Services using webhooks and unlock endless possibilities.

Seamlessly integrate your IoT devices with over 2 000 apps, enabling tasks like recieving phone notifications, automating social media updates, streamlining data logging to external files, creating calendar events, or sending e-mail alerts.

Get 30% off on the yearly Maker plan with code ELEKTOR30*

cloud.arduino.cc/elektor



Secure IoT Manufacturing

Why and How

By Aditya Patwardhan, Espressif

The increased intelligence of the devices at the edge, and particularly, their continuous evolution, has required higher levels of security and constant dynamic adaptation by manufacturers. Here we see how Secure IoT Manufacturing of Espressif responds to these changing needs.

The internet of things (IoT) has seamlessly integrated into our daily lives. Thanks to advancements in sophisticated development frameworks, creating new IoT products has become notably more accessible. As the IoT industry continually evolves with ongoing innovations, the applications are getting more complex. One important aspect of this evolution is that people are becoming more concerned about security. In the market, we now have stricter security standards, each with its own requirements. This has made the manufacturing process more complicated, requiring a thorough security check at every step.

In this article, we shall go over several aspects of secure IoT manufacturing. We will discuss what issues are faced in each step and how we at Espressif make it easy for our customers.

Device Secrets and Unique Per-Device Data

The older MCU-based device manufacturing processes were relatively straightforward. After flashing the firmware onto the MCU and subjecting it to a series of quality assurance tests, the device was ready for deployment in the field. However, the landscape evolved significantly when connectivity came into the picture. Nowadays, most IoT products demand more than just a unique MAC address; they require individualized credentials unique to each device. Some of these are secrets needed to communicate securely with remote servers, some are secrets used for secure onboarding of the device, and some are secrets specific to the MCU — such as the

secret used to encrypt the data-at-rest that's stored in the flash.

Cryptography based on public key infrastructure (PKI) plays an important role in device security. Typically, a device has a public key that directly or indirectly can authenticate the server it communicates with, a public key that can verify the authenticity of any new firmware that is being installed on the device, and a public-private key pair that forms a unique device certificate that is registered with the cloud server or central database to enable authentication of the device from the cloud or a client. Additionally, the device also needs to store sensitive information, such as Wi-Fi network credentials.

With this typical case, you can see that the following are the key security requirements:

1. Ensure that the public keys that are used to authenticate other entities are not tampered with.
2. Ensure that the device's private key is generated in a secure environment with a good quality random number generator.
3. Ensure that the device's private key is securely stored on it in such a way that it doesn't fall prey to attackers, as it forms the device's identity.
4. The device certificate is signed by an entity that is trusted to perform certificate signing.
5. Ensure that sensitive information, such as Wi-Fi credentials, is stored in flash memory in an encrypted format so that simple flash reads won't reveal this information.

This translates into more fundamental security requirements:

1. Ensure that the device executes only the trusted firmware so that it can't be programmed with malicious code that can leak sensitive data.
2. Ensure that each device can encrypt its flash memory; preferably with a unique encryption key so that even if this key is found, the entire fleet of the devices is not compromised. Hence, it is preferred to generate this unique encryption key randomly inside the chip, and not have it accessible outside the chip.
3. Ensure that the manufacturing process generates a private key on the chip, and that the private key is not accessible externally.
4. Ensure that the manufacturing process works with trusted hardware or a cloud service for certificate signing.

Complexity in Manufacturing

While modern microcontrollers, including the ESP32 series, provide the security features required to implement the above requirements, there is still great care that needs to be taken at the time of manufacturing. Typically, contract manufacturers where this type of device programming takes place might not have know-how of the best security practices, and the complexity of secure manufacturing can be overwhelming.

Espressif provides multiple solutions to simplify this.

Auto Secure Boot Enablement

Secure boot is the feature that ensures that the hardware authenticates the firmware that is being executed on the chip upon each boot. For that, the chip is locked with a public key that can verify the signature present in the signed firmware for authenticity. Espressif's ESP-IDF provides for enabling a secure boot in the second-stage bootloader. This ensures that the bootloader, when programmed on the chip and executed the first time, programs the one-time-programmable (OTP) memory with the public key that is used by the hardware for secure boot. The bootloader can also take care of disabling the debug and programming interfaces to lock the chip. This way, your manufacturing process doesn't have to take responsibility for programming the OTP correctly in order to secure the device. This can all be handled by the developers, who understand security better, than those who manufacture the device.

Figure 1 demonstrates the transitive trust model for secure boot.

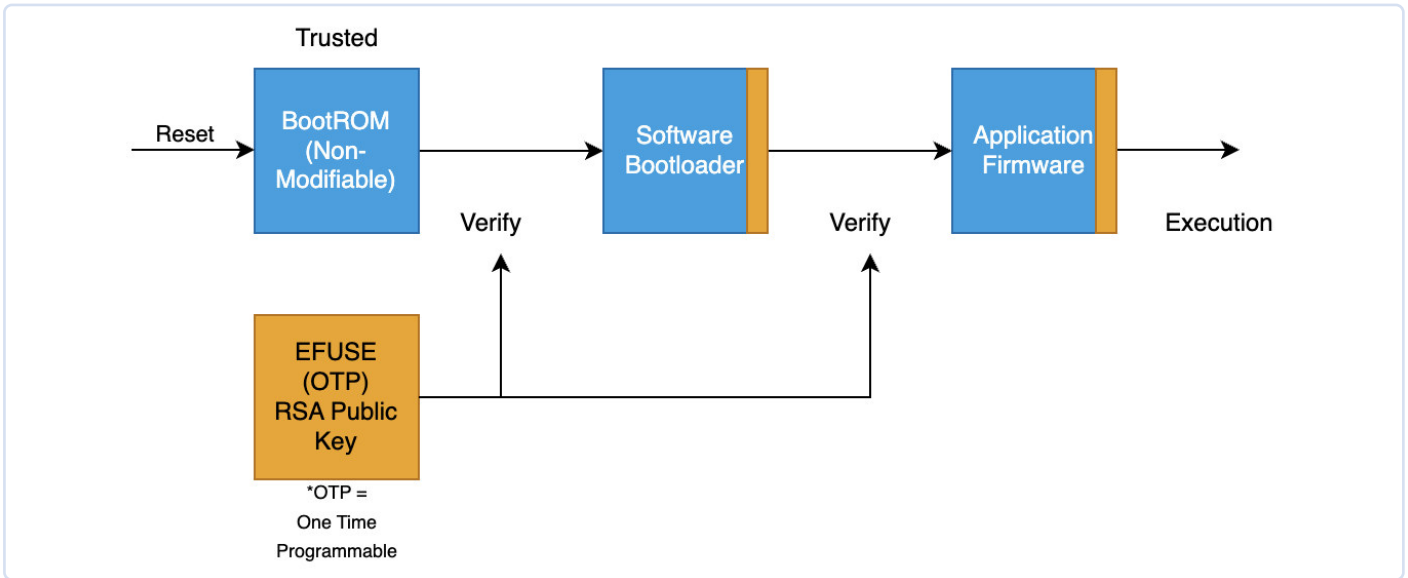


Figure 1: Diagram of the transitive trust model for secure boot.

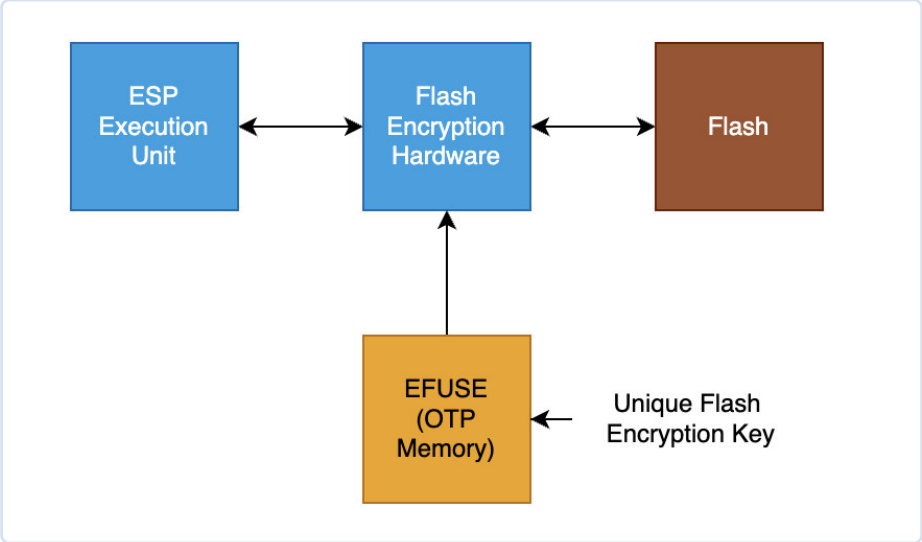


Figure 2: Diagram of a simplified version of flash encryption.

Once the secure boot feature is enabled, the chip can not only execute trusted firmware, but the trusted firmware can hold the various public keys mentioned above, and these public keys can't be modified forcefully.

Auto Flash Encryption Enablement

We discussed the requirement of having a unique, symmetric encryption key in the device, used to encrypt the flash memory. ESP32 series microcontrollers have a flash encryption feature whereby the hardware supports a "software non-readable" flash encryption key in the OTP. This key can transparently encrypt and decrypt flash contents dynamically. ESP-IDF's software bootloader provides runtime enablement of this feature, whereby upon the first boot of the device, the software bootloader can generate the flash encryption key in the chip using its true random number generator (TRNG), and programs it into the OTP memory with read-out protection enabled. The software bootloader can optionally also encrypt the firmware and other required flash contents that need to be encrypted upon first boot.

This way, the flash encryption, when enabled through the software bootloader, provides a way to generate a per-device random flash encryption key and enables flash encryption securely that can be used to protect any sensitive device data, such as Wi-Fi network credentials or the device certificate's private key.

Figure 2 shows a simplified version of flash encryption.

Secure Certificate Pre-Provisioning

For the device certificate, we discussed how it's important to protect private key generation and storage. It's also important to get the device certificate signed using a trusted entity — a certificate authority (CA). Espressif implements a secure certificate provisioning process in its factories, which provides a way for customers to order pre-provisioned modules.

In this module pre-provisioning process, there are three entities that work together. A provisioning host, which is a PC that works with an ESP32-series chip or module for provisioning. The provisioning host is connected to a local or cloud hardware security module (HSM), which holds the certificate authority and can sign any certificate without leaking the private key used for this signing. Most of the HSMs are also capable of generating non-repudiable logs that can provide assurance of how many certificates are signed by the HSM.

The process is shown in **Figure 3**.

The device certificate's private key is generated on-chip and never leaves it. The host provides certificate parameters such as validity time, common name, etc., and receives the certificate signing request (CSR). The CSR is then sent to the local or cloud HSM to get a

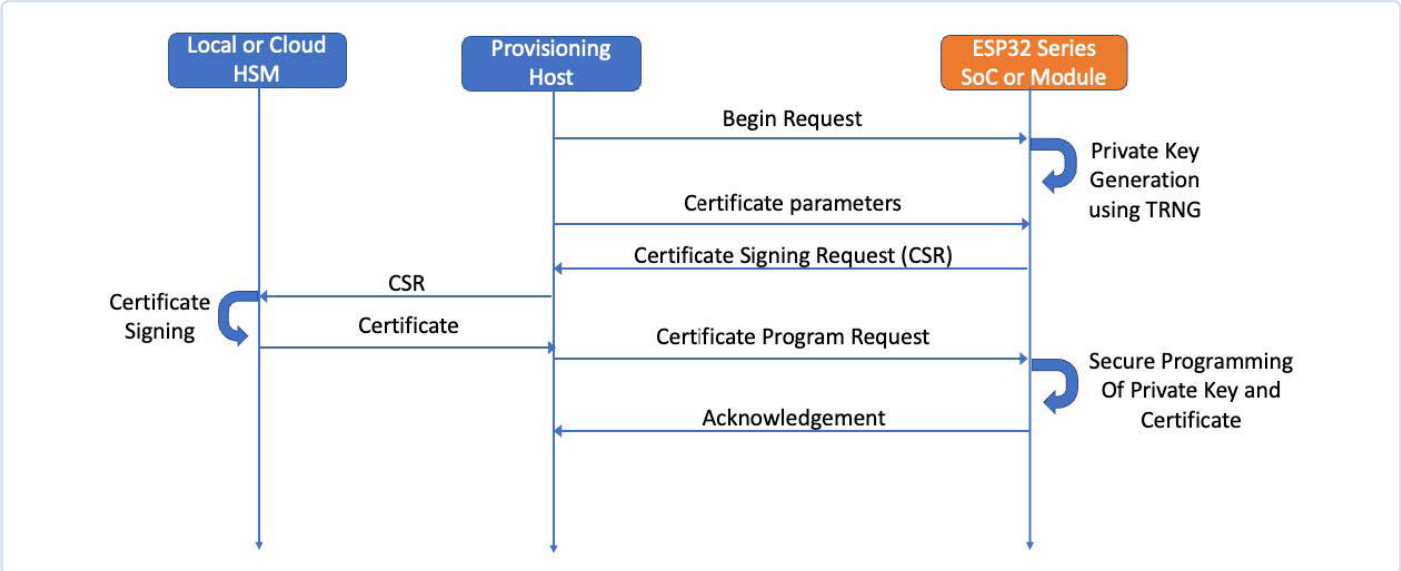


Figure 3: Flow diagram of the secure certificate pre-provisioning process.

signed certificate. This certificate then is sent to the device, and the device is then locked using secure boot and flash encryption. Many of the new ESP32-series chips also have a dedicated "Digital Signature" (DS) peripheral, which protects the device private key with a special hardware block. This DS peripheral can provide certificate operations directly with the encrypted private key, ensuring that the plaintext private key is inaccessible to the software, either.

Other Per-Device Unique Data

Beyond these security artifacts, the device may also need to have other per-device unique data, such as device ID. Espressif provides easy-to-use tools to generate per-device data binaries from a CSV file that has unique per-device data in tabular form. Please check out [1].

About the Author

Aditya Patwardhan is a Software Engineer at Espressif with over four years of experience. His areas of interest span systems, security, machine learning, and the exciting world of IoT. Aditya is deeply passionate about staying up to date with new developments in the security domain, and leveraging these developments to create robust and secure IoT applications.

Everything Comes Together

Security requirements and per-device unique data programming make IoT device manufacturing complex. Espressif offers great flexibility in the manufacturing process, which helps customers enable security features such as secure boot and flash encryption, in the Espressif manufacturing line without compromising security. Espressif's module pre-provisioning process enables secure device certificate programming. Espressif is also a Connectivity Standards Alliance (CSA) approved Product Attestation Authority, and has the ability to pre-provision the chips and modules with the Device Attestation Certificates (DAC) required for building Matter-compliant devices. In addition, Espressif also assists in custom firmware programming and unique data preprogramming, which greatly simplifies the manufacturing of IoT devices.

230638-01

WEB LINK

[1] Manufacturing utility: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/mass_mfg.html

A Simpler and More Convenient Life

An Amateur Project Based on the Espressif ESP8266 Module

Contributed by Transfer Multisort Elektronik Sp. z.o.o.

With each passing year, the concept of SmartHome is becoming increasingly popular, and the availability of solutions that help us manage our living space more efficiently is growing. In addition, some products that appear on the market offer compatibility with older devices, thanks to which we can use the existing equipment together with the latest technological advancements. Remote control of home appliances and the automation of various processes helps improve energy efficiency, protect the environment, increase our comfort and save money. The Smart ESP8266 remote project developed for a contest held by TechMasterEvent combines all these advantages.

Espressif is a manufacturer of well-received SoC integrated circuits and wireless transmission modules, many of which are available at TME. Thanks to their compact size and low-energy consumption, the products from Espressif can be successfully used both in consumer and industrial electronics.

Below, you can read about a device based on the ESP8266 module. It is an amateur project created by a participant of a TechMasterEvent contest. The entrants were asked to design an electronics project which seeks to make life easier.

You can find the ESP8266 module as well as several other components which may come in handy when you build your IoT projects (single-board computers, communication and memory modules, displays and much more) at [1].

ESP8266, IR LED and IR receiver

Smart ESP8266 remote is a project that aims to make controlling your home devices a breeze. With the use of an ESP8266, IR LED and IR receiver, this project eliminates the need for multiple remotes for different devices such as air conditioners or televisions. The project connects to a phone app, allowing users to easily send commands to their devices and even save the signals sent by their current remotes for future use.

In addition to its convenience and ease of use, the Smart ESP8266 remote is also a great solution for older devices that may not be compatible with traditional smart home technology. With the ability to read and save signals from traditional remotes, the Smart ESP8266 remote allows you to control older devices that may not have the capability to connect to the Internet or other smart home systems. This makes it a cost-effective alternative to upgrading your devices or purchasing expensive smart home devices.

The IR LED and IR receiver are used to transmit and receive IR signals respectively, which are used to control the household devices. The project can read and save signals from traditional remotes, allowing the user to control older devices that may not have the capability to connect to the internet or other smart home systems.

In addition to the hardware components, the Smart ESP8266 remote project also requires software to function, which you will find at [2].

The Smart ESP8266 remote offers several benefits such as convenience and ease of use, cost-effectiveness and flexibility. It eliminates the need to purchase expensive smart home devices or upgrading older devices, making it a cost-effective alternative. The project is also flexible enough to be easily modified or customized to work with different devices and different IR protocols, making it a versatile solution for controlling different devices with a single app.

230656-01

WEB LINKS

[1] TME shop: https://tme.eu
[2] Source code for this project: https://techmasterevent.com/project/how-to-make-old-devices-smarter-with-a-esp8266

MACNICA
ATD EUROPE

Your official authorized distributor
in Europe for Espressif Systems

ESPRESSIF

Macnica ATD Europe
+49 (0)89 899 143-11
sales.mae@macnica.com
www.macnica-atd-europe.com

Blynk

How to Build IoT Apps without Software Expertise

With Blynk IoT Platform and Espressif Hardware

Contributed by Blynk Inc.

What if you could develop a mobile app without writing a line of code, brand it, and publish to app stores within a month? Launch production-grade IoT software without hiring software engineers? With Blynk IoT it's possible within a month, not years!

Blynk firmware library supports:
- ESP32
- ESP32-S2
- ESP32-S3
- ESP32-C3
- ESP8266
- and others

What is included in Blynk IoT?

Blynk is a low-code IoT software platform featuring cloud, firmware libraries, no-code native mobile app builder, and a web console to manage it all. You get Wi-Fi device provisioning, data visualization, automations, notifications, OTA updates, and a robust user and device management system [1].

Blynk App Builder for iOS and Android

Enables the creation of quick prototypes and fully functional standalone apps without coding skills. In constructor mode, you can choose from 50+ customizable UI elements like buttons, sliders, charts, maps, gauges, etc., and drag and drop them to the canvas to create a custom UI for your connected product. You can set up multiple app pages, use various interactions, customize images, fonts, colors, and icons to make your app unique.

Web Dashboard Builder

It has similar architecture for creating historical and real-time data visualizations, and for controlling and monitoring devices using pre-built UI elements. The cool thing is you can build independent interfaces for mobile and web, based on your needs.

Figure 1: No-Code Interfaces created with Blynk.

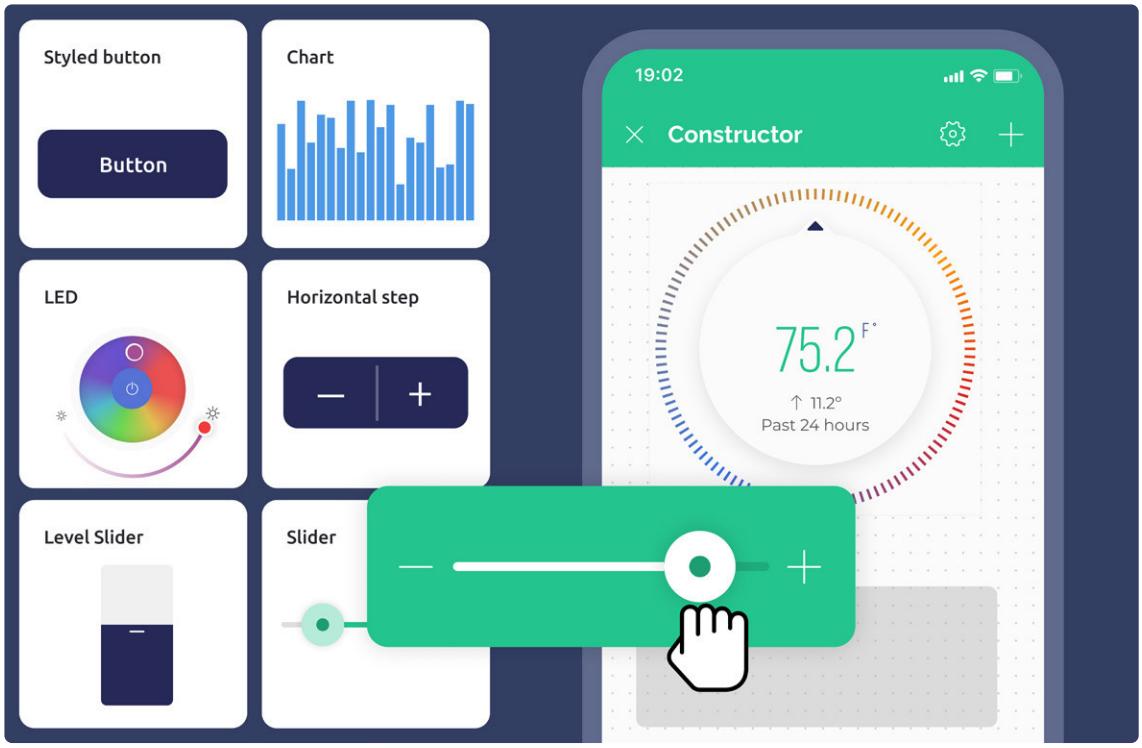


Figure 2: Blynk Drag-n-Drop App Builder.

Advanced User Management System

It helps keep everything structured, even at enterprise scale. You can create a multi-level organization structure and manage devices and user's roles, permissions, passwords, and much more.

Built-in Device Lifecycle Management

The functionality covers all needs related to token management, Wi-Fi provisioning with dynamic token generation, adding devices, and assigning them to users. It offers reliable and secure OTA firmware updates managed in a simple interface.

No-Code Automation Scenarios

Can be set based on date, time of the day, user actions, or device state. You can notify users about important events on the hardware via pushes, in-apps, emails, or SMS.

How to Connect Your ESP to Blynk? What's the Integration Effort?

Depending on your hardware setup, you can go one of the two routes for connecting your Espressif device. Both enable all the Blynk IoT features out of the box, including Wi-Fi provisioning, OTA, and secure connection to Blynk Cloud. Choose *Blynk.Edgent* [3] for single-MCU devices. If you're offloading connectivity to a secondary MCU, go with *Blynk.NCP* [4][5].

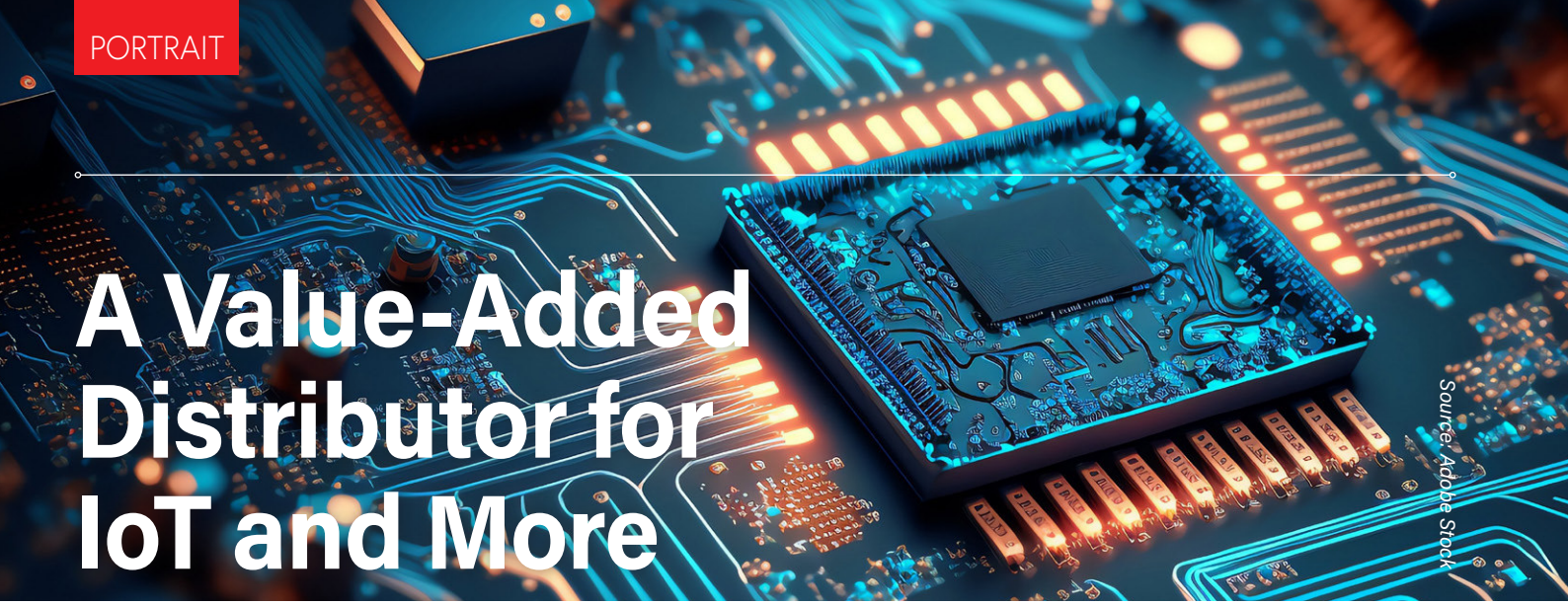
Both routes require minimal implementation effort, with code examples provided by Blynk. For the dual-MCU setup, it's a ready binary for the NCP and a lightweight library for the primary MCU communicating with the Network Co-Processor over the UART interface.

Your journey from device setup to full-scale IoT infrastructure and app launch can take just weeks [6].

230659-01

WEB LINKS

[1] Official website: https://bit.ly/blynk-iot
[2] Blynk.Console — create your free account: https://bit.ly/blynk-cloud
[3] Blynk.Edgent documentation: https://bit.ly/doc-edgent
[4] Blynk.NCP documentation: https://bit.ly/doc-ncp
[5] What is Blynk.NCP: https://bit.ly/blynk-ncp
[6] Ready-made weather station project to play around: https://bit.ly/weather-blueprint



A Value-Added Distributor for IoT and More

Source: Adobe Stock

Contributed by Steliau Technology

Steliau Technology is an innovative company specializing in electronic solutions. The company stands out for its engineering expertise and passion for innovation. Steliau Technology, through its partner Espressif, provides essential electronic components for wireless connectivity and IoT, such as the ESP32-C5, ESP32-C6, ESP32-P4, ESP32-S6 and many more products.

Founded in 2018, Steliau Technology [1] defines itself as a value-added distributor of electronic solutions. Human-Machine Interfaces, screens and touch solutions, connectivity & IoT are all areas of expertise largely mastered by Steliau and which give it an already well-established reputation in the electronics market.

Steliau Technology is well-known for its strategic partnerships with leading electronics companies, enabling it to strengthen its position in the fields of IoT and connectivity. Espressif and Steliau Technology share a long-standing partnership, as the first distributor, consolidated over the years. As the official distributor for France and Italy, Steliau Technology is the one-stop shop for Espressif solutions.

Steliau Technology is perfectly equipped to offer full support for the entire range of Espressif products, both in terms of hardware and embedded software. The team has in-depth technical expertise covering all aspects of connectivity, from hardware design to software programming. This means that Steliau Technology is able to provide full support to customers, ensuring robust and high-performance connectivity solutions.

This strong partnership guarantees our customers privileged access to the best connectivity solutions on the market, such as the latest Espressif generations: ESP32-C5, ESP32-C6, ESP32-P4, ESP32-S6.

Steliau Technology through its partner Espressif provides essential electronic components for wireless connectivity and IoT in a variety of markets and sectors, contributing to the constant evolution of the technology.

IoT Solutions and More

First of all, in the field of the Internet of Things (IoT), Espressif's Wi-Fi and Bluetooth microcontrollers are widely used. These components are vital for smart home applications such as connected thermostats, security cameras and lighting control devices. They play a leading role in interoperability in connected home products with compatible Matter solutions (through WiFi and Thread in particular). Espressif's solutions are also used in the industrial sector for remote monitoring, data collection, and machine control. In addition, Espressif's products are present in the healthcare sector, where they power connected medical devices and fitness tracking devices.

As a global electronics partner, Steliau has the ability to offer solutions integrating Espressif's products for touchscreen control. Thanks to its expertise, Steliau Technology is able to design integrated solutions in which Espressif's products control the touch screen, with a number of success stories to our credit, particularly for screen sizes up to 7 inches. Our ability to offer a global solution is reinforced by specific technical support covering all these areas.

230661-01

Any requests?

Steliau is available to assist customers with any queries they may have. Please contact remi.krief@steliau-technology.com for any request about Espressif solutions.

WEB LINK

[1] Steliau Technology: https://steliau-technology.com/en

The Next Era of **Microcontrollers**

ESPRESSIF

High Performance MCU

With RISC-V Dual-Core Upto 400MHz

AI Acceleation

High-Speed **MEMORY**

Powerful **IMAGE & VOICE** Processing Capabilities

HMI Capabilities

- MIPI-CSI with ISP
- MIPI-DSI - 1080P
- Capacitive Touch
- H.264 Encoding - 1080P@30fps
- Pixel Processing Accellerator

Best-in-Class Security

- Cryptographic Accelerators
- Secure Boot, Flash Encryption
- Private Key protection
- Access Controls

ESP32-P4

Connectivity

- USB2.0 High Speed
- Ethernet
- SPI
- SDIO3.0
- UART
- I2C, I2S

.....


IP Camera


Touch Panel


Video Door bell


Robotic Control


Industrial Robot

www.espressif.com

Learn More About **ESP SoCs**

Quick & Easy IoT Development with M5Stack

Contributed by M5Stack

As a world-renowned modular IoT development platform based on ESP32, M5Stack builds hundreds of controllers, sensors, actuators and communication modules in modularized style that can be connected via standard interfaces. By stacking modules with different functionalities, users can accelerate product verification and development.

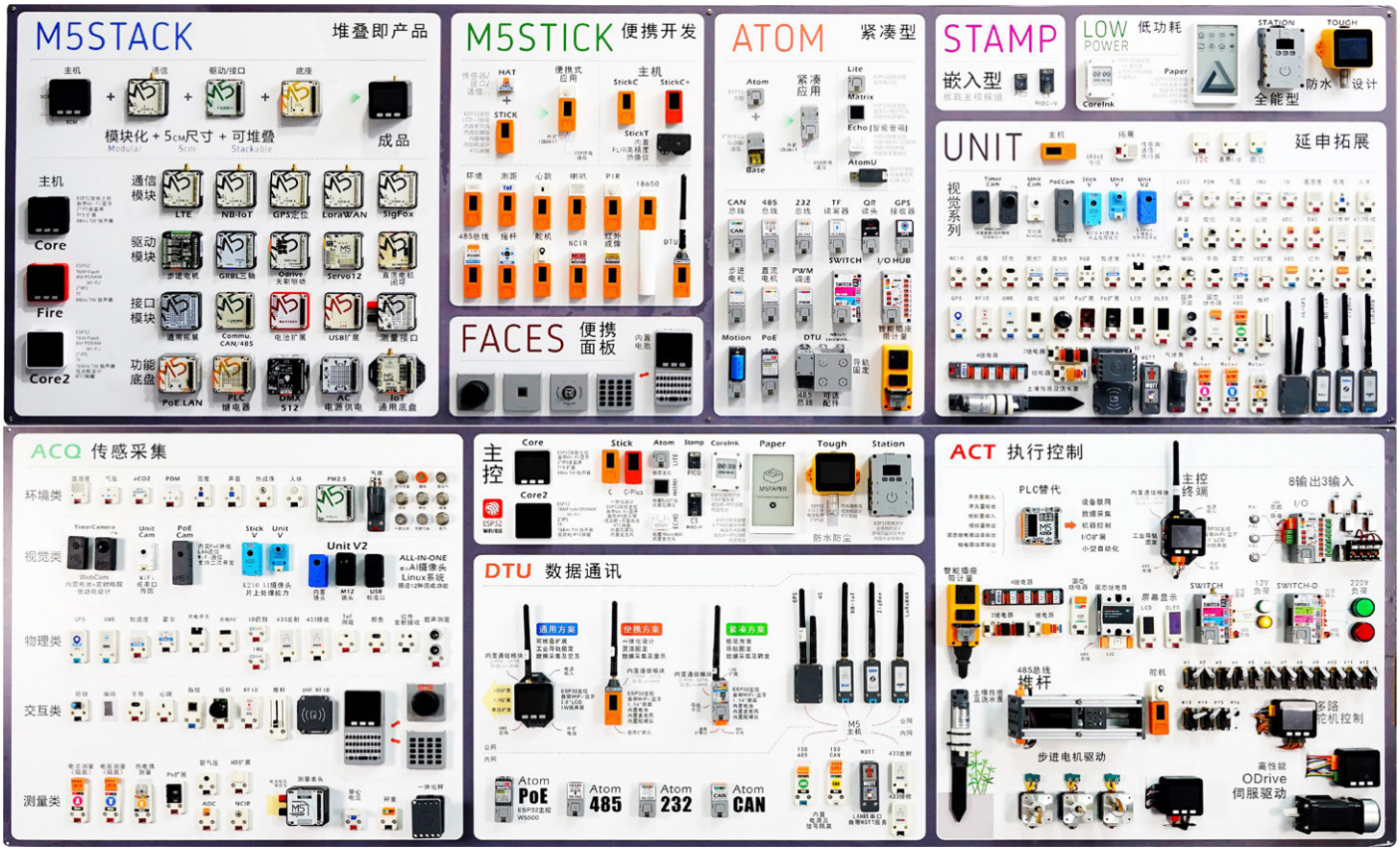


Figure 1: M5Stack Eco-system Family.

Figure 2: M5Dial is suitable for Smart Home.

The M5Stack modules (**Figure 1**) [1] can be plugged and played with the UIFlow low-code graphical programming IDE to provide the best experience for prototyping IoT projects, from entry-level hobbyists to professional developers.

With Stackable hardware modules and a user-friendly graphical programming platform, M5Stack provides clients in Industrial IoT, Home Automation, Smart Retail, Smart Agriculture and STEM education, with efficient and reliable Quick & Easy IoT Development experience.

New: The M5Dial

The recently launched M5Dial [2] is a highly suitable product for Smart Home. As a versatile embedded development board, M5Dial integrates various functionalities and sensors required for Smart Home control (**Figure 2**).

If you are a fan of ESP32, then M5Stack is a must-have!

The main controller of M5Dial is M5StampS3, a microcontroller based on the ESP32-S3 chip, known for its high performance and low-power consumption. It supports Wi-Fi and Bluetooth communication, as well as multiple peripheral interfaces such as SPI, I2C, UART, ADC, and more. M5StampS3 also comes with 8 MB of built-in flash, providing sufficient storage space for users.

M5Dial features a 1.28-inch circular TFT touch screen, a rotary encoder, an RFID detection module, an RTC circuit, a buzzer, physical buttons, and other functionalities, enabling users to easily implement various projects.

The standout feature of M5Dial is its rotary encoder, which accurately records the position and direction of the knob, providing users with an enhanced interactive experience. Users can adjust settings such as volume, brightness, menus, or control home appliances like lights, air conditioning, curtains, etc., using the rotary knob. The built-in display screen of the device can also show different interactive colors and effects. With its compact size of 45 mm × 45 mm × 32.2 mm and light weight of 46.6 g, M5Dial is easy to implement.

Whether it's used to control household appliances in Smart Home or to monitor and control systems in industrial automation, M5Dial can be easily integrated to provide smart control and interactive functionalities.

230662-01

WEB LINKS

[1] The Innovator of Modular IoT Development Platform | M5Stack: https://m5stack.com/
[2] ESP32-S3 Smart Rotary Knob w/ 1.28" Round Touch Screen:
https://shop.m5stack.com/products/m5stack-dial-esp32-s3-smart-rotary-knob-w-1-28-round-touch-screen

Building a Smart User Interface on ESP32

Contributed by Slint

Smartphones have redefined the user experience of touch-based user interfaces (UIs). Building a modern smart UI necessitates the use of modern graphical libraries and tools. In this article, we'll share tips and showcase Slint, a toolkit for creating interactive UIs that meet and exceed user expectations.

Figure 1: C++ and Rust logos.

Slint is a next-generation toolkit for building native graphical UIs in C++, Rust, and JavaScript, with a broad cross-platform support, including bare metal, RTOSs, and embedded Linux. On GitHub, Slint has more than 10.000 stars.

Choose a Programming Language — C/C++ or Rust

In embedded programming, C/C++ have been the favorite programming languages for a long time. But Rust, known for its memory safety and performance, is becoming popular among embedded developers.

Slint, the only toolkit to provide native APIs for both C++ and Rust (**Figure 1**), offers developers the choice: Write your business logic in either language. Furthermore, it provides a transition path for those interested in moving from their code from C/C++ to Rust.

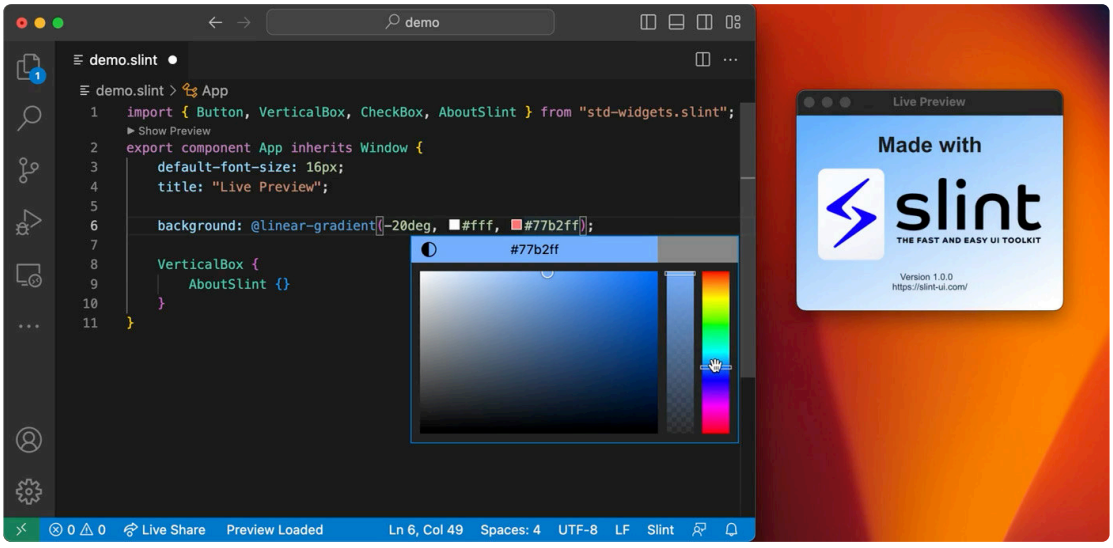


Figure 2: Quick iterations with Slint's Live-Preview.

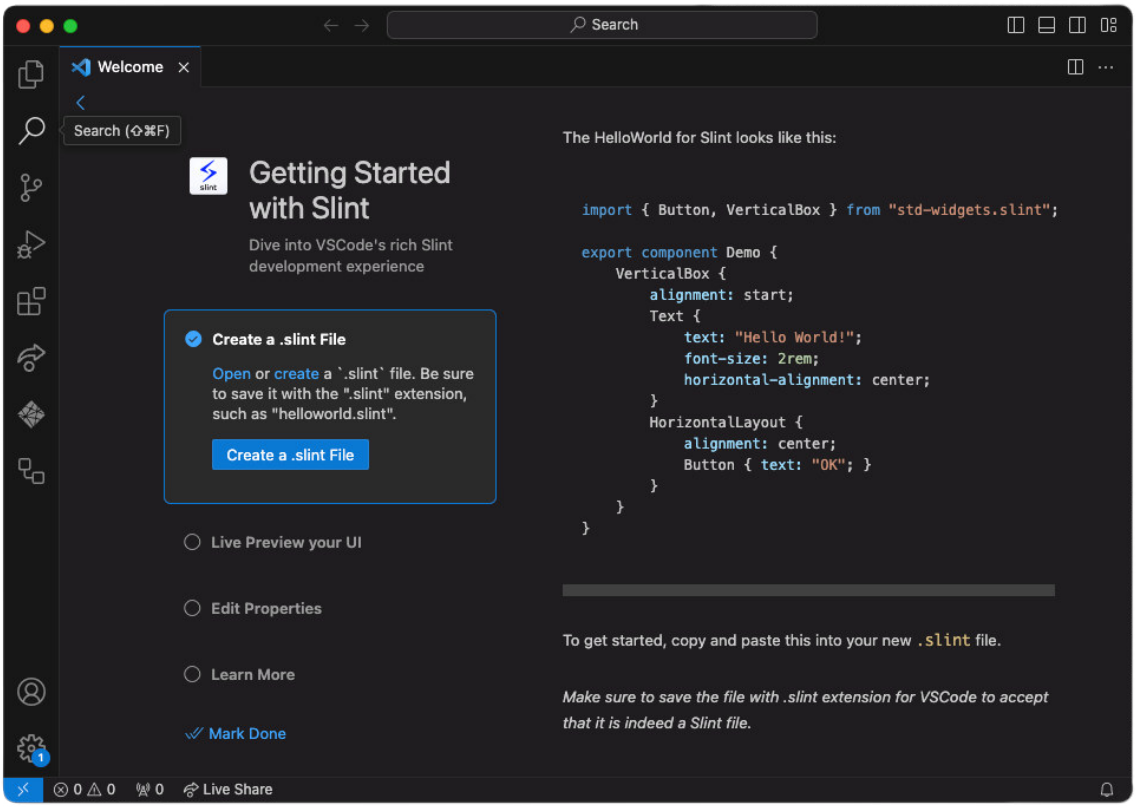


Figure 3: Getting started with Slint.

Separate UI from Business Logic

Common patterns like MVC or MVVM promote separating business logic from the UI to enhance efficiency and code quality.

In Slint, the UI is defined using a language akin to HTML/CSS, promoting a strict division between presentation and business logic. Complete your UI design through quick iterations with Slint's Live-Preview (**Figure 2**).

Enjoy a Good Developer Experience (DX)

Today's complexity in software development requires a good DX: Developers build with confidence, drive greater impact, and feel satisfied.

You can keep using your favorite IDE. Choose between Slint's VS code extension (**Figure 3**) and the generic language server: Enjoy code completion, syntax highlighting, diagnostics, live-preview, and more. Additionally, Slint offers an ESP-IDF component, simplifying its integration with the Espressif IoT Development Framework (IDF).

Deliver an Exceptional User Experience (UX)

UI performance is critical for an exceptional UX. Enjoy flexibility in hardware design with Slint's line-by-line and framebuffer rendering capabilities on the ESP32 platform, ensuring a more versatile approach to device development (**Figure 4**).

To get started with Slint on ESP32, visit [1].

230670-01

Figure 4: A Slint demo on ESP32.

WEB LINK

[1] Slint on ESP32: https://slint.dev/esp32

Get your hands on new ESPRESSIF Hardware!

There is nothing that excites us more than getting our hands on new hardware, and so this collaboration with Espressif has been a treat! Want to experience the real deal yourself? Elektor has stocked up the stores to accommodate all products that are featured in this edition!

ESP32-S3-Box-3

ESP32-S3-BOX-3 is a fully open-source AIoT development kit based on the powerful ESP32-S3 AI SoC, and is designed to revolutionize the field of traditional development boards. ESP32-S3-BOX-3 comes packed with a rich set of add-ons, empowering developers to easily customize and expand this kit's functionality.

www.elektor.com/20627

ESP32-S3-Eye

The ESP32-S3-EYE is a small-sized AI development board. It is based on the ESP32-S3 SoC and ESP-WHO, Espressif's AI development framework. It features a 2-Megapixel camera, an LCD display, and a microphone, which are used for image recognition and audio processing.

www.elektor.com/20626

ESP32-S3-DevkitC-1

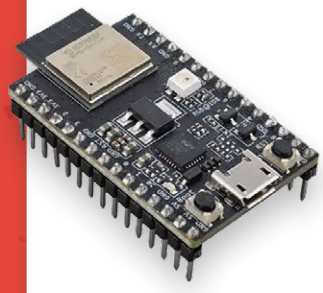
The ESP32-S3-DevKitC-1 is an entry-level development board equipped with ESP32-S3-WROOM-1, ESP32-S3-WROOM-1U, or ESP32-S3-WROOM-2, a general-purpose Wi-Fi + Bluetooth Low Energy MCU module that integrates complete Wi-Fi and Bluetooth LE functions.

www.elektor.com/20697

ESP32-Cam-CH340

The ESP32-Cam-CH340 development board can be widely used in various Internet of Things applications, such as home intelligent devices, industrial wireless control, wireless monitoring, QR wireless identification, wireless positioning system signals and other Internet of Things applications.

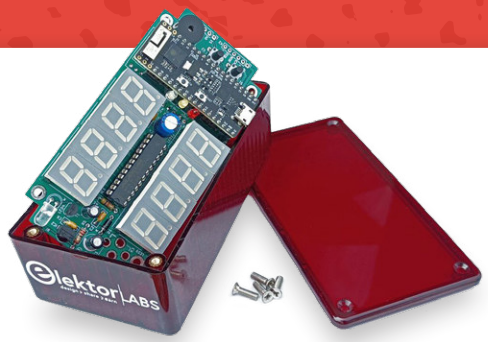
www.elektor.com/19333



ESP32-C3-DevKitM-1

ESP32-C3-DevKitM-1 is an entry-level development board based on ESP32-C3-MINI-1, a module named for its small size. This board integrates complete Wi-Fi and Bluetooth LE functions. Most of the I/O pins on the ESP32-C3-MINI-1 module are broken out to the pin headers on both sides of this board for easy interfacing. Developers can either connect peripherals with jumper wires or mount ESP32-C3-DevKitM-1 on a breadboard.

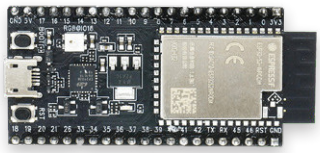
www.elektor.com/20324



Elektor Cloc 2.0 Kit

Cloc is an is an easy to build ESP32-based alarm clock that connects to a timeserver and controls radio & TV. It has a double 7-segment retro display with variable brightness. One display shows the current time, the other the alarm time.

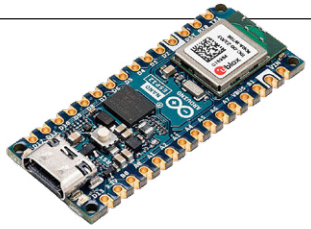
www.elektor.com/20438



ESP32-S2-Saola-1M

ESP32-S2-Saola-1M is a small-sized ESP32-S2 based development board. Most of the I/O pins are broken out to the pin headers on both sides for easy interfacing. Developers can either connect peripherals with jumper wires or mount ESP32-S2-Saola-1M on a breadboard. ESP32-S2-Saola-1M is equipped with the ESP32-S2-WROOM module, a powerful, generic Wi-Fi MCU module that has a rich set of peripherals.

www.elektor.com/19694



Arduino Nano ESP32

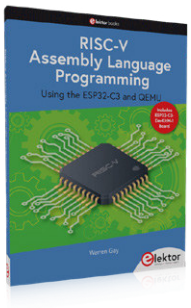
The Arduino Nano ESP32 is a Nano form factor board based on the ESP32-S3 (embedded in the NORA-W106-10B from u-blox). This is the first Arduino board to be based fully on an ESP32, and features Wi-Fi, Bluetooth LE, debugging via native USB in the Arduino IDE as well as low power.

www.elektor.com/20562

MakePython ESP32 Development Kit

The MakePython ESP32 Kit is an indispensable development kit for ESP32 MicroPython programming. Along with the MakePython ESP32 development board, the kit includes the basic electronic components and modules you need to begin programming. With the 46 projects in the enclosed book, you can tackle simple electronic projects with MicroPython on ESP32 and set up your own IoT projects.

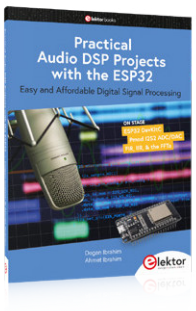
www.elektor.com/20137



RISC-V Assembly Language Programming using ESP32-C3 and QEMU (+ FREE ESP32 RISC-V Board)

The availability of the Espressif ESP32-C3 chip provides a way to get hands-on experience with RISC-V. The open sourced QEMU emulator adds a 64-bit experience in RISC-V under Linux. These are just two ways for the student and enthusiast alike to explore RISC-V in this book. The projects in this book are boiled down to the barest essentials to keep the assembly language concepts clear and simple.

www.elektor.com/20296



Practical Audio DSP Projects with the ESP32

The aim of this book is to teach the basic principles of Digital Signal Processing (DSP) and to introduce it from a practical point of view using the bare minimum of mathematics. Only the basic level of discrete-time systems theory is given, sufficient to implement DSP applications in real time. The practical implementations are described in real-time using the highly popular ESP32 DevKitC microcontroller development board.

www.elektor.com/20558

MicroPython for Microcontrollers

Powerful controllers such as the ESP32 offer excellent performance as well as Wi-Fi and Bluetooth functionality at an affordable price. With these features, the Maker scene has been taken by storm. Compared to other controllers, the ESP32 has a significantly larger flash and SRAM memory, as well as a much higher CPU speed. Due to these characteristics, the chip is not only suitable for classic C applications, but also for programming with MicroPython. This book introduces the application of modern one-chip systems.

www.elektor.com/19736