RASP PI TRICKS
TRY ARTIFICIAL INTELLIGENCE ON A PI

MakerSpace #04

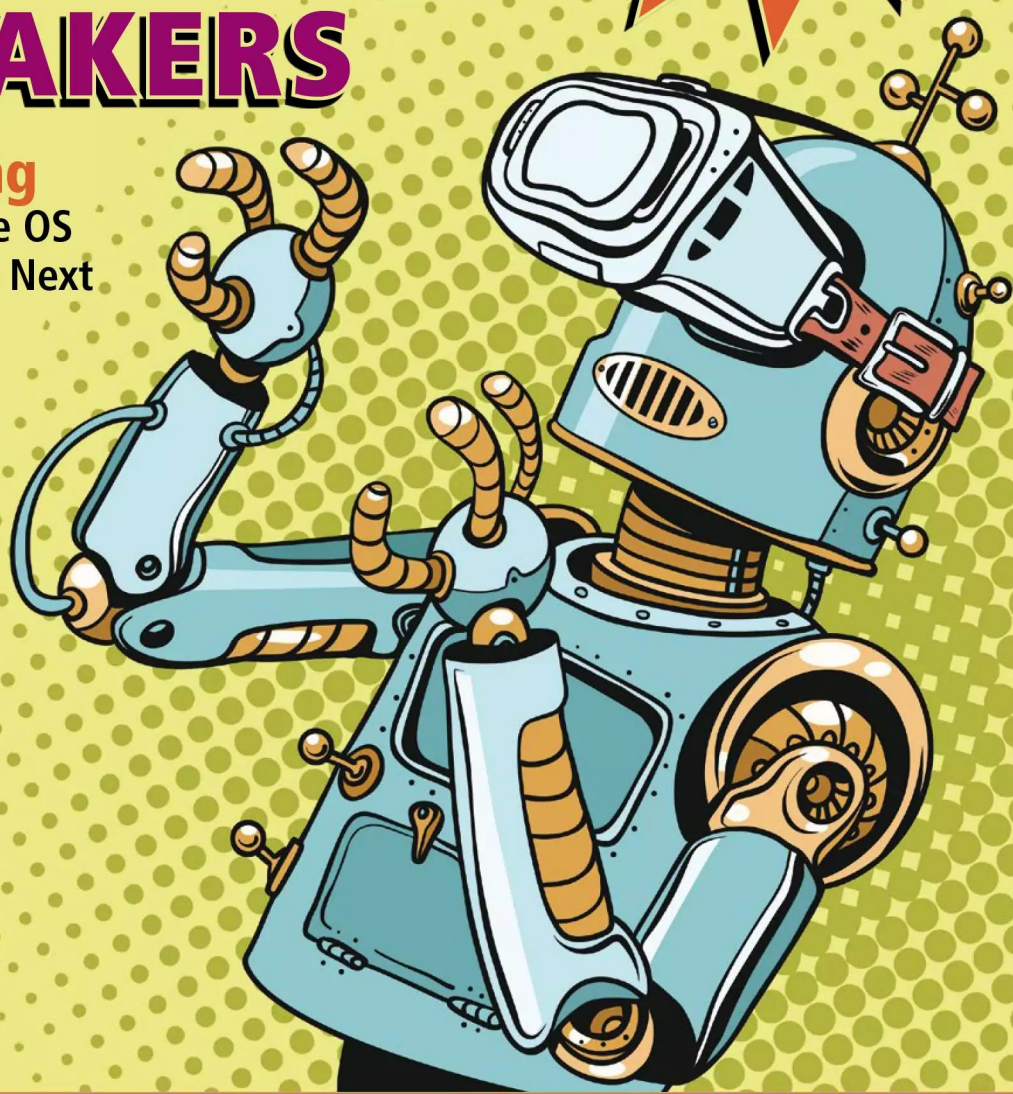
HANDS-ON PROJECTS FOR MAKERS

BUILD INFO DISPLAYS

Retro Computing
C64x with Commodore OS
FPGA-based Spectrum Next

Smart Home
Automate your house
Check plants and pets

Low-Code Programming
Learn Node-RED, then add Python



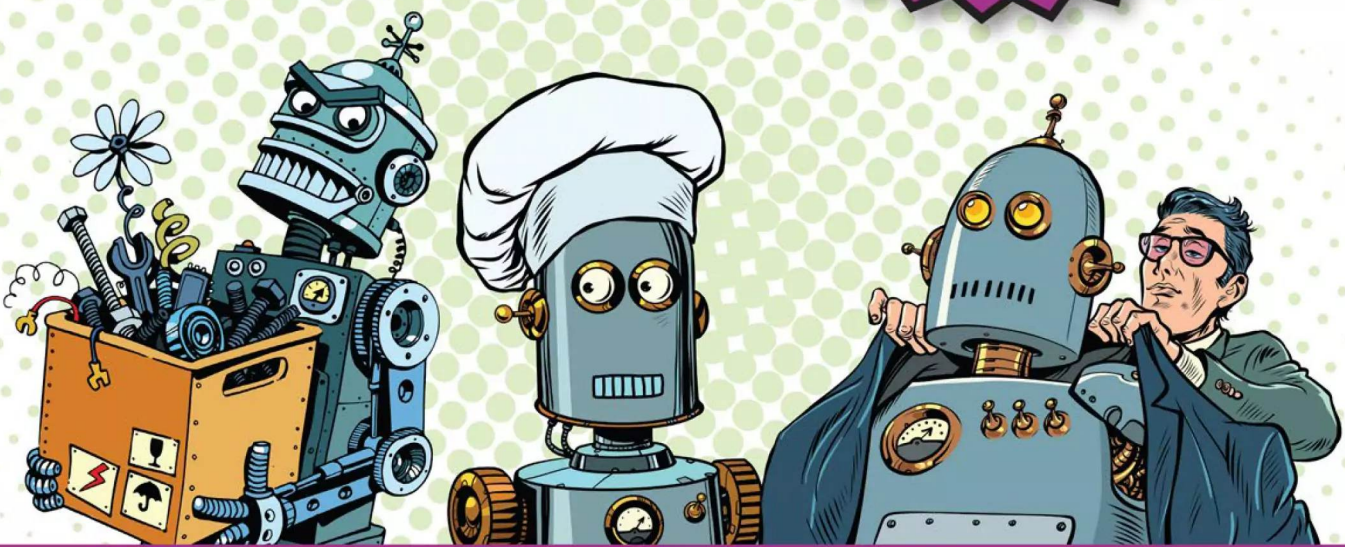
LINUX NEW MEDIA
The Pulse of Open Source

Turn your ideas into reality!

This is not your ordinary computer magazine! *MakerSpace* focuses on technology you can use to build your own stuff.

If you're interested in electronics but haven't had the time or the skills (yet), studying these maker projects might be the final kick to get you started.

Issues available now!



Lead Image © Valeriy Kachaev and studiostoks, 123RF.com

COLLECT ALL 3

ORDER ONLINE: shop.linuxnewmedia.com

Imagine It, Make It

By Hans-Georg Eßer

Welcome to *MakerSpace 4*, an all new collection of exciting, hands-on projects! If you're new to *MakerSpace*, we are a computer magazine that covers more than just the latest PCs or Apple computers – or Android or iOS smartphones and tablets for that matter. In this magazine, we talk about technology that you can use to build your own stuff and bring your ideas to life. If you don't have the time or the necessary skills, we hope looking at the projects in this magazine will inspire you to start a programming class, learn soldering, or work on some other skill.

Computing Can Be Fun

We use computers in the office; we use them at home. Every smartphone is a powerful computer. If you bought a TV in the last 10 years, it also will likely run apps. Computers are tools, and they provide entertainment, but in most cases only as a medium. A few decades ago, that was a different story. If you had a computer in the 1980s, you might remember how these machines used to fascinate us. Computers let us watch in awe when they "learned" a new feature: "Oh wow, it has stereo sound; you hear that?" "Oh, look how smooth that animation runs on the screen."

Today we expect that machines can do anything we can imagine. If there's something they can't do today, then it's just a question of time (for advancing the technology) and creativity (someone has to implement it) before it becomes a reality. Living in the 2020s, it's hard for us to become fascinated with technology because we're surrounded by it, and we're accustomed to using it all the time. Part of the problem is that modern devices are very complex. They are preconfigured at the factory, and we don't have to understand how to assemble them. We don't need to care about what kind of processors and other chips make them work.

Yet even today, it is possible to understand, to assemble, and to build. Thanks to small, cheap devices such as the Raspberry Pi and a huge family of microcontrollers, we can create systems that were impossible 15 years ago.

Projects

We've collected quite a few project ideas that we found interesting, and we hope you do as well. On page 8, we show you how to build a retro alarm clock for your bedroom with four shiny seven-segment LEDs, controlled by an STM32 microcontroller, to display the time like it's the 1980s again.

The next article uses the Raspberry Pi Pico (a different microcontroller) as the heart of a mobile e-ink information display (p. 14). Using a few Python commands, you can make it show the information that is relevant to you: weather forecasts, local temperatures, or free disk space on your personal server.

On pages 18 and 23, you'll find two camera-based projects. In the first article, we show you how to build a camera that uses four image sensors to create so-called lenticular images: By simultaneously shooting the same picture from several positions, you can later create 3D or motion effects. This article shows what you need to assemble the camera with a Raspberry Pi, and it also discusses several applications that let you transform the photos. The second article needs only one camera and lets you set up a VPN and livestream so that you can watch your pets while you're away (Figure 1).

If you need an image viewer that lets you use gestures to scroll, take a look at the "Hands Free" article (p. 26). As a good use case, imagine you're following a long recipe and you've dirtied your hands with the ingredients. With normal tech, you would have to first wash and dry your hands before scrolling the tablet screen to get to the next recipe step. With this project, that won't be necessary.

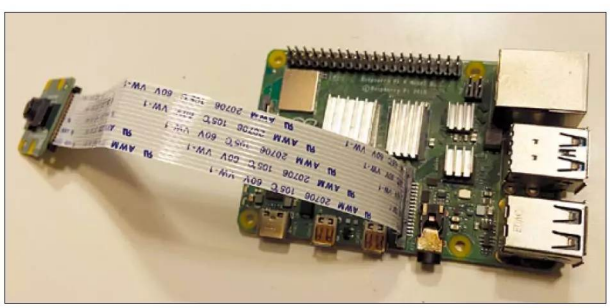


Figure 1: Add a camera module to your Raspberry Pi and check your pets' activities while you're gone.

Figure 2: Automate your greenhouse with a Raspberry Pi Pico W.

The next article (p. 29) lets you pilot an in-house drone. The DJI Ryze Tello is an educational tool that lets you program a steering application in Scratch or Python. Besides flying around and transmitting video, you can query its sensors to discover the height, temperature, and barometric pressure.

We close the section with an introduction to artificial intelligence (AI) on the Raspberry Pi (p. 32): Learn how to get a pre-trained object classification model and deploy it in TensorFlow Lite or OpenCV. You will need a Pi 4 (or better) for this since the older Pis aren't sufficiently powerful.

Automation

In the *Automation* section, we show you even more projects – but they're all about home automation! With ESPHome (p. 36), you can create your own home automation devices with a supported microcontroller board that you connect to LEDs, sensors, or switches. Use a Raspberry Pi Pico to control ventilation, heat, and windows in your greenhouse (p. 42, Figure 2), and add a RaspBee II module (p. 46) to your Raspberry Pi and have it talk to Zigbee devices all over your house. Finally, on page 50 we show you how to integrate the MQTT protocol into Home Assistant.

Retro Computing

If you've read earlier issues of *MakerSpace*, you know that we have a soft spot for retro computing projects and products: We celebrate revivals of the home computer technology of the 1980s. In this issue, we review the second edition of the FPGA-based Sinclair ZX Spectrum Next (p. 54, Figure 3), and we look at a beautiful Commodore C64-shaped computer case for Mini-ITX mainboards (p. 58) – both are successful Kickstarter projects with backers receiving their hardware a few months ago. On page 62, we introduce you to the BCPL programming language, C's ancient predecessor that was created in 1967.

Programming

The last section offers even more programming, but with current technology. Control an LED display with Go (p. 68); start learning to use Node-RED, a "low-code" programming platform (p. 74); and add Python scripts to your Node-RED projects so that you can use all of the Raspberry Pi's features (p. 81).

Another low-code tool, Snap4Arduino (p. 86), is an implementation of Scratch and provides a unique set of libraries that will upload and configure Arduino modules without requiring any Arduino knowledge. An alternative approach to programming these microcontrollers is Arduino CLI. If you prefer the command line over graphical development tools, this might be just the right tool for you.

Let's Get Excited

I hope you'll find the topics we've selected exciting. Perhaps you'll modify our suggestions and evolve them into something even better. If you do, we'd love to hear your "success stories." Take a few pictures, and tell us how you've modified our ideas. You can reach us at `edit@makerspace-magazine.com`. ■■■

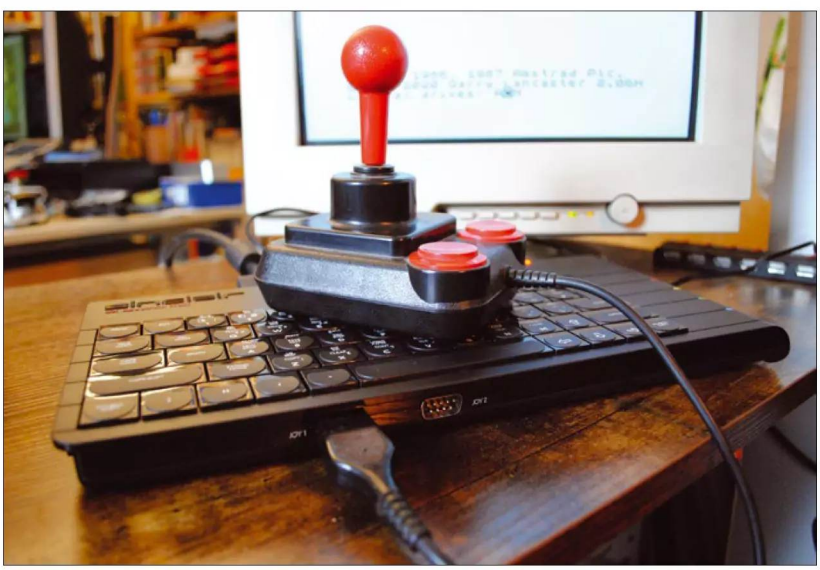
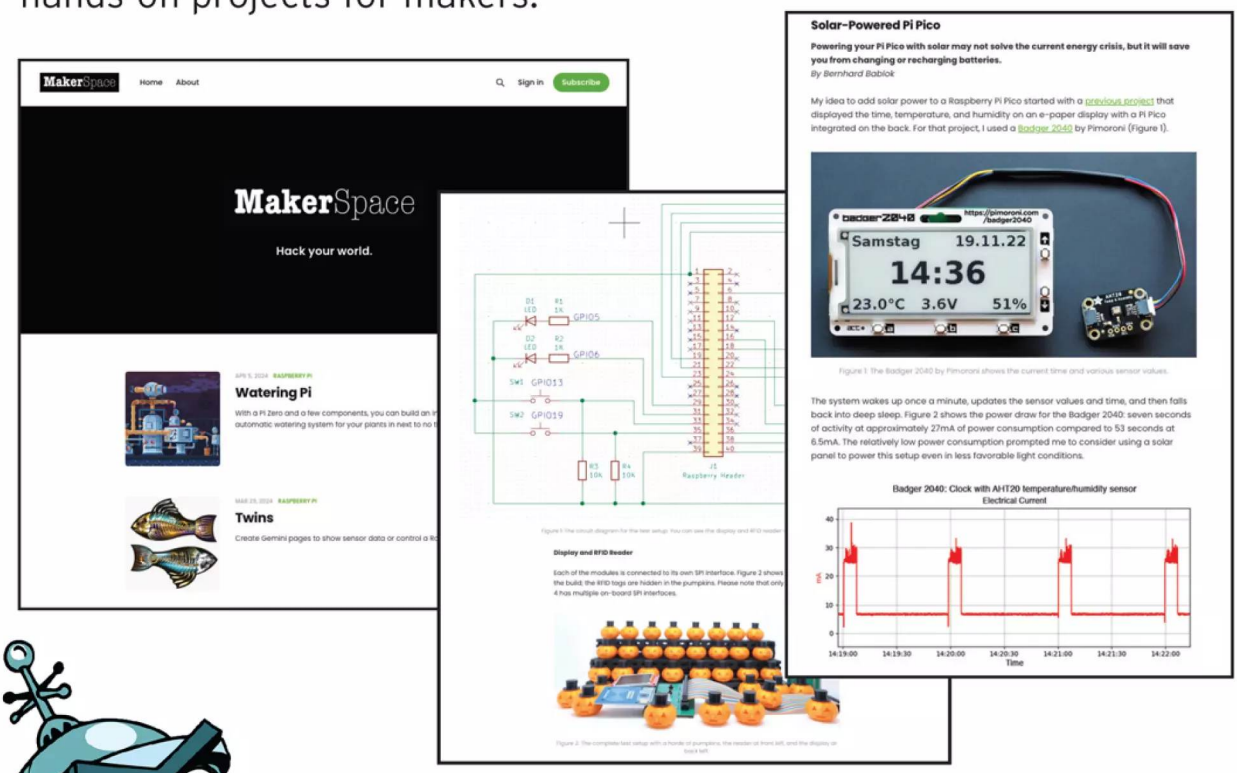


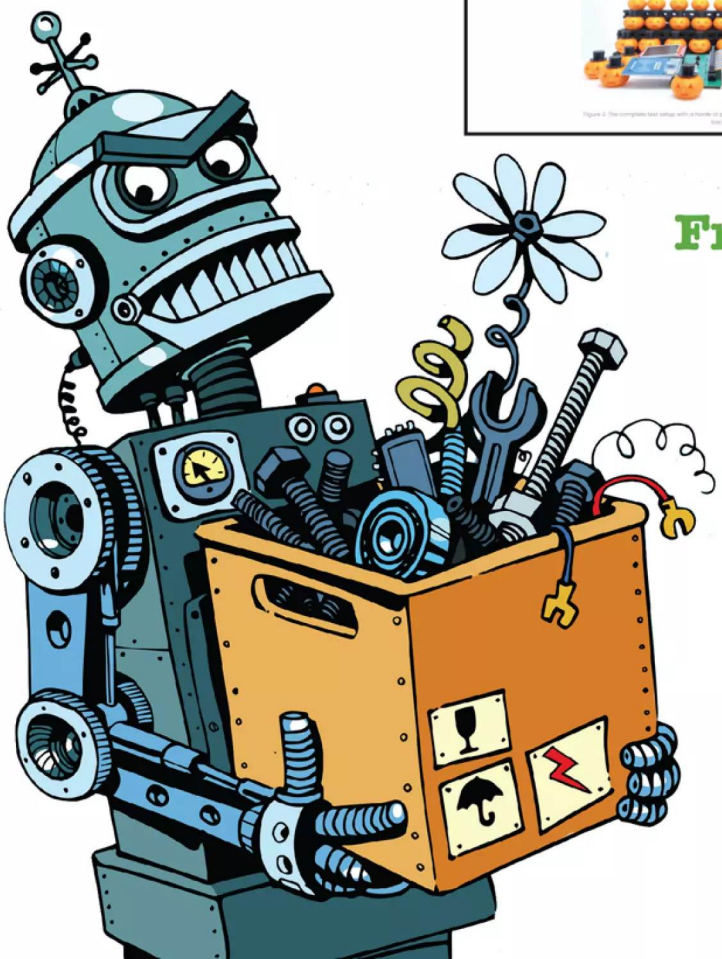
Figure 3: The Sinclair ZX Spectrum Next is back, *again*: We look at the second edition of the FPGA-based home computer.

MakerSpace-Online

New Maker Content Every Week

At MakerSpace, we are all about technology you can use to build your own stuff. Our goal is to help you turn your ideas into reality with hands-on projects for makers.





Fresh Content, Delivered

Subscribe now and join the MakerSpace community. You'll stay up-to-date when new content is published.

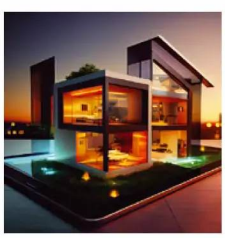
Join Now!

https://makerspace-online.com

MakerSpace #04

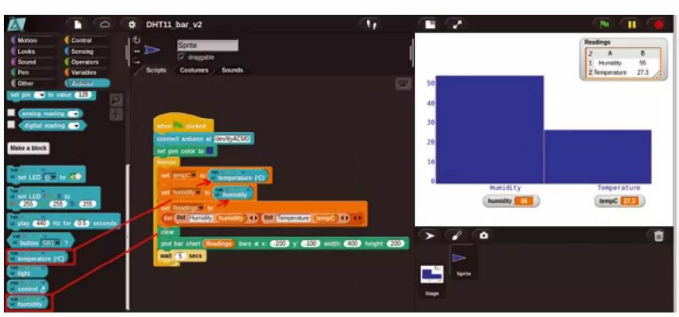
PROJECTS

AUTOMATION



RETRO COMPUTING

PROGRAMMING



SERVICE

Making a retro-style alarm clock

Clocking On

A few electronic components, some code, and a handmade wooden case make a fine retro-style bedside clock.

By Andrew Malcolm

When my venerable bedside clock radio and alarm – a present from my parents in the 1980s – finally died, as a maker, my first thought was not, "Where can I buy another?" but "Can I make one?"! I didn't really use the radio (I think it was AM only, and here in the UK, AM is no longer used much), so my ideas began to form around a simple, retro-style digital clock with an LED display in an attractive wooden case.

The case needed to be simple to build because my woodworking skills are limited. I chose a large green LED display for its restful color in the dark, and I wanted the clock to run off a USB wall socket to simplify the power supply design whilst retaining the possibility of running from a small external USB battery pack. I felt some user interface would be required to set the time, set and cancel an alarm, and control the display brightness. Five push-buttons are sufficient for this task. The alarm, of course, requires a buzzer or speaker. Table 1 lists the essential hardware elements.

Table 1: Bedside Clock

Components
A microcontroller (32-bit ARM CPU, 48MHz internal clock)
32KHz watch crystal for the real-time clock
8MHz crystal for microcontroller
Backup battery for the real-time clock
Four seven-segment displays, one inch high, for hours and minutes
Two discrete LEDs for the colon between hours and minutes
Piezoelectric buzzer for alarm
Four push-buttons to set time and brightness, one to cancel the alarm
Connectors for power, programming, debugging

Hardware Design

The core of the system is an STM32 microcontroller, specifically the STM32F-072CBT6 [1]. It drives the display, interrogates the switches, and drives the buzzer for the alarm. You will find a huge variety of microcontrollers on the market, so the choice of an appropriate device can be daunting. Previous experience tells me that ST Microelectronics devices perform well at a good price point and are very well supported in terms of development tools and online resources. Having designed a great many projects around these devices, I also know that software development will be accelerated by my familiarity with the STM32 family and their development tools.

This project could just as easily have been based on an Arduino, a PIC, or a Pi Pico; however, as I say, choice comes down to familiarity and suitable package configurations with the required I/O. The key feature required for a clock is clearly a battery-backed real-time clock built into the microcontroller, and with ST Microelectronics' excellent device

Lead Image © almoond, 123RF.com

selection tools, I was able to select a suitable device with this feature, as well as sufficient pins to drive the display, switches, buzzer, and debug port.

As you will see, onboard timers are key to this project and all STM32 devices come with several configurable timer units. The device needs an external crystal as a frequency source for the real-time clock, but in all other regards, is entirely self-contained. The frequency accuracy and temperature coefficient of this crystal will determine the overall accuracy of the clock, so it is important to choose this device carefully.

The project was designed to run from a standard USB socket, and the 5V provided powers the displays directly, whilst a small linear regulator provides the 3.3V supply to the microcontroller. A standard 3V button cell provides the battery backup voltage, which prevents having to reset the time if the unit is unplugged or power is lost in some other manner.

The display consists of four one-inch-high, green, seven-segment LEDs [2] for hours and minutes, arranged into two groups of two, and two color-matched discrete LEDs to form the flashing colon between hours and minutes. The seven-segment displays are multiplexed together (see the "Multiplexing" box) and driven by a seven-channel open collector driver chip and discrete transistors.

I chose four switches to mount to the left of the display and was able to find PCB-mounted switches with a button stem tall enough to protrude through the front panel past the displays. The fifth button is a large circular push-button in the top of the unit, used to cancel the alarm.

Schematic capture and PCB layout were both performed in KiCad, a free and open source CAD tool originally developed at CERN [3]. It really is an excellent suite of tools and handles the whole process of electronic design from schematic capture right through to generating files for manufacture. A 3D viewer generates an image of your design, including the components, that you can pan and rotate. Although PCB layout is a 2D activity, the ability to add 3D models of all the components and view the PCB assembly in a 3D viewer has saved me from mechanical clashes not apparent from the 2D design perspective.

Multiplexing

Multiplexing is used to limit the number of pins and drivers required to interface them to the microcontroller. The seven-channel open collector driver is connected to the cathodes of each display segment by current-limiting resistors, and each separate display unit has its own transistor to connect the common anodes to the 5V power supply at the correct time. The LEDs that form the colon separator are controlled by a single transistor, and they are separate from the multiplexing scheme. The decimal point in the rightmost seven-segment display is illuminated when an alarm is active, so a driver is provided for that, too. The buzzer is also driven by a transistor, with the microcontroller providing a 1KHz square wave pulsed at one-second intervals to indicate an alarm.

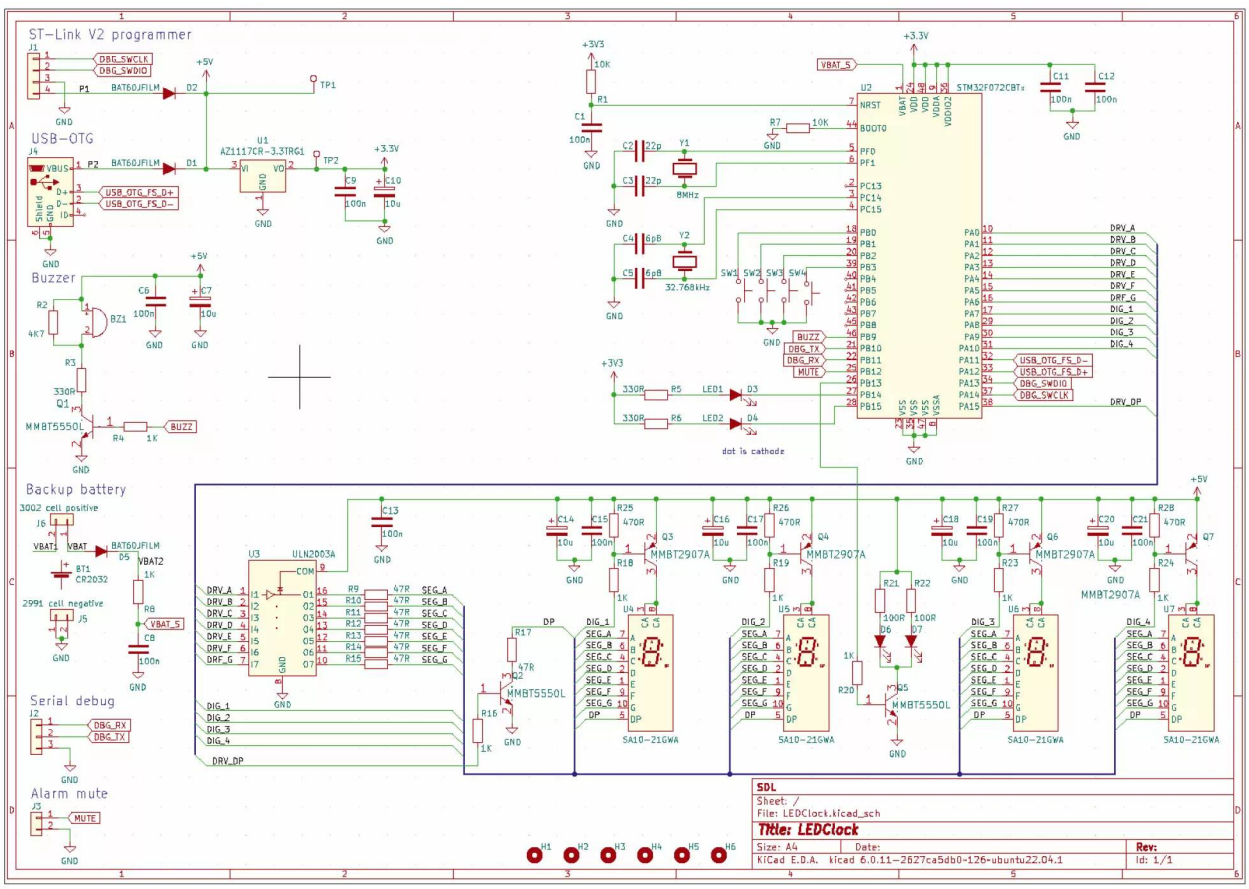


Figure 1: Schematic of PCB board.

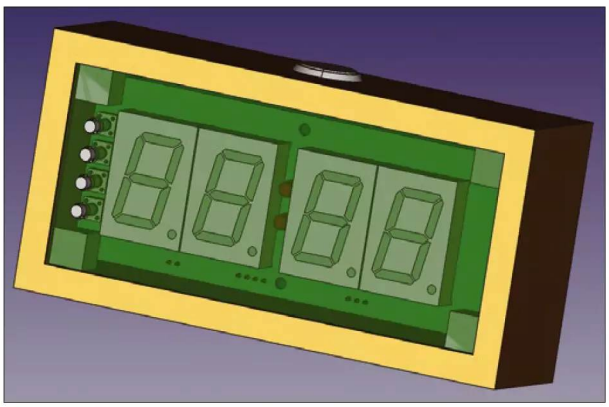


Figure 2: Three-dimensional model of bedside clock.

Exporting the 3D model as a STEP file and importing that into 3D CAD tools such as FreeCAD [4] builds more complex assemblies, aiding the design of parts (e.g., enclosures) suitable for 3D printing. Many excellent PCB companies online will build good-quality PCBs in a few days for less than $5 (EUR5, £5), so building prototype or experimental PCBs is not prohibitively expensive.

Once the PCB design was complete in KiCad, a 3D model of the PCB assembly was exported into FreeCAD, and an assembly of the case and front panel was created to ensure all the assumptions about dimensions were correct. The full schematic for the final design is shown in Figure 1.

The 3D model (Figure 2) ensured mechanical dimensions matched and provided drawings for the manufacture of the front panel and dimensions for the wooden case. The PCB assembly can be seen behind the transparent front panel and the alignment of push-buttons with the holes in the front panel is visible, too.

Listing 1: Number Lookup Table

```
// digit (0 to 9) to seven-segment pattern
// 1 in binary constant means segment on
// 0 means segment off
static uint8_t  patterns[16] =
{
    //ABCDEFG (segment id)
    0b1111110, // 0
    0b0110000, // 1
    0b1101101, // 2
    0b1111001, // 3
    0b0110011, // 4
    0b1011011, // 5
    0b1011111, // 6
    0b1110000, // 7
    0b1111111, // 8
    0b1111011, // 9
    0b0000000, // blank
    0b0000000, // blank
    0b0000000, // blank
    0b0000000, // blank
    0b0000000, // blank
    0b0000000, // blank
};

/** write to selected digit */
static void write_digit(int digit)
{
  // extract the required pattern by using
  // the number as an index into the lookup table
  uint8_t pattern = patterns[digit & 0x0f];

  HAL_GPIO_WritePin(SEG_G_GPIO_Port, SEG_G_Pin, (pattern >> 0 & 1));
  HAL_GPIO_WritePin(SEG_F_GPIO_Port, SEG_F_Pin, (pattern >> 1 & 1));
  HAL_GPIO_WritePin(SEG_E_GPIO_Port, SEG_E_Pin, (pattern >> 2 & 1));
  HAL_GPIO_WritePin(SEG_D_GPIO_Port, SEG_D_Pin, (pattern >> 3 & 1));
  HAL_GPIO_WritePin(SEG_C_GPIO_Port, SEG_C_Pin, (pattern >> 4 & 1));
  HAL_GPIO_WritePin(SEG_B_GPIO_Port, SEG_B_Pin, (pattern >> 5 & 1));
  HAL_GPIO_WritePin(SEG_A_GPIO_Port, SEG_A_Pin, (pattern >> 6 & 1));
}
```

Firmware

The real-time clock feature of the STM32 family of microcontrollers is central to the firmware design. Much like the real-time clock found in every PC, it provides a convenient record of the current time in hours, minutes, and seconds (although seconds are not used in this design) and runs even when the unit is powered down, as long as backup power is provided from a button cell. The time is set with two of the buttons on the front panel. When the upper button is pressed, the time increments by a minute. If the button is held down, the time advances at one minute every half second. The lower button operates in a similar manner, but decrements the time. The alarm is set in the same way, whilst holding down the button on the top of the unit to cancel the alarm.

Seven-Segment Displays

Seven-segment displays are generally used only to display numeric characters, although a small subset of the alphabet is also possible. In this case, only numbers are required. The means to turn the numbers 0 to 9 into the correct pattern for the seven segments is easily achieved with a lookup table (Listing 1).

Timers

The clock firmware relies heavily on timers internal to the microcontroller. These timers count cycles of the system clock (in this case, 48MHz) and generate an interrupt when a specified target is reached. In most cases the timers are

Multiple Displays on a Single Bus

Driving four seven-segment displays directly from the microcontroller would require at least 36 pins and 32 driver channels (if you include the decimal point) and would require a much larger microcontroller, four eight-channel driver chips, and a considerably more complex PCB layout. The driver chips are required because the microcontroller cannot sink sufficient current to light the LEDs). The usual solution is multiplexing, that is, to connect the cathodes in all the matching segments in each display together and to a low-side driver and the common anodes to high-level switches.

Each display is turned on in turn by these high-level switches whilst the correct seven-segment pattern is applied to the low-side driver. Therefore, each display is on for a maximum of one-quarter of the time, but by setting the LED current to an appropriate level, it's quite possible to provide a display that is clearly visible in daylight. This setup requires 12 I/O pins, four high-side transistors, and a single low-side driver chip. An excellent description of display multiplexing is described on Wikipedia [7].

programmed to reload the same target, resulting in periodic interrupts.

The first timer has a one-second period used to toggle the display colon on or off, giving the clock a live feel, because the time only changes once a minute. The timer also reads the latest time (to the nearest second) from the real-time clock and saves it in a variable accessible by the display multiplexer. The display multiplexer (see the "Multiple Displays on a Single Bus" box for details) itself uses a timer, which is set in pulse width modulation (PWM) mode, generating a periodic interrupt at the multiplexing rate (20ms), used to move to the next digit and update its data.

A second interrupt is generated at the end of the PWM period (less than or equal to the 20ms multiplexing rate) and switches off the current digit. In this way, if the PWM period is less than the multiplexing period, each display will be off for more than a quarter of the total time, causing the display to be dimmer. In this way, the PWM is used to control display brightness.

The clock has five push-buttons used to set the time, set or cancel an alarm, and control the display brightness. Four of these buttons are on the front panel adjacent to the display. They are connected to microcontroller I/O pins configured to generate interrupts when the push-button is pressed or released. Unfortunately, most push-buttons (and switches in general) do not change state cleanly but "bounce" between open and closed for several milliseconds before settling to the new state. This bounce can cause all sorts of weird effects if not dealt with, and push-button debouncing is a common problem often fixed in firmware with debouncing techniques. In this case, a timer is used for this function (see the "Debouncing Push-Buttons with Timers" box for details).

The push-buttons used to control the clock have varying functions depending on context: You need to differentiate between a brief push and a sustained press, the latter used to advance the clock rapidly when setting the time and alarm. Again, a timer is used – in this case, in one-shot mode. The timer is set when a button push is detected, and if, when the timer's interrupt fires, the push-button is still pressed, the "sustained hold" function is initiated.

Listing 2: Debouncing Switches

```
01 static bool timer_running = false;
02 static bool last_switch1_state = 1;
03
04 // interrupt callback called when a switch is pushed
05 void interrupt_callback(void)
06 {
07   if(!timer_running)
08   {
09     last_switch1_state = read_pin();
10   }
11
12   // start a timer to wait for 20ms
13   // then resample in timer ISR. only
14   // act if the two samples match
15   timer_running = true;
16
17   register_timer_callback(timer_callback);
18   start_timer();
19 }
20
21
22 // debounce switch: if state is still same
23 // as initial after 20ms confirm action
24 void timer_callback(void)
25 {
26   stop_timer();
27   bool switch1_state = read_pin();
28
29   if(switch1_state == last_switch1_state)
30   {
31     // switch push verified, do required action
32   }
33 }
```

Debouncing Push-Buttons with Timers

Push-buttons and switches rarely exhibit ideal behaviour: You would like them to go from open to closed in an instant. In reality, most exhibit what is known as switch bounce [8]. De-bouncing switches can be achieved in many ways, some of which require extra hardware (resistor-capacitor (RC) networks, set-reset latches, etc.). However, in a microcontroller environment timers are often seen as a simple solution when a delay of a few tens of milliseconds is acceptable.

The operation goes like this: The microcontroller is set up to produce an interrupt once the button is pushed. At this point the state of the switch is recorded, and a 20ms timer is started. When the timer expires, the state is recorded again. If the two states are equal, a switch event is deemed to have occurred. If the switch "flapped" between states in the intervening 20ms, those transitions must be ignored. Thus, it is important not to resample the switch state while the 20ms timer is running, so the interrupt routine that does the initial sampling must either be disabled in that period or contain some logic that has a similar effect (Listing 2).

Finally, a timer is used to generate a 1KHz square wave to drive the buzzer. The timer interrupts every half-cycle (500µs), and the buzzer I/O pin state is toggled. The number of toggles is counted, and the toggling is controlled in such a way as to produce bursts of 1KHz sound at regular intervals – not quite enough to raise the dead, but a gentle reminder that it's time to get up! One of the five push-buttons is a large circular type on the top of the clock case; pressing that button cancels the alarm.

Listing 3: Overriding Defaults

```
01 // the default implementation (in syscalls.c)
02 __attribute__((weak)) int _write(int file, char *ptr, int len)
03 {
04     ...
05 }
06
07 // the replacement to redirect output to a UART serial device.
08 int _write(int file, char *ptr, int len)
09 {
10     ...
11   HAL_UART_Transmit(&huart3, ptr, 1, -1);
12     ...
13 }
```

Software Development

I know the use of integrated development environments (IDEs) can be controversial and very much a matter of taste, and it's certainly possible to do this type of microcontroller development without one. The ARM compilers and standard libraries can be downloaded from your distro's repository, and you're off, with the use of any editor that suits you and `make` or `cmake`, again, at your choice. Once you have a compiled binary, ST-Link utilities allow you to program your device, and you can use the `gbd` utility to debug your program. If you don't want to use the hardware abstraction layer (HAL) libraries provided by ST, you can generate your own header files with the addresses of the microcontroller registers and all the bit patterns required for configuration.

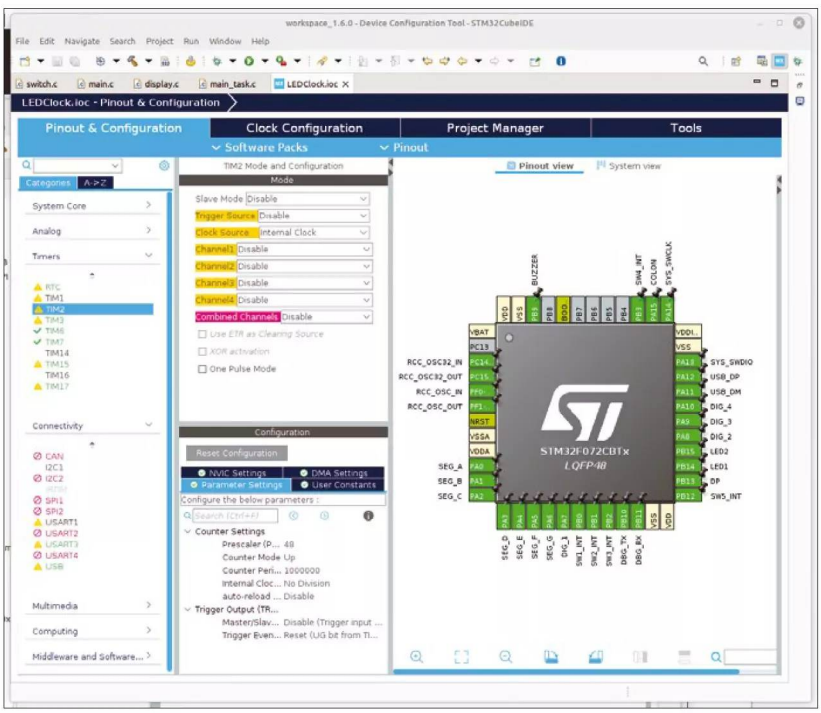


Figure 3: The pin configuration for this design and an example of a timer setup.

That said, ST's STM32CubeIDE [5] (Figure 3), which is based on Eclipse, does streamline the process by integrating ST's CubeMX tool, a utility that lets you configure your microcontroller and generate a software framework that does all the initialization and leaves you with a blank `main()` function, to which you add your own code. You can label the pins of the microcontroller (bonus points if you use the same names as on the schematic!).

The HAL libraries hide a lot of the complexity of setting up some of the peripherals, but they are not perfect and so must be used with caution. Lots of resources online show how to use the IDE to set up the clocks, UARTs, timers, USB ports, and the like on an STM32 processor, including ST's own getting started guide [6].

Once saved, the IDE then generates a set of `#define` lines for the I/O pins that you can use in your code, as well as a complete set of initialization routines. At this point, you can continue to use the IDE or ignore it and use `make` with the generated Makefile. However, if you stay with the IDE and have your hardware connected by an ST-Link programmer, a single mouse click in the IDE on the Run menu will compile, download, and run your code. This level of preconfiguration (including, if you want, the inclusion of a real-time operating system (RTOS) such as FreeRTOS) can leave you free to concentrate on your application code. In a commercial environment, time-to-market is everything, and time savings like this can be invaluable.

Redirecting stdin and stdout

Adding `printf()` statements to code is a time-honoured and useful debugging technique. In the STM32 environment, redirecting the `printf()` output to a serial port is straightforward; it then can be connected to a terminal emulator (e.g., Minicom) running on a laptop.

The low-level `_write()` function is defined in an STM32 library. The default implementation calls `__io_putchar()` in the C standard library, and in the embedded version of the library, the data goes to the equivalent of `/dev/null`.

The default `_write()` has weak attributes, which means it can be overridden by an alternative implementation simply

by providing a replacement function with the same signature without the `weak` attribute (Listing 3).

Once implemented, the full formatting power of `printf()` is available to output data to an external terminal. A similar technique can be applied to reading data from the terminal with `getchar()` or `scanf()`:

```
// read from serial port
int _read(int file, char *data, int len)
{
  uint8_t ch;
  HAL_UART_Receive(&huart3, &ch, 1, -1);
  return 1;
}
```

Software Summary

Unlike many command-line programs running on Linux or any other OS, nothing much happens in the `main()` of much embedded firmware, and in the present case, once the appropriate timers have been started and interrupt handlers registered, `main()` simply enters an idle loop. Hardware initialization has occurred in code generated by the IDE before `main()` is called, according to the device configuration shown in Figure 3, and in other settings in the IDE. The timers handle the display refresh and multiplexing, and the interrupt service routines are called when the push-buttons are pressed to change the time, set the alarm, cancel the alarm, or change the display brightness. That's it!

The Completed Unit

I used FreeCAD to design an enclosure for the completed unit with the use of a STEP file of the PCB assembly exported from KiCAD, to ensure a good fit and calculate the locations of fixing holes and the like. Although the case was made by hand from wood, as described below, the 3D model ensured I was able to cut the required pieces correctly the first time, eliminating wasted time and materials. The 3D model was also very useful in simply visualizing the finished item.

FreeCAD is another open source package, and every time I return to it, I find the developers have taken another step in increasing its functionality. It really is an exemplary open source project.

Construction

The front panel of the unit is a 3mm-thick piece of green-tinted Perspex, and the PCB with all the displays and other components are mounted directly behind this acrylic sheet. The PCB was manufactured with a black solder resist (mask) to avoid it being visible. The green tint does a good job of hiding the internals whilst letting through the green LED light.

The case is very simple (Figure 4). I wanted a real wood case for a retro feel. It's cut from a strip of oak 50mm wide and 8mm thick (2 inches by 5/16 inch for my American friends). The four sides are cut at an angle of 45 degrees to form mitred joints and glued together. Small wooden blocks are glued internally for mounting the PCB, and the front panel is a simple friction fit into the aperture. The back panel is a similar piece of opaque white Perspex fitted to the PCB with stand-offs. It has holes for mounting screws and a larger aperture to allow the micro-USB plug to pass through to the PCB-mounted USB socket.

The wooden case has a hole drilled in the top to receive the alarm cancel button. Small wooden feet are cut at an angle from an off-cut of the same wood, to set the clock at a slight upward tilt. Clear varnish protects the wood and completes the retro look. The dimensions were taken from the 3D model, which again proved invaluable in eliminating any nasty surprises.

Figure 4: The front and back of the alarm clock showing the wooden enclosure.

Wrap Up

This clock was a fun project: Once built, it is not dependent on any other equipment but is a useful object in its own right. I know these days there's always a phone to tell the time, but once you look at your phone, you can be drawn into your day rather than turning over and going back to sleep! This clock is clearly visible from my beside cabinet without raising my head, and with the display dimmed, it does not flood the room with light. I think the retro look is quite pleasing, but that's a matter of taste, of course. I've made a few of these clocks to give as gifts, and it will be interesting to see how they are received. The completed design, both hardware and software, can be found at my GitHub page [9]. ■■■

Info

[1] STM32F072CBT6: *https://www.st.com/resource/en/datasheet/DM00090510.pdf*

[2] Seven-segment LEDs: *https://www.kingbrightusa.com/images/catalog/SPEC/SA10-21GWA.pdf*

[3] KiCad: *https://home.cern/news/news/computing/kicad-software-gets-cern-treatment*

[4] FreeCad: *https://www.freecad.org/*

[5] STM32CubeIDE: *https://www.st.com/en/development-tools/stm32cubeide.html*

[6] STM32CubeIDE getting started: *https://wiki.st.com/stm32mpu/wiki/How_to_get_started_with_STM32CubeIDE_from_scratch*

[7] Display multiplexing: *https://en.wikipedia.org/wiki/Multiplexed_display*

[8] Switch bounce: *https://en.wikipedia.org/wiki/Switch#Contact_bounce*

[9] Author's GitHub project page: *https://github.com/andrewrussellmalcolm/LEDClock*

Author

Andrew Malcolm (MIET, CEng) is a retired hardware and firmware engineer. He maintains a keen interest in engineering in general and building embedded projects like this one in particular. He is a keen user of Linux and all its available open source and free engineering tools. You can contact Andrew at *andrewrussellmalcolm@gmail.com*.

Create an information center with the Raspberry Pi Pico W and Python

Pocket Sized

You don't need much to create a smart home information center – just a Raspberry Pi Pico, an ePaper panel, a battery, and some Python. *By Swen Hopfe*

People want a variety of information presented in the same place. On the road, a smartphone plays a central role for most. At home, you might have legacy displays for heating control, an alarm system, a weather station, and so on. A common display is useful for grouping data from different sources without the need to call up different apps or read the data in different places. A Raspberry Pi Pico W lets you build your own model.

A home automation solution already collects many useful values. For this sample project, I added news and a weather forecast through a connection to the Internet. To make the device compact and mobile, it has a battery. The display and controller need to be frugal in terms of power consumption, which is where the Pico W comes in handy. It connects to the WiFi network, collects the information, and displays it on a small screen (Figure 1).

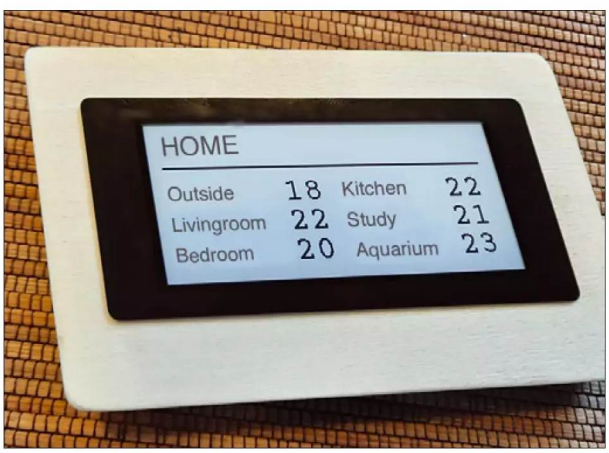


Figure 1: A lean, mobile information display fits anywhere and is frugal in terms of resources.

I deliberately kept the controls as simple as possible. Three screens show the news, weather, and home temperatures; where needed, screens branch out into submenus. I did not want to switch the various actuators of the home automation system – simply display their status.

I specifically prioritized minimal power consumption, which is why I chose the Waveshare 2.9-inch Cap-Touch ePaper display module for the Raspberry Pi Pico [1]. It has a slot for the Pico at the back, which in turn supports touch operation, and as with the Pico, you can switch the mini-screen to sleep mode. The complete circuit can be disconnected from the battery during extended periods of disuse. So I wouldn't need to change the battery, I used a lithium-ion polymer (LiPo) battery and charging electronics, which meant I was able to charge the device with a power supply unit over USB when required.

Pico W

The basis of the project is the Raspberry Pi Pico W and its Python firmware. Unlike the single-board computers (SBCs) in the Raspberry Pi family, the microcontroller requires very little preparation. To program the microcontroller unit (MCU), you just need to connect it to a computer over USB. In the development phase, the controller can be fed external commands in an integrated development environment

Lead Image © donatas1205, 123RF.com

Parts List

- Raspberry Pi Pico W
- ePaper panel (Waveshare Pico CapTouch ePaper 2.9)
- LiPo battery (1000mAh)
- Pimoroni LiPo shim for Pico charging electronics
- Alternative 5V power supply
- Case
- Push-button switch
- Wiring, installation material

(IDE) such as Thonny. You only transfer the finished code to the module at the end.

One advantage of microcontrollers over computers is their robustness. You don't run the risk of damaging a storage medium by turning it off abruptly; this feature is useful for the current project.

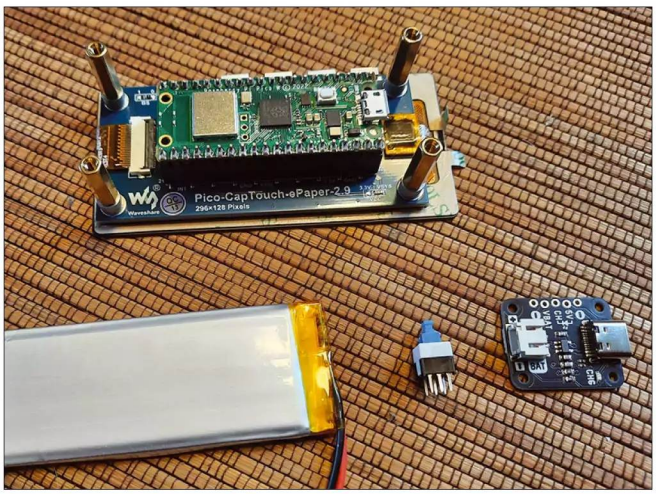


Figure 2: Thanks to the protruding front on the panel, you can do without a separate cover frame.

Layout

The Pico W slots in at the back of the ePaper panel. The protruding front of the touch variant of the 2.9-inch panel (Figure 2) proves to be useful for the construction, removing the need for an additional covering frame. Slotted into a recess in the case, everything looks neat from the front. During the installation, you need to work quite precisely to use the narrow adhesive border: Take care not to damage the delicate connections from the printed circuit board (PCB) to the display panel.

The goal of the build was to keep the case as flat as possible, so it can be used as a tabletop variant and as a wall unit. The Pi Pico's slot at the back of the ePaper doesn't prove to be conducive to a low build height; therefore, it makes sense to position the LiPo battery at the side. You can choose flatter batteries that involve some sacrifices in terms of capacity, but they are still powerful enough for a few days of runtime. LiPo batteries are compact and lightweight and will survive many hundreds of charge cycles – ideal for this small device.

The number of components is manageable (see the "Parts List" box). You also need to assign some space for the charging electronics, which incidentally also act as a DC/DC converter for converting the LiPo battery's 3.7V to 5V operating voltage. I also integrated a separate on-off switch. I did not want to use the buttons on the ePaper module; instead, the device is controlled by touch input only during operation. In the end,

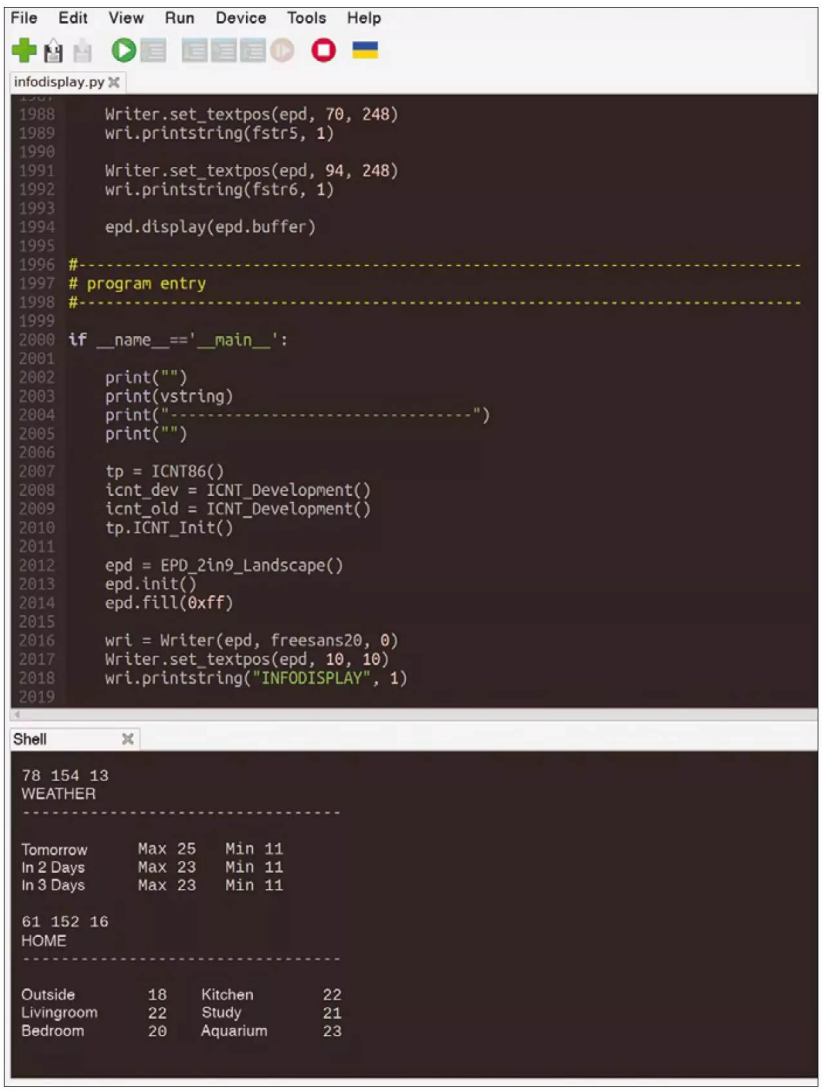


Figure 3: The Thonny IDE is ideally suited for integrating various Python interpreters.

I only had to provide an opening for the USB charging port.

Control

The circuitry is controlled by a script on the Pico. As usual, before applying the supply voltage, pressing the BOOTSEL button of the Pico opens a window in the file manager where you place your program text. This operation works smoothly in both Linux and Windows environments. The Python firmware expects a file named `main.py`, which in this case already contains all the libraries needed.

If you do not want to switch manually between mass storage and programming mode every time, you will appreciate a development environment such as Thonny (Figure 3). At first glance it seems relatively simple, but turns out to be extremely convenient when integrating various Python interpreters. On the Pico, it is absolutely my favorite tool.

The project requires Internet access for the display. After the first start-up, registering the display (or the Pico) as a supported device on the WiFi network in the settings of the home router and assigning it a permanent local IP address and unique name is recommended. In the test phase, a USB connection from the PC to the Pico is sufficient and supplies the attached display with power, as well.

The script [2] shows three screens from different sources. *NEWS* lists headlines from national news providers, *WEATHER* uses weather data from the weather service. *HOME* presents values from the home server.

For further processing, you need to import data in XML and JSON formats for the news and the weather forecast. In the Python script, you can do this easily without having to include additional classes with elaborate import filters. Of course, if the provider changes, this means making adjustments to match.

For the values around the house (e.g., the temperatures of the individual rooms, the outdoor area, or the aquarium), my home automation solution provides its data as files that have been prepared on my own server for the sake of simplicity. I just need to read the files.

ePaper Programming

ePaper does not just mean the electronic edition of a printed magazine. In electronics, ePaper or eInk displays work on the basis of electrophoretic "ink." Because they are not self-luminous, they are best read when ambient light is sufficient.

A key technical feature of such displays is that they only require energy when changes occur. In idle mode, the control consumes an almost negligible amount of power, which predestines ePaper displays for mobile and battery-powered solutions. The catch is that you have a long wait when deleting and updating content, especially on large displays; also, some flicker can occur. Only a few models can handle partial updating, which eliminates such problems somewhat.

ePaper and eInk displays are particularly suitable for displaying content that changes infrequently (Figure 4), which is not a problem in this project because you don't have to check the news, weather, or temperatures every minute. Also, the display does not need a constant time display. The monochrome display also has a partial refresh function, which is a useful prerequisite for convenient reading.

Like other modules with different technologies (OLED, LCD), the ePaper display requires an appropriate driver in MicroPython to provide simple methods for control, such as outputting text and graphics, which is the Python `EPD_2in9_Landscape` class in the code. Because the display does not have any built-in fonts in the hardware and the Pico does not address external memory, all font definitions are integrated into the source code.

One special feature of the panel is its touch input. More detailed instructions can be found in the sample code on Waveshare [3] and its GitHub site [4]. Controlled by interrupt requests (IRQs), you field the necessary coordinates and then use them for further processing to follow different branches in the program sequence.

Operations

I wanted to keep the operation of the informational display as simple as possible. After switching on, the display tells you about the successful connection to the Internet and services before displaying the first screen.

Figure 4: The ePaper display only requires energy when changing the display.

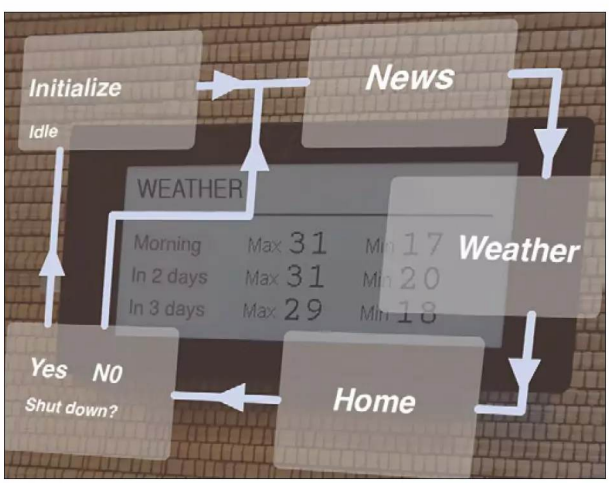


Figure 5: A simple tap lets you jump from one screen to the next.

A single tap takes you to the next screen in each case (Figure 5). Once you have browsed all three screens, the device will ask on the – now split – screen if you want to turn it off. At this point, you can either scroll forward from the beginning or switch the display and the Pi Pico to sleep mode. The device transitions to this mode if there is no input for a long period of time. After switching off and back on with the outer switch, processing starts again.

Conclusions

The information display sits on my desk because I want to have it conveniently in sight in a place where I often spend time. It's been in operation for a few days and has already proven its value; I find myself taking a quick look at it every now and then.

As is so often the case, you have some scope for improvement. I would like to add more sources around the house and web and integrate an icon bar to make everything more interesting and more convenient to use. My GitHub site [5] has the current state of the software and provides more detailed information on the project. ■■■

Author

Swen Hopfe works for a medium-sized company with a focus on smart cards and near-field communication (NFC). When he is not taking photos in the great outdoors, or in his garden, he focuses on topics such as the Raspberry Pi, Internet of Things, and home automation.

Info

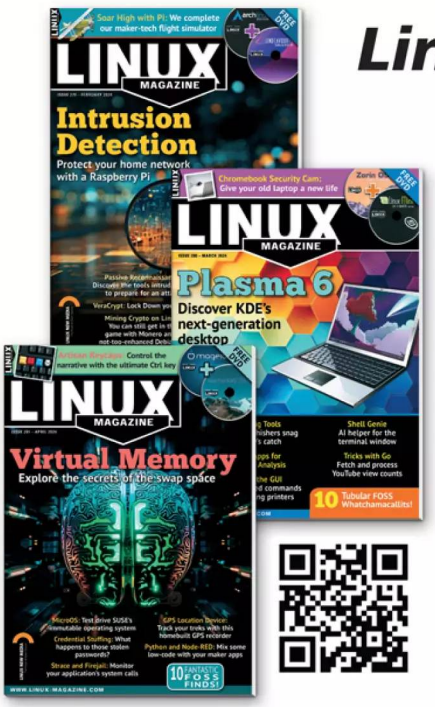
[1] Waveshare CapTouch ePaper 2.9: *https://www.waveshare.com/pico-captouch-epaper-2.9.htm*

[2] Code for this article (English text and comments): *https://linuxnewmedia.thegood.cloud/s/XnzsiEKtagjHKr3*

[3] CapTouch wiki: *https://www.waveshare.com/wiki/Pico-CapTouch-ePaper-2.9*

[4] CapTouch GitHub: *https://github.com/waveshareteam/Pico_CapTouch_ePaper*

[5] Project on GitHub: *https://github.com/swenae/infodisplay*



Linux Magazine Subscription

Print and digital options

12 issues per year

SUBSCRIBE

shop.linuxnewmedia.com

Expand your Linux skills:

- In-depth articles on trending topics, including Bitcoin, ransomware, cloud computing, and more!
- How-tos and tutorials on useful tools that will save you time and protect your data
- Troubleshooting and optimization tips
- News on crucial developments in the world of open source
- Cool projects for Raspberry Pi, Arduino, and other maker-board systems

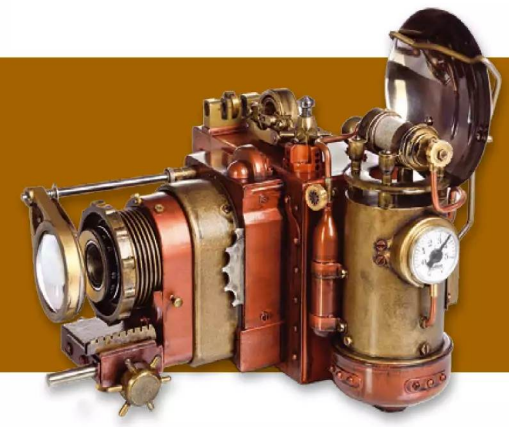
Go farther and do more with Linux, subscribe today and never miss another issue!

Need more Linux?

Subscribe free to Linux Update

Our free Linux Update newsletter delivers insightful articles and tech tips to your inbox every week.

bit.ly/Linux-Update



Make a camera for lenticular photography

Wiggle Time

You can take lenticular images with a homemade camera to recreate the "wiggle" pictures of your childhood.

By Günter Pomaska

Lenticular images store multiple exposures in the same area. Animation is achieved by tilting the image. Another application creates a spatial appearance without special tools (autostereoscopy). The digital version of this often shows up on social media as a "wigglegram."

Lenticular Cameras

On the consumer market, lenticular cameras are sold under the name ActionSampler. More than 40 years ago, the four-lens Nishika (Nimslo) appeared, followed by Fuji's eight-lens Rensha Cardia in 1991. Unlike the Nishika's synchronous shutter action, the Fuji exposed the 35mm film sequentially. Even today, the analog scenes are still very popular on Instagram and the like.

One way of creating a multilens digital recording system is to use a Raspberry Pi and a Camarray HAT [1] (hardware attached on top) by ArduCam [2]. The camera I make in this article uses four Sony IMX519 sensors arranged at a distance of 4cm apart (Figure 1). After the first exposure, you can move the device by half the camera distance, which produces eight shots of a subject at equal distances with a total of 32 megapixels (MP).

Figure 1: With four cameras, you can take a total of eight images at the same distance by moving the camera a short distance.

Lenticular Technology

The predecessors of today's lenticular screens are corrugated and lamellar screens that take two and three displayable images, respectively. Unlike the planar image strips of their predecessors, the lens screens commonly used today are transparent films of semi-cylindrical strips that show multiple images simultaneously [3]. Depending on the viewer's angle of view, the left eye sees something different than the right eye, and the viewer perceives the view as three-dimensional (Figure 2).

The lenses differ in terms of thickness and curvature radius; resolution is stated in lines per inch (lpi). Changing the

Lead Image © 3355m, 123RF.com

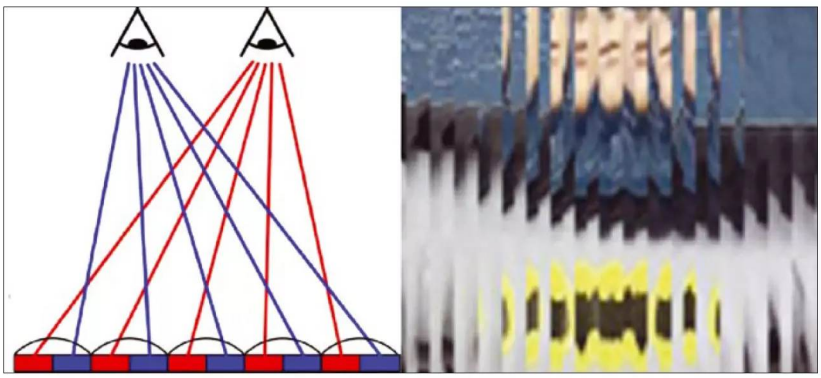


Figure 2: Depending on the viewing angle, the left and right eyes see different images, and together perceive a three-dimensional image.

image, animation, and zooming and morphing effects can be achieved with image strips arranged horizontally. To separate spatial images you need vertical image strips; the input images are encoded strip by strip in line with the lens spacing and are printed on a self-adhesive foil or as mirror images on the reverse side of the foil.

You can achieve a spatial vision effect by nesting the individual images inside each other, which leads to image separation for the viewer. However, you do not need to restrict yourself to two images; instead, you can compile a series of images. To do this for static scenes, you move the camera step-by-step. Alternatively, you can use camera technology with multiple lenses, which is also how to capture dynamic scenes. StereoPhoto Maker [4] is freeware for preparing image series. If you want to look more closely into wigglegrams, it is a good idea to take a look at the Triaxes 3DMasterKit [5] software.

Four-Lens DIY Camera

As the control unit, I will add the ArduCam Camarray HAT to a Raspberry Pi 4B. The Pivariety manufacturer makes extended solutions for Raspberry Pi standard cameras that act as Video for Linux version 2 (V4L2) devices. The HAT operates four Sony IMX519 sensors over a Camera Serial Interface (CSI), which you address on the I2C bus. The sensors have an image memory of 16MP, but depending on the addressing, one or more sensors share the image memory. The sensors can be operated in autofocus mode or with manual adjustment. The field of view is 80 degrees horizontally, and sharpness starts at about 8cm.

While you are on the move, a 5V power bank supplies the unit with energy. Three-dimensional printed components let you design a case. The camera boards sit side by side in the supplied brackets. Of course, you can't adjust a setup like this with single-pixel accuracy, but you don't need that because you can use the application software instead. The housing is designed so you can move the entire lens board by half the camera distance. In this way, the data for a lenticular image can be assembled from eight exposures, each 2cm apart over a base of 14cm.

The multicamera adapter is connected to the sensors by four interfaces that use ribbon cables. You then need to connect it to the computer on the CSI interface. Three spacer screws secure the mechanical connection to the Raspberry Pi; the 5V power supply comes through the GPIO pins. How you arrange the cameras is entirely up to you. The boards each come in a small case, and you install them 40mm apart. If you do not use the housings, the minimum distance is reduced to 24mm. The software addresses the sensors as a single frame. By setting the corresponding I2C parameters, you can configure one, two, or four sensors. The cameras always have to share the available resolution.

The default is

```
i2cset -y 10 0x24 0x24 0x00
```

(i.e., Quadro mode). Accordingly, the resolution for each image is restricted to a maximum of 2328x1746 pixels, and synchronization is in pairs at frame level. If you use the following parameters:

```
i2cset -y 10 0x24 0x24 0x01
```

the result is a resolution of two times 2328x3496 pixels in dual mode, which is extrapolated to two times 4656x3496 pixels later in the application. You may

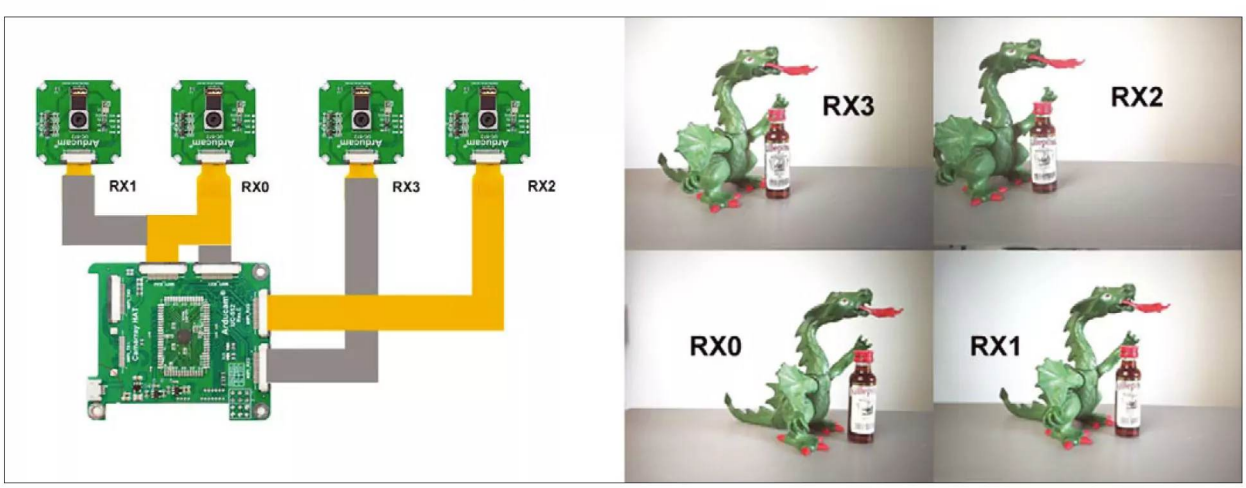


Figure 3: The close-ups on the right reflect the camera arrangement. The cabling diagram is shown on the left.

already be familiar with image compression from stereoscopy.

The images in Figure 3 on the right were taken from close up and therefore clearly reflect the camera layout and cabling (from left to right: RX2, RX3, RX1, and RX0). Despite the convenient autofocus mode, it is important not to forget the manual focus options. Especially at close range, manual focus results in some interesting photographic options (Figure 4).

Installing the Camarray HAT

You can install the required applications and the driver for the quad kit with the shell script install_pivariety_pkgs.sh (Listing 1). More information is available in the ArduCam documentation.

After the `libcamera-hello` command, the camera will respond for a short while. The

```
libcamera-still --list-cameras
```

command (Listing 1, last command) checks which cameras are connected. As mentioned before, the software identifies the four sensors as a single device.

Libcamera

The release of the Raspberry Pi OS "Bullseye" operating system in November 2021 fundamentally changed the handling of the camera module. Brand new *libcamera* commands have since replaced the tried and trusted command-line tools `raspistill` and `raspivid`. You can still use `raspistill` in legacy mode, but makers with more ambitious goals need to get comfortable with the *libcamera* library.

The transition of the camera control to the Linux kernel's Libcamera driver ensures a standards-compliant solution without proprietary code. New commands such as `libcamera-still` or `libcamera-vid` are available, and you can build your own apps on the Libcamera code. Extensive documentation can be found on the Raspberry Pi Foundation [6] website.

If you have already worked with `raspistill` or `raspivid`, it should not be difficult to come to grips quickly with Libcamera. The sample code

```
$ i2cset -y 10 0x24 0x24 0x00
$ libcamera-still ➥
  -t 30000 ➥
  --ev -5 ➥
  --gain 8 ➥
  --roi 0,0,1,1 ➥
  --autofocus ➥
  --info-text "Killepitch" ➥
  -o testQuadro.jpg
```

captures the entire image (region of interest, `--roi`) in autofocus mode after a preview time of 30 seconds (time out, `-t`) with an exposure compensation of -5 (exposure value, `--ev`). The `--gain 8` parameter corresponds to an ISO value of 800, and the `--info-text` flag lets you manipulate the header in the application; the output file is assigned the name `testQuadro.jpg` (output, `-o`).

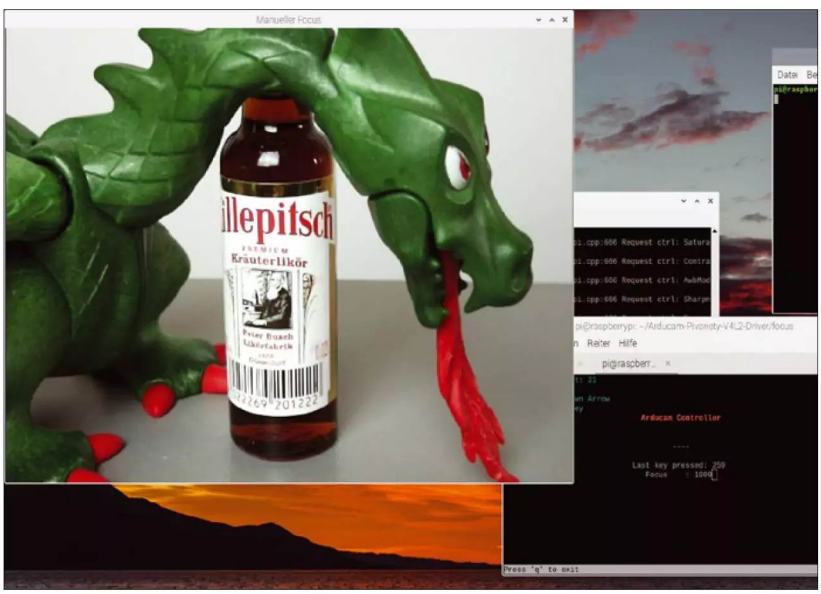


Figure 4: By focusing manually, you can significantly increase your freedom as a photographer.

Listing 1: Installing the Camarray HAT

```
$ wget -O install_pivariety_pkgs.sh https://github.com/ArduCAM/
  Arducam-Pivariety-V4L2-Driver/releases/download/install_script/install_
  pivariety_pkgs.sh
$ chmod +x install_pivariety_pkgs.sh
$ sudo apt-update
$ ./install_pivariety_pkgs.sh -p libcamera_dev
$ ./install_pivariety_pkgs.sh -p libcamera_apps
$ ./install_pivariety_pkgs.sh -p imx519_kernel_driver_low_speed
[...]
$ libcamera-still --list-cameras
  0 : imx519 [4656x3496] (/base/soc/i2c0mux/i2c@1/imx519@1a)
  Modes: 'SRGGB10_CSI2P' : 1280x720 1920x1080 2328x1748 3840x2160 4656x3496
```

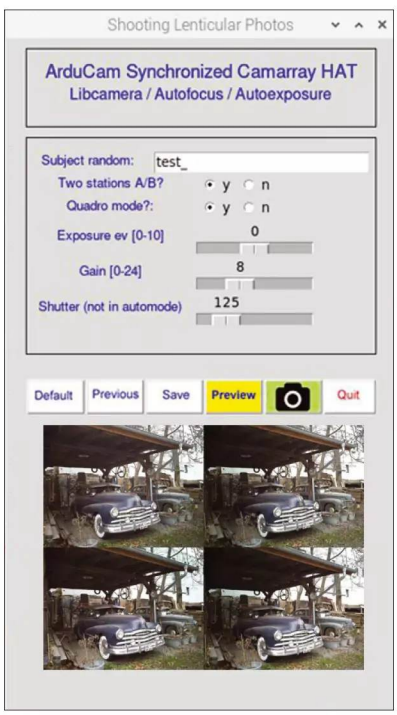


Figure 5: You can set the basic values for the shot in a GUI.

Shooting Lenticular Photos

The DIY camera is designed to be point-and-shoot, but the implementation is a little more modest because of the available technology. In my test environment, the system is connected to the local WiFi network behind a mobile router. After switching on the camera, the operating system boots and logs on to the WiFi network. The Virtual Network Computing (VNC) server starts up at boot time.

The same applies to the graphical user interface (GUI; Figure 5), with simple setup functions such as image name, shutter speed, exposure value, preview image, and shutter trigger. The GUI offers more functions, but I will not be using them for the time being. The software, written in Python, uses Guizero with object-oriented controls. It keeps its settings in a dictionary and uses system commands to call the camera functions (see also the "Graphical User Interface" box).

The images for positions A and B and the individual image tiles end up in the `dcim/randomCode/` directory with randomized image labels. The GUI displays the generated random code, which can be changed alphanumerically if required. You decide in advance whether you want to create two, four, or eight exposures. The camera settings can be saved so that you can reuse them for later shots. To align the camera, click the *Preview* button; status messages are displayed in the header.

Finding a shooting scenario is now the problem. The camera is oriented horizontally. By rule of thumb, the distance to the object is 30 times the distance from the right to the left camera (close-up distance 1/30). Intermediate images split this distance evenly, as you can see in Figure 6, working with four sensors at a distance of 12cm with two intermediate images. This results in a close-up distance of about 3.6m. If you move the lens board by 2cm, you end up with four shots at 8cm apart, and ideally approach the subject to within 2.4m.

These approximations are only rough, based on experience, and by no means binding. The close-up value could well be closer to 1/20 than 1/30. As soon as you move your camera between two positions, you are forced to limit your work to static objects. Manual exposure by shutter speed and gain settings is generally recommended.

After capturing an image, you then need to break the frame down into individual tiles for further processing with the help of the FFmpeg suite or ImageMagick tools, which you can install with:

```
sudo apt-get install imagemagick
sudo apt-get install ffmpeg
```

To ensure that the images are ordered correctly, add a numerical suffix in the file name.

Graphical User Interface

In the download section for this article you will find the `lentiCam.py` GUI [7], which I programmed in Python. The `dcim/` directory also contains some recent images as examples of the components of a lenticular image.

Making One Out of Many

Once you have done the field work, you can continue processing the image series on the computer. Download the images with an FTP client; then, StereoPhoto Maker (for example) will give you all the functions you need for downstream processing. Triaxes [8] is also a good choice for lenticular images. With just two exposures, you can create an anaglyph image for red and cyan glasses or a simple wiggle image. More uniform motion and lenticular images will always require a series of images.

To begin, get a series of eight images for a spatial image. The images must be aligned uniformly; even slight skew and small vertical differences will make the images unusable. In StereoPhoto Maker, select *File* | *Multiple Images* | *Auto rotation adjustment* and select the images to be adjusted. In the second step, you need a common reference point in each image. The function for this can be found in *File* | *Multiple Images* | *X-Y adjustment and cropping*.

Now you can print the image by selecting *Edit* | *Create Lenticular Image*. Set the *Lenticular Lens Pitch* and the printer resolution to match the lenticular film. Finally, print the image with *File* | *Print preview*. Lenticular film of 15x10cm is available with different lens spacings with vertical and horizontal alignment. You need to align the self-adhesive films over the printout and then carefully press them on. A laminator is useful for larger formats.

If you want to process the image series as a wigglegram, use the ImageMagick `convert` function. The following command adds all JPEGs with the `image` prefix in the current directory to the animated GIF:

```
$ convert -delay 10 -loop 0 ➥
  image*.jpg <Wiggle>.gif
```

The playback is in an infinite loop at 10fps. The procedure depends on the

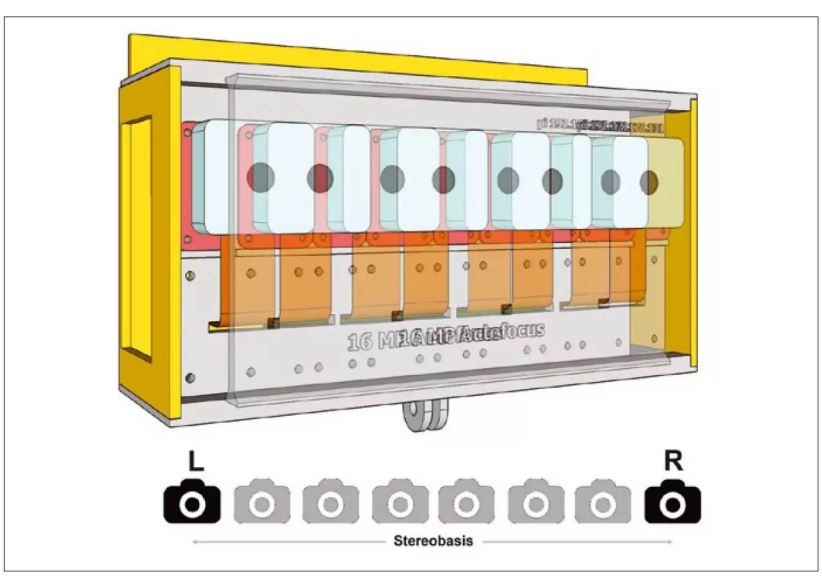


Figure 6: Intermediate frames divide the base (distance between the right- and left-most cameras).

size and number of images. Instead of a GIF, you can use the MP4 format. A wiggle cannot be printed, of course, so check out the examples online [9] if you need a visual reference.

The Triaxes 3DMasterKit, which is a commercial product, is a good choice for lenticular images. The license will not cost you much, and the investment is definitely worthwhile. After you upload the frames, you can reorder them and orient them alternately before cropping the images and computing the lenticular image. The kit also has other useful features, such as animations and layered 3D.

Conclusions

Lenticular images as analog 3D representations, and animations and wiggles for the Internet, give photographers a creative tool. Even with a conventional camera, you can achieve presentable results with a little practice.

The Camarray HAT by ArduCam lets you use a multisensor system in single, dual, or quadro mode and construct a DIY camera that suits your ideas. This hardware opens up a wide field of experimentation for amateur photographers, ranging from high-quality stereo images to low-resolution shaky images.

All you need for the build is a Raspberry Pi, a multicamera system, and a power pack. On the local WiFi network, a smartphone or tablet gives you a graphical user interface, and Python, Libcamera, and Guizero form the software underpinnings. StereoPhoto Maker and Triaxes take care of downstream processing. ■■■

Author

Dr. Guenter Pomaska worked for many years in the fields of photogrammetry, application software development and support. He was teaching computer graphics and related topics in higher education institutes. After retirement he directed his activities to 3D imaging.

Info

[1] Installing the Camarray HAT: *https://forum.arducam.com/t/imx519-quad-hat-mode-switching-and-faq/2399*

[2] Documentation for the Camarray HAT: *https://www.arducam.com/docs/cameras-for-raspberry-pi/raspberrypi-libcamera-guide/*

[3] Source for Lenticular film: *https://www.glaserde.de/shop/Lentikular_Folien_DIN_A6/index.html*

[4] StereoPhoto Maker: *http://stereo.jpn.org/ger/stphmkr/index.html*

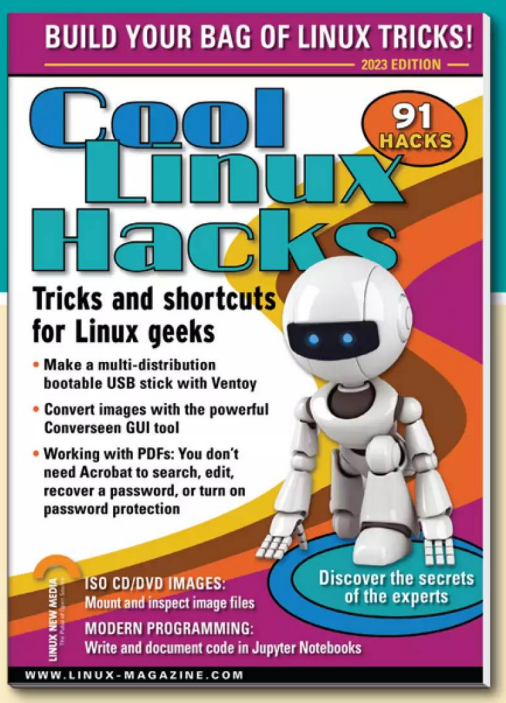
[5] 3DMasterKit: *https://triaxes.com/3dmasterkit/*

[6] Libcamera documentation: *https://www.raspberrypi.com/documentation/computers/camera_software.html*

[7] Code for this article: *https://linuxnewmedia.thegood.cloud/s/XnzsiEKtagjHKr3*

[8] Triaxes: *https://triaxes.com/legend/*

[9] 3D-Foto und Video: *https://www.3d.imagefact.de* (in German)



SHOP THE SHOP
shop.linuxnewmedia.com

GET PRODUCTIVE WITH
COOL LINUX HACKS

Improve your Linux skills with this cool collection of inspirational tricks and shortcuts for Linux geeks.

- Google on the Command Line
- OpenSnitch Application Firewall
- Parse the systemd journal
- Control Git with lazygit
- Run Old DOS Games with DOSBox
- And more!

ORDER ONLINE:
shop.linuxnewmedia.com

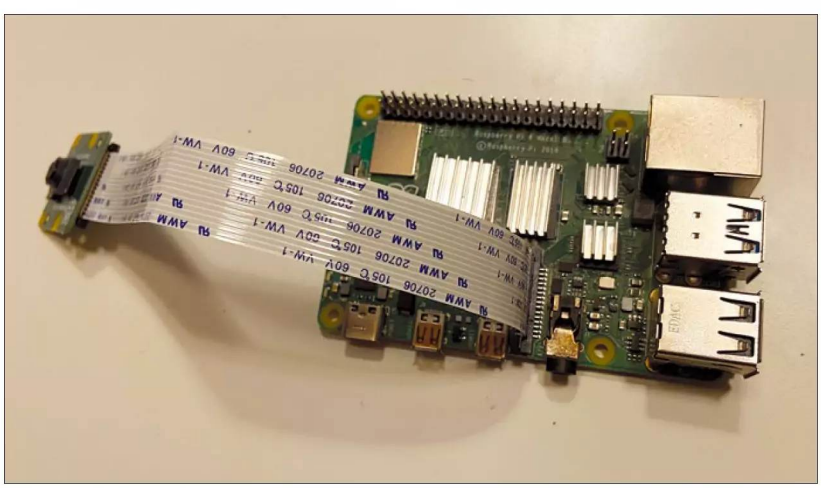
Watching your pets with a Raspberry Pi and a mesh VPN

Good Dog

A Raspberry Pi, a Pi-compatible camera, and a mesh VPN are all you need to watch your pets from afar. *By Bruce Hopkins*

You have a pet, but you need to be away from home for a day. Is it really worth paying for a kennel when all you want is to keep an eye on your pet while you're away? Of course, you could use a video doorbell service (like Ring or Arlo) and simply locate the camera device inside your home, but it seems like overkill to pay for a monthly service for something that you only need occasionally. Also, for privacy reasons, some people might not want to invite a streaming video service into their home when they don't have control over how the video is processed and stored.

On the other hand, if you have a Raspberry Pi, an Internet connection, and a Pi-compatible camera, it is actually pretty simple to set up your own remote access pet cam with a mesh VPN. Years ago, in order to get something like this to work, you'd need to open a port on your home router/firewall and enable port forwarding in order to allow incoming connections. That's a risky undertaking, because you're opening the door for anyone who port scans your router to find an entrance to your home network. Thankfully, there are free services available that allow you to host your own VPN using your Internet connection at home.



Lead Image © OLEG KIRILLOV, 123RF.com

Figure 1: Installing the camera on a Raspberry Pi.

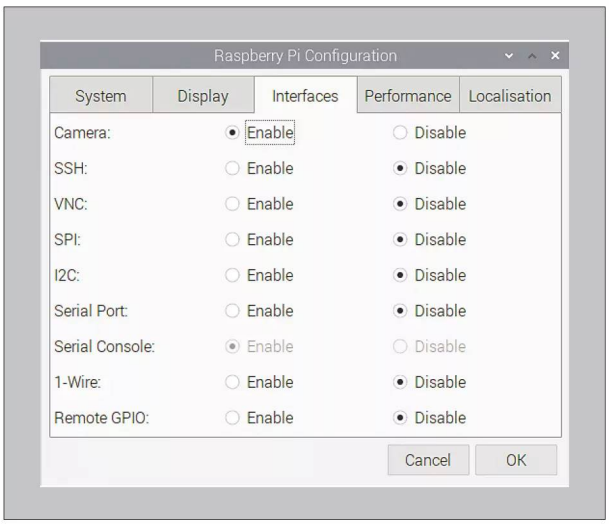


Figure 2: By Default, Raspbian has the camera interface disabled, so don't forget to enable it in order to record video or take pictures.

Setting Up a Camera on Your Raspberry Pi

The first step is to set up a Pi-compatible camera for your Raspberry Pi. I say, "Pi-compatible" because there are many compatible cameras that work with the Raspberry Pi. Therefore, feel free to find any camera that works with your version of the Raspberry Pi and fits within your budget. I tested my setup with the Pi Camera Module 2 [1], which has an 8MP image sensor and only cost about $10 online. In case you've forgotten, the Raspberry Pi includes a dedicated camera port directly on the board that is used exclusively for any of the Pi-compatible cameras (Figure 1). Therefore, installing the camera is a four-step process:

1. Shut down your Raspberry Pi if it is already turned on.
2. Lift (but don't try to detach) the plastic guard that's on top of the camera port.
3. Insert the ribbon cable for your Pi-compatible camera into the camera port, and ensure that the blue strip on the ribbon is facing towards the USB and Ethernet ports.
4. Push down and close the plastic guard on the camera port to lock everything in place.

Now that you've got your camera installed, the next step is to ensure that your Raspberry Pi recognizes the camera. Go to *Preferences* | *Raspberry Pi Configuration* to access the *Preferences Configuration* screen for your Raspberry Pi.

Click on the *Interfaces* tab and make sure that you select the option that enables the camera (Figure 2). To save the settings, click the *OK* button, and then reboot your Raspberry Pi.

Taking a Still Photo

Now that you've gotten your camera installed and configured, you need to make sure that the drivers work properly and can recognize the camera. Therefore, take a simple still photo (no need to try out streaming video yet) to make sure that the camera is recognized by your Raspberry Pi. Execute the following command at the terminal:

```
raspistill -o petcam1.jpg
```

By default, the `raspistill` utility waits for five seconds before taking a photo, so be sure to wait the appropriate time, and check your local directory to verify that the picture was taken.

Live Streaming on the Local Network

Now that you know the Pi is able to use the camera, the next steps are to use the camera in video mode, open a port on the network interface, and serve the video feed to anyone who knows the local IP address and port of your Raspberry Pi.

These steps require both the `raspivid` command and the open source VLC media player. If you don't already have the VLC player installed on your Raspberry Pi, execute the following command to install it:

```
sudo apt install -y vlc
```

Next, create an 800x600 H264 video feed at 12 frames/sec and open port 3141 on the Raspberry Pi to serve the video as a livestream using the Real Time Streaming Protocol (RTSP) with:

```
raspivid -o - -t 0 -w 800 -h 600 -fps 12 | cvlc -vvv stream:///dev/stdin --sout '#rtp{sdp=rtsp://:3141/}' :demux=h264
```

You might be more familiar with common TCP-based protocols such as HTTP and FTP, but RTSP is an alternative protocol used exclusively for streaming video and audio [2]. Because web browsers don't support RTSP, you need to use video tools such as VLC or QuickTime in order to view an RTSP

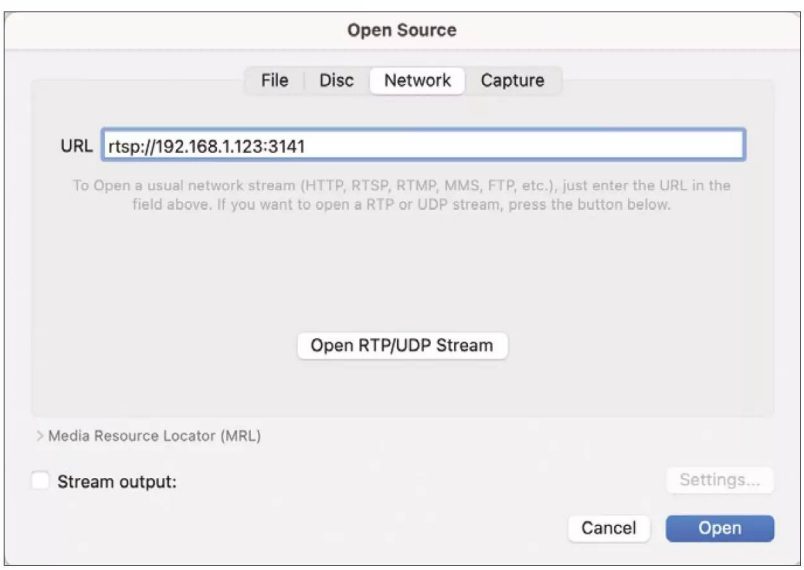


Figure 3: Use an RTSP client (like the VLC media player or QuickTime) on your home computer to test the video from your Raspberry Pi.

video stream. To view the video stream within your home network, you need to install an RTSP client on your desktop or laptop computer.

If you're using VLC, navigate to *Media | Open Network Stream* in order to open a dialog window that enables you to type in the IP address and port number for your video stream (Figure 3). The URL for the video stream must be in the following format:

```
rtsp://ip_address:port_number
```

Viewing the Stream While Away

Now that you've got the camera and the video streams working from inside the local home network, it's time to set things up to view the pet cam while you're away from home. Of course, to test this out, you'll need access that is different from your local LAN. You might want to connect over your cell phone's hotspot for testing purposes.

Most home routers today provide network address translation, which means that the hosts on your home network are not addressable from the Internet. One way around this is to go to the admin settings of your home router, open a particular port, enable port forwarding on the router, and assign the forwarded port to the specific device (and port) that you would like to access remotely. However, this approach is ill-advised in today's security environment, because anyone with a port scanner can scan your router, find the open port, and use it to attack your network.

Thankfully, free services exist today that allow you to securely host your own VPN service and connect to your personal devices located behind firewalls and LAN routers. A traditional "old school" VPN would allow you, while at home, to remotely connect and access the computers and servers at your job. Using a self-hosted mesh VPN service (in this case, I'm using Tailscale [3]), I can be away from home at my job (or anywhere for that matter) and connect to the devices inside my home. Although Tailscale does not have a free software license, the company that maintains it provides a no-cost version for personal and hobby projects [4].

In order to set up a mesh VPN using Tailscale, all you need to do is install the free Tailscale client on each device that you want to participate in your personal VPN. Then log in to each client that you want to connect in the personal VPN.

The command to install Tailscale on the Raspberry Pi is:

```
curl -fsSL
  https://tailscale.com/install.sh | sh
```

Once you've connected your devices over your self-hosted mesh VPN, you can test the connectivity between devices by sending a ping request to your home PC while outside of your home network. If the ping works, the devices are connected.

The final step is to use your RTSP client app just as before, but this time, swap out the local-only IP address with the mesh VPN address of your Raspberry Pi. Voila, a self-hosted, mesh-VPN-enabled pet cam (Figure 4)!

Conclusion

This simple exercise illustrates the power of the Raspberry Pi as a practical tool for customizing your home environment. If you don't have a pet, you can easily adapt these techniques to watch your yard or observe wildlife. If you really get ambitious, you could even integrate a motion detector or set up a Pi-compatible infrared camera for night vision. ■■■

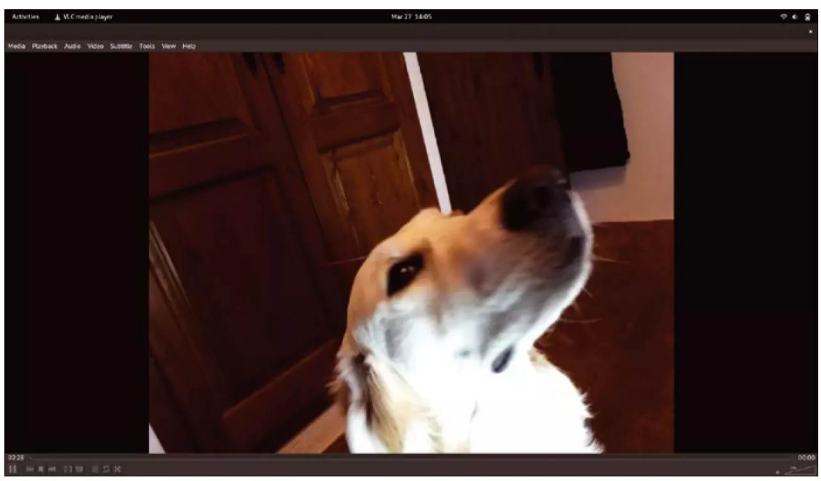


Figure 4: I always wonder what he's up to while I'm away at work. This is the face of an innocent puppy staring at a sausage behind the Raspberry Pi.

Info

[1] Pi Camera Module 2: *https://www.raspberrypi.com/products/camera-module-v2/*

[2] RTSP: *https://en.wikipedia.org/wiki/Real_Time_Streaming_Protocol*

[3] Tailscale: *https://tailscale.com/*

[4] Tailscale free plan: *https://tailscale.com/kb/1154/free-plans-discounts/*

Author

Bruce Hopkins is a technical writer, and is the author of the book *Bluetooth for Java*, by Apress Publishers.

Use gestures to browse a document on your Raspberry Pi

Hands Free

Have you found yourself following instructions on a device for repairing equipment or been halfway through a recipe, up to your elbows in grime or ingredients and then needed to turn or scroll down a page? Wouldn't you rather your Raspberry Pi do the honors? *By Bernhard Bablok*

This article is about the joy of tinkering, and the project I look at is suitable for all kinds of situations when your hands are full or just dirty. The hardware requirements turn out to be quite low: a Raspberry Pi, a screen, and a gesture sensor. My choice of sensor was the APDS9960 (Figure 1), for which you can get breakouts and an I2C connector for a low price at the usual dealers ($3.20-$7.50). However, you should note whether the sensor has soldered jumpers. The left jumper (PS) controls the power supply of the infrared lamp with the pin for positive supply voltage (VCC) and definitely needs to be closed. The right jumper (labelled 12C PU on the sensor in Figure 1) enables the pullups on the clock line (SCL) and the data line (SDA), which is superfluous on the Raspberry Pi; however, it doesn't hurt to have it.

Modern kitchens sometimes feature permanently installed screens. If you don't have one, go for a medium-sized TFT screen like the 7-inch Pi screen or a model by Waveshare (Figure 2). If you are currently facing the problem that the Raspberry Pi is difficult to get, as many people have, you can go for a laptop instead, which I talk about later in this article.

Figure 1: You can pick up the APDS9960 gesture sensor from the usual retailers for around $4.00.

Installing the Software

The Pi Image Viewer program is implemented in Python and is very minimalist. In fact, it is an image viewer that performs precisely one function: scrolling through an image in response to gestures. The software would even work with a small four-inch screen with a Raspberry Pi clamped behind it, but it would not be particularly user friendly.

You can pick up the software for a gesture-driven recipe book on GitHub [1] by cloning the repository and installing the software with the commands

```
git clone https://github.com/bablokb/
pi-image-viewer.git
cd pi-image-viewer
sudo tools/install
```

Additional information is provided in the installation instructions in the `Readme.md` file.

Implementation

The implementation is built on Blinka [2] for the sensor and PyGame [3] for the interface. PyGame is a game engine, but it is also suitable for other applications. Moving objects is understandably as easy as pie (groan) for PyGame. Instead of moving sprites, the software shifts the

Photo by Sebastian Dumitru on Unsplash

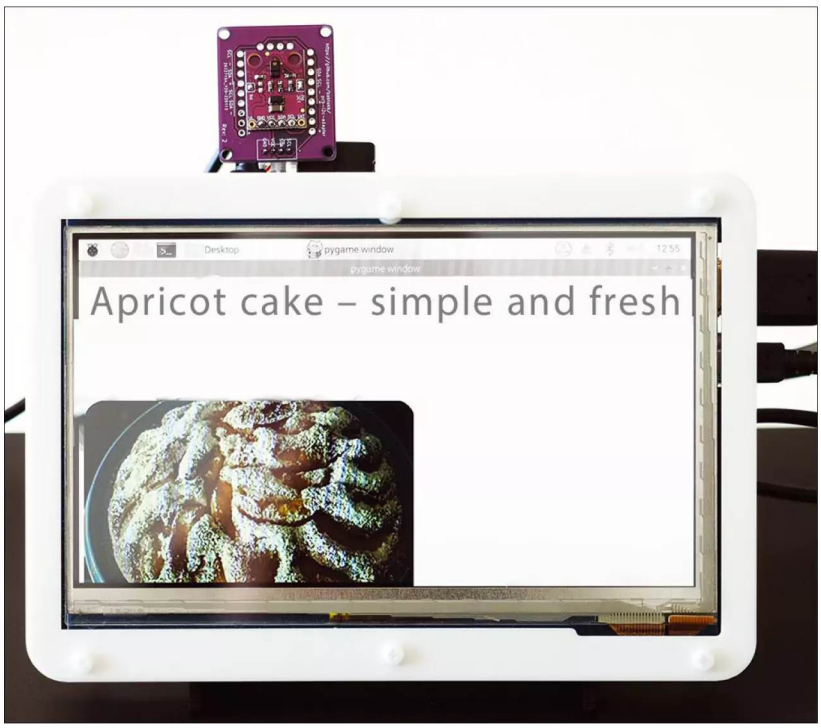


Figure 2: In the sample project, the gesture sensor sits above the Waveshare TFT screen.

image to show a different section each time (Figure 3).

In PyGame, rectangles stand in for both the screen window and the image. The window defines the global coordinate system, and its upper left corner marks the zero point; the (0,0) coordinate in turn determines the location relative to the screen. If the coordinates are (0,0), users will see the upper left part of the image (Figure 3, left).

If, on the other hand, the coordinates are negative, say (-50,-50), the top left corner is outside the window, and you see the bottom right area of the image (Figure 3, right). This arrangement might sound confusing at first, but moving the image toward the upper left (negative coordinates) makes the bottom right part of the image visible.

PyGame is controlled by events. The program processes key events for the four cursor keys (Listing 1, lines 12-17). Each key is backed up by a method that is responsible for moving in one of the four directions. To keep the code manageable, a key-value pair is defined up front for each direction (lines 2-8).

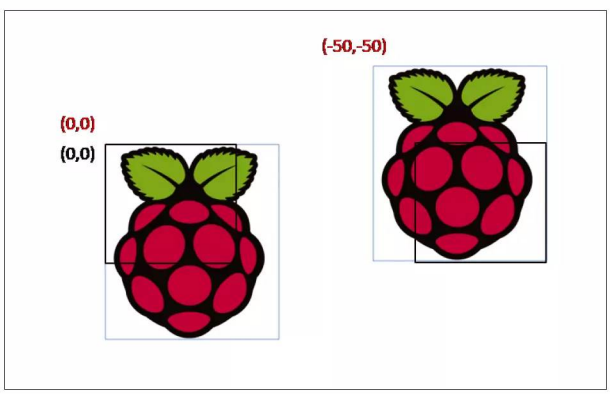


Figure 3: PyGame displays the image and screen as rectangles.

Listing 1: Keyboard Control

```
01 ...
02 self._MAP = {
03     K_RIGHT:  self._right,
04     K_LEFT:   self._left,
05     K_UP:     self._up,
06     K_DOWN:   self._down,
07     K_ESCAPE: self._close
08   }
09 ...
10
11 ...
12 for event in pygame.event.get():
13     if event.type == QUIT:
14       self._close()
15     elif event.type == KEYDOWN:
16       if event.key in self._MAP:
17         self._MAP[event.key]()
18 ...
```

Processing Gestures

Gesture processing is handled in a second thread that polls the sensor (Listing 2, line 4) and, from the detected gestures, simply synthesizes the matching key events for the PyGame main program (line 16), which closes the circle.

The program shown here with the gesture control does not completely solve the problem. You still need to convert your printed recipe into a (JPG) image, but you can easily scan or take a photo

Listing 2: Gesture Control

```
01 evnt = {}
02 while not self._stop.is_set():
03   time.sleep(0.1)
04   gesture = self._apds.gesture()
05   if not gesture:
06     continue
07   elif gesture == 0x01:
08     evnt['key'] = pygame.K_UP
09   elif gesture == 0x02:
10     evnt['key'] = pygame.K_DOWN
11   elif gesture == 0x03:
12     evnt['key'] = pygame.K_LEFT
13   elif gesture == 0x04:
14     evnt['key'] = pygame.K_RIGHT
15
16   event = pygame.event.Event(pygame.KEYDOWN,evnt)
17   pygame.event.post(event)
```

of a recipe book or grab a screenshot to do that. Fairly low resolutions are absolutely fine for the purposes of this application.

If the recipe is a PDF, the following one-liner will help:

```
convert -density 150 in.pdf ➥
        -append out.jpg
```

This command uses the `convert` command from the ImageMagick package, which is typically already in place. If not, just grab it with your distribution's package manager. The `-density` option lets you control the image resolution. If the PDF has multiple pages, the command arranges the pages one below the other. If you prefer horizontal scrolling, replace `-append` with `+append`. Two more parameters handle fine tuning: `-trim` removes the white border, whereas `-sharpen 0x1.0` sharpens the result.

You still need two things before you can start the image viewer with a double-click: a `pi-image-viewer.desktop` file, which registers the image viewer as a program for processing JPGs, and a file that stores the image viewer as the default display program. Both points are described in the Readme file for the GitHub project.

Figure 4: Laptop, MCP2221, and gesture sensor.

Laptop Instead of Pi

The image viewer and gesture control also work without a Pi on a normal laptop (Figure 4), because Blinka and PyGame run the same way on popular desktop operating systems. However, because these systems don't usually have a freely accessible I2C port, you might need to retrofit one on a USB-to-I2C bridge. The MCP2221 microchip does this easily and inexpensively for $3.00 and up [4] or with a Raspberry Pi Pico [5].

Conclusions

A few lines of PyGame code and a few lines of APDS9960 code, mostly copied from sample code online, is all it takes for this application. Because the key events are simulated, you can do without a keyboard. The principle can also be transferred to other hardware. For example, you can find low-cost displays without touch input. Instead of a full keyboard, a simple MPR121 keypad [6] connected by I2C might also do the trick. Just as the code in the image viewer translates gestures into strokes, it would translate touch events for the key sensor.

You can take this solution one step further with the *python3-evdev* library, which lets you generate arbitrary (system) key events, allowing you to control any program with gestures or by touch – not just those that are designed for touch control like the Pi Image Viewer.

Voice control is an alternative to gesture control and is now suitable for practical use on a Raspberry Pi with voice interface modules such as the Seeed ReSpeaker [7]. ■■■

Info

[1] Pi Image Viewer: *https://github.com/bablokb/pi-image-viewer*

[2] "CircuitPython for Raspberry Pi and MCUs" by Bernhard Bablok, *Linux Magazine*, issue 234, May 2020, *https://www.linuxpromagazine.com/Issues/2020/234/CircuitPython/*

[3] PyGame: *https://www.pygame.org*

[4] Adafruit guide to the MCP2221: *https://learn.adafruit.com/circuitpython-libraries-on-any-computer-with-mcp2221*

[5] Adafruit guide to the Pico as an I2C USB bridge: *https://learn.adafruit.com/circuitpython-libraries-on-any-computer-with-raspberry-pi-pico*

[6] MPR121 keypad: *https://www.sparkfun.com/products/retired/12017*

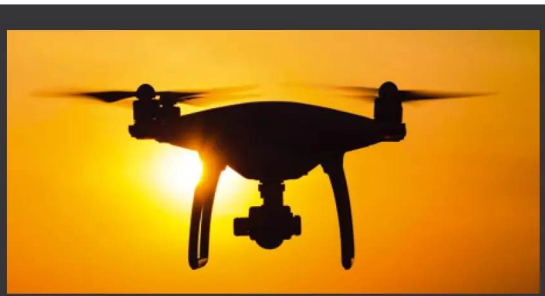
[7] Seeed ReSpeaker: *https://wiki.seeedstudio.com/ReSpeaker/*

Author

Bernhard Bablok works at Allianz Technology SE as an SAP HR developer. When he's not listening to music or out and about, he's busy with topics related to Linux, programming, and small-board computers. You can contact him at *mail@bablokb.de*.

Programming the DJI Ryze Tello drone with Python

Fleet Flyer



Drones are more fun if you can program the unmanned aerial vehicle yourself. The DJI Ryze Tello and Python make this possible. *By Martin Mohr*

In this article, I work with the DJI Ryze Tello drone, which is available on Amazon [1] and from other vendors for around $99 (£99/EUR100). Why this model? The drone was designed for educational use and has open interfaces for that purpose. In other words, you can develop programs for the drone in Scratch or Python.

To familiarize yourself with the drone's capabilities, you first need to download the app that gives you manual control of the drone. The drone's battery lasts about 13 minutes, and the control range is about 100 meters. Because it is intended for operation inside buildings, this range is fine. If you get lost, the built-in 720p camera with electronic image stabilizer lets you view your current location.

For more intensive use, a Tello with the indispensable boost combo add-on adds $49 (£39/EUR45) to the price. Among other things, you get two extra batteries and a propeller protection set. During your first programming attempts, the propeller guards will definitely help prevent serious damage (Figure 1). Further information about the DJI Ryze Tello can be found on the manufacturer's website [2]; the essential technical data is shown in Table 1.

Figure 1: The DJI Ryze Tello with the highly recommended propeller guards from the accessories range. Source: DJI

Table 1: DJI Ryze Tello Overview

Feature	Spec
Dimensions	98.0x92.5x41.0mm
Weight	80g, including propeller and battery
Battery	1.1Ah/3.8V (removable)
Connector	Micro-USB port for charging
Other	Rangefinder, barometer, LED, vision system, WiFi 802.11n, 720p live streaming
Performance	
Max. flight range	100m
Max. flight altitude	30m
Max. speed	8m/s (18mph)
Max. flight time	13 min
Camera	
Field of view (FOV)	82.6 degrees
Photography	5Mpx (JPG, 2592x1936px)
Video	HD720p30 (MP4)

Software Installation

To download the app for the drone, search for *Tello* in the Google or Apple app store. The app also lets you update

Lead Image © Leonid Eremeychuk, 123RF.com

Updating the Firmware

You never know exactly how long the drone has been sitting around in a warehouse, so before doing anything else, it's a good idea to update the device's firmware from the app on your smartphone. To begin, connect to the drone and then switch to the app's settings with the gear icon. When you get there, tap *More*, then the three dots (Figure 2), and *Update* in the Firmware Version line. You will then see the release notes and can start the update process by tapping *Update*.

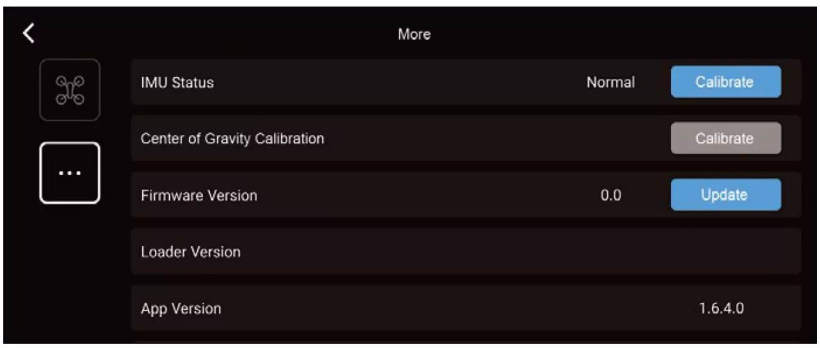


Figure 2: Update the drone's firmware with the Tello app.

the drone's firmware. Instructions on how to do this can be found in the "Updating the Firmware" box.

To connect to the drone in the app, first turn on the drone and wait until the LED flashes, which indicates it is starting a separate WiFi network to which you need to connect. Now launch the Tello app and connect to the drone. The app tells you the steps required. If everything worked, you will see the camera image in the app. The drone can now be launched with the lift-off icon at top left.

You can control four degrees of freedom in the app: forward and backward, left and right, up and down, and rotation about the *z* axis in both directions. You will encounter these four degrees of freedom again when creating programs. To make sure everything is working properly, go on a small test flight before you try controlling the drone with Python.

Before you start programming, you might want to install a suitable integrated development environment (IDE). PyCharm Community by JetBrains is a good choice for this project. You will find versions for different operating systems on the project website [3]. The installation completes the preparations, and you can now proceed to write your first program for the drone.

Connection Test

To create a program with PyCharm, select *File* | *New Project*. Leave all the default settings as is except for Location: Enter a meaningful project name in this field as the last component of the path. After pressing *Create*, the IDE directly creates a `main.py` file, where you will save your own program after deleting the sample code in the file.

To use your scripts, you need to include a drone control library [4] in your

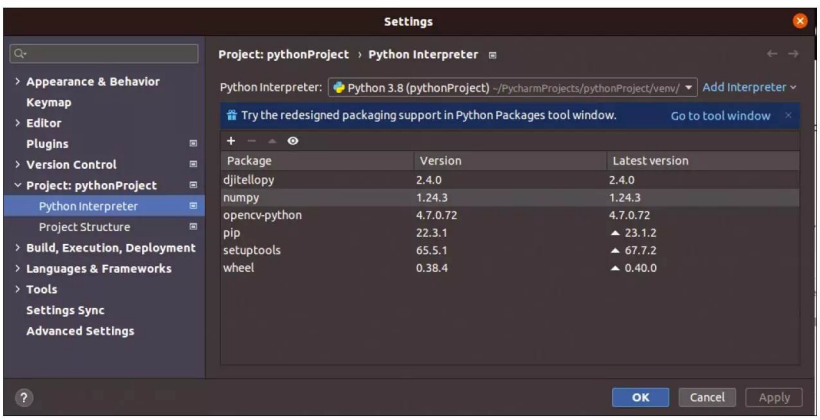


Figure 3: The *djitellopy* library in PyCharm allows you to control the drone.

projects by selecting *File* | *Settings* | *Project:* < *name* > | *Python Interpreter* and pressing the plus icon to install the *djitellopy* library (Figure 3).

The program in Listing 1 connects to the drone and displays the temperature and battery level. These two readings are very important for drone operations. If the charge level is too low, you do not want to fly. If the electronics or the battery are overheating, it's time for a short break. To start the program, just click the green arrow at the top of the IDE. Listing 2 shows the program output.

Depending on your computer's feature set, if nothing is working you might only see an error message. The program must connect to the drone's WiFi to achieve control. Most desktop PCs do not have a WiFi interface. A simple and inexpensive solution to this problem is a USB WiFi adapter. I had one [5] in my tinkering box that worked perfectly with Ubuntu.

Ready to Start

You now know how to connect the computer to the drone and query simple status information, so it's time for a small test flight. With the program in Listing 3,

Listing 1: Opening a Connection

```
from djitellopy import tello
drone = tello.Tello()
drone.connect()
print("Temperature: "+drone.get_temperature())
print("Battery: "+drone.get_battery())
```

Listing 2: Connection Output

```
[INFO] tello.py - 122 - Tello instance was initialized.
       Host: '192.168.10.1'. Port: '8889'.
[INFO] tello.py - 437 - Send command: 'command'
Temperature: 60.0
Battery: 93
[INFO] tello.py - 461 - Response command: 'ok'
Process finished with exit code 0
```

Listing 3: Flying the Drone

```
from djitellopy import tello
drone = tello.Tello()
drone.connect()
drone.takeoff()
drone.move_forward(30)
drone.land()
```

launch the drone, let it fly forward a few centimeters, and then land. Controlling the drone always follows the same pattern. You create an object for the drone and call its methods to perform the specific actions.

To discover what actions the object supports, mouse over a method and press Ctrl + Left mouse button to access the library's source code, where you can snoop around a bit. Alternatively, type *drone.* and scroll through the menu (Figure 4). I did look for better documentation for the library, but my search turned up nothing.

I need to talk about one more special method for the drone object: `send_rc_control` specifies motion speeds for the drone's individual degrees of freedom. The permissible values range from `-100` to `100`. The method supports four parameters:

- Speed left or right
- Speed forward or back
- Speed up or down
- Rotational speed clockwise or counterclockwise

This method is a simple way to control the drone remotely.

And ... Action

To complete the picture of the DJI Ryze Tello's feature set, take a look at how you access the camera. To begin, you need to include the *opencv-python* library in the project. Proceed in the same way as described for the *djitellopy* library. The very powerful OpenCV [6] library provides extensive functions for image processing.

The program in Listing 4 opens the video stream from the camera and displays it in a new window. Note that the camera displays the many individual images quickly one after the other. To change the resolution of the images, remove the comment hash (`#`) at the start of line 8.

Good to Know

In testing, the DJI Ryze Tello turned out to be a light-loving device. If it is too dim in your office, the drone will not work properly and will constantly output messages like *error No valid imu*. In flight, the camera is used for navigation, and if the camera does not have enough light, the image becomes unusable.

As soon as you take a closer look at the Tello's camera, you will notice that image processing is computationally intensive, so reducing the resolution of the videos to counteract this problem is helpful. Often, even images with a considerably lower resolution are sufficient for further processing.

Nothing is more annoying than a drone that crashes because of a lack of power, so you should check the battery charge level in your program before each start and automatically exit the program if it is too low.

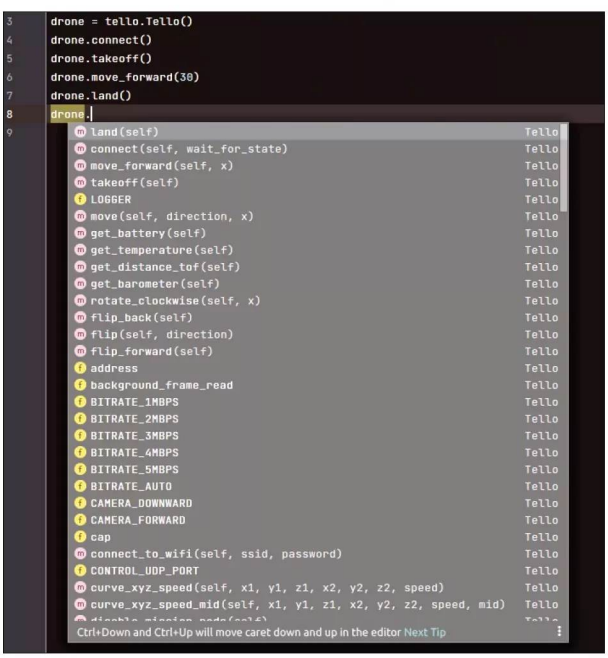


Figure 4: All available method calls for the DJI Ryze Tello.

Conclusions

During this test, the drone suffered quite a few collisions with all sorts of objects in the office. Two propellers disappeared never to be seen again after a crash. A houseplant standing in the flight path was involuntarily cut back – and it's surprising how far shredded leaves fly.

During your first flight attempts, take meticulous care to remove any objects potentially standing around in the flight path area and fly the drone in as large a space as possible. At the end of the day, coming to grips with the DJI Ryze Tello and learning more about the little flying machine is massive fun. The only drawback of the compact drone is the relatively poor battery capacity, which will force you to take many breaks. ■■■

Info

[1] Tello drone on Amazon: *https://www.amazon.com/dp/B07BDHJJTH*

[2] Tello drone website: *https://www.ryzerobotics.com/tello*

[3] PyCharm download: *https://www.jetbrains.com/pycharm/download/*

[4] djitellopy Python library: *https://github.com/damiafuentes/DJITelloPy*

[5] USB WiFi adapter: *https://www.amazon.com/Edimax-EW-7811Un-Wi-Fi-Nano-Adapter/dp/B08D3DBP55/*

[6] OpenCV: *https://opencv.org*

Author

Martin Mohr has experienced the complete development of modern computer technology live. After completing university, he mainly developed Java applications. The Raspberry Pi helped him rediscover his old love of electronics.

Listing 4: Video Stream

```
01 from djitellopy import tello
02 import cv2
03 drone = tello.Tello()
04 drone.connect()
05 drone.streamon()
06 while True:
07   image = drone.get_frame_read().frame
08   #image = cv2.resize(image, (200, 200))
09   cv2.imshow("Tello Drone", image)
10   cv2.waitKey(1)
```

Artificial intelligence on the Raspberry Pi

Learning Experience

You don't need a powerful computer system to use AI. We show what it takes to benefit from AI on the Raspberry Pi and what tasks the small computer can handle. *By Erik Bärwaldt*

Artificial intelligence (AI) is on everyone's minds, not least because of chatbots and the ChatGPT text generator. Of course, the capabilities that AI has developed go far beyond chats with a chatbot on countless websites. For example, AI can be used to process acoustic speech signals, and it is the precondition for autonomous driving. Some applications – and generating AI models – require computers with powerful processors and a generous helping of storage space and RAM. Small computers like the Raspberry Pi, on the other hand, are more likely to benefit from ready-made methods and applications that draw on AI for their implementation.

All of these processes are founded on machine learning (ML), which itself is based on self-adapting algorithms that process information from reference data. Deep learning, as a subset of machine learning, uses artificial neural networks that comprise multiple hierarchical processing layers. The neurons of the network are interconnected in multiple ways, with the individual layers increasingly abstracting the reference data they receive. Solutions or actions are then derived from the results.

TensorFlow

TensorFlow [1], released by Google AI in 2015, is an open source framework that aims to simplify the development and training of deep learning models. It supports numerous programming languages and can be used for various purposes, such as the linguistic data processing in various Google services. It can also be used to recognize and classify patterns and objects in images.

TensorFlow Lite [2] is a solution designed specifically for the embedded and Internet of Things (IoT) spaces, addressing the hardware limitations that exist there. The version does not require Internet access because it does not send data to servers, which is a good thing not just in terms of data protection, but to avoid latency and reduce energy requirements. TensorFlow Lite is not suitable for training models, but it can apply pre-trained models. The framework uses reduced model sizes, but the models are still useful for various cases. Google also provides a web page for generating models on the basis of object classifications; you can use these to create your own model and then deploy it in TensorFlow Lite.

To detect objects on the Raspberry Pi with TensorFlow Lite, you need a fourth generation device with a connected camera. Although some third generation Raspberry Pis are suitable for AI applications in principle, they are very slow because of their hardware limitations, especially in terms of RAM. When it comes to the camera for AI applications, it

Lead Image © Charles Taylor, 123RF.com

doesn't matter whether you choose one designed specifically for the small computer that connects directly or use an arbitrary USB camera. If you prefer an external camera, however, make sure the Raspberry Pi OS supports your choice.

The first step is to download the latest 64-bit release of Raspberry Pi OS [3] and transfer it to a microSD card with at least 16GB. To do so, either use a graphical tool such as balenaEtcher or enter the following command at the prompt:

```
dd if=</path/to/
      operating system image>
   of=/dev/mmcblk0 bs=4M
```

Make sure the microSD card supports fast read and write mode. It should at least comply with the Class 10 specification. Boot your Raspberry Pi from the microSD card and turn to the basic graphical configuration of the system. Run the usual commands to update the operating system:

```
sudo apt-get update
sudo apt-get upgrade
```

If you want to use an external camera for object detection, connect it to the Pi and install an application that accesses the camera on the system, such as the Cheese graphical program or the `fswebcam` command-line tool. Also, if you are using an external USB camera, make sure that its resolution is sufficient: The fewer clear-cut distinguishing features the objects to be detected have, the higher the camera resolution needs to be. If you use the Raspberry Pi's own camera, it must be connected to the camera port of the single-board computer before you boot the system for the first time.

Installation

Because of the fast pace of technical developments in the field of deep learning and the many components required, installing TensorFlow Lite on the Raspberry Pi is anything but trivial and is by no means something you can do quickly. Constantly changing dependencies and new versions make it difficult to give universal guidance. However, you will definitely want to make sure that you are using the 64-bit variant of Raspberry Pi OS. To verify that you have the correct version of the operating system, enter:

```
uname -a
```

The output must include the `aarch64` parameter. If it is missing, you are running the 32-bit variant of Raspberry Pi OS, which rules out any meaningful deployment of TensorFlow Lite. You also need the correct matching version of the C++ compiler (GCC) in place. To check, type

```
gcc -v
```

at the prompt; the output must contain `--target=aarch64-linux-gnu`.

If these conditions apply, the next step is to adjust the swap size of the system. By default, only 100MB are reserved as a swap partition on the Raspberry Pi 4. You will want to increase this value to 4GB if you are using a Raspberry Pi 4 with 4GB of RAM. Unfortunately, Raspberry Pi OS limits swap memory to a maximum of 2GB, and you will need to edit two files to be able to continue. The first task is to disable the swap space

```
sudo dphys-swapfile swapoff
```

and open `/sbin/dphys-swapfile` in an editor to look for the `CONF_MAXSWAP` parameter (Figure 1). Set the value specified to its right to `4096` and save your change. In a second file, `/etc/dphys-swapfile`, look for the `CONF_SWAPSIZE=100` option, and replace the value of `100` with `4096` for a Raspberry Pi 4 with 4GB of RAM. For a device with only 2GB of RAM, the swap size should be set to 4096MB, whereas 2048MB is fine for a model with 8GB of RAM. After saving the modified file, enable the new swap size and check it by running:

```
sudo dphys-swapfile swapon
free -m
```

If everything meets the specifications, you can install TensorFlow Lite. The software will work with Python, but the C++ API libraries are preferable because of the far superior processing speed. Listing 1 shows how to get TensorFlow Lite v2.6.0, including all of its dependencies, and how to compile with C++. The build takes about half an hour.

After compiling, you need to install modified TensorFlow Lite FlatBuffers [4]; otherwise, numerous GCC error messages will appear. Listing 2 shows you how to remove the old FlatBuffers and replace them with a bug-fixed version.

This change is essential because the original TensorFlow FlatBuffers no longer work with current GCC versions. The bug-fixed variant replaces the obsolete serialization libraries with adapted versions.

Options

TensorFlow Lite offers the option of recognizing objects with pre-built

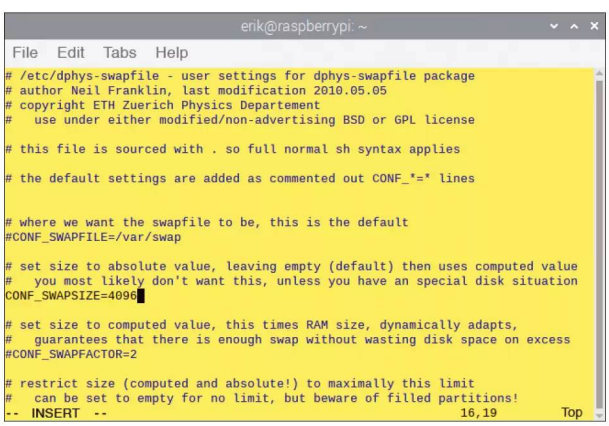


Figure 1: Raspberry Pi OS initially requires some adjustments for use with AI applications.

Listing 1: Installing TensorFlow Lite

```
$ sudo apt-get install cmake curl
$ wget -O tensorflow.zip https://github.com/tensorflow/
   tensorflow/archive/v2.6.0.zip
$ unzip tensorflow.zip
$ mv tensorflow-2.6.0 tensorflow
$ cd tensorflow
$ ./tensorflow/lite/tools/make/download_dependencies.sh
$ ./tensorflow/lite/tools/make/build_aarch64_lib.sh
```

Listing 2: Installing FlatBuffers

```
$ cd tensorflow/lite/tools/make/downloads
$ rm -rf flatbuffers
$ git clone -b v2.0.0 --depth=1 --recursive
   https://github.com/google/flatbuffers.git
$ cd flatbuffers
$ mkdir build
$ cd build
$ cmake ..
$ make -j4
$ sudo make install
$ sudo ldconfig
$ cd ~
$ rm tensorflow.zip
```

models that can be classified. However, you can only create models in the "full-fledged" TensorFlow variant. TensorFlow Lite and a Raspberry Pi are not suitable because you need masses of compute power. The recommended approach is therefore to create new models from reference data with GPU processors because they will perform the required computations far faster than CPUs. Also, the models generated in TensorFlow are not compatible with TensorFlow Lite. You will need to convert them for use in the Lite variant. Google has already created numerous models for TensorFlow Lite that you can deploy on the Raspberry Pi. The TensorFlow project website provides detailed information [5] on how to convert models to the TensorFlow Lite format.

OpenCV

The Open Computer Vision Library (OpenCV) [6] has another set of libraries that you can use on your Raspberry Pi. OpenCV is used for gesture, face, and object recognition and classification. The OpenCV deep neural network (DNN) module works with pre-trained networks for this purpose and can be used in combination with TensorFlow Lite. To install OpenCV on the Raspberry Pi, though, you need to resolve a large number of dependencies, and you need to specify manually a large number of flags during the build. This difficulty prompted Dutch AI specialists at Q-engineering [7] to publish a freely available and BSD-licensed script on GitHub that lets you work around these steps. To install and run this OpenCV script, enter:

```
$ wget https://github.com/Z
  Qengineering/Install-OpenCV-Z
  Raspberry-Pi-64-bits/raw/main/Z
  OpenCV-4-5-5.sh
$ chmod 755 ./OpenCV-4-5-5.sh
$ ./OpenCV-4-5-5.sh
```

As a final step, you need to integrate the graphical Code::Blocks integrated development environment (IDE) [8] into your system (Figure 2). With its help, you can then use TensorFlow Lite and OpenCV to recognize and classify objects by drawing on various sample networks. These capabilities apply not only to photos, but also to livestreams from the connected camera. Code::Blocks supports the C and C++ programming languages and is therefore ideally suited for AI applications. The command

```
sudo apt-get install codeblocks
```

installs the package and automatically creates a starter on the desktop and in the Raspberry Pi OS menu system.

Examples

After completing the installation, you can test some sample scenarios by drawing on a number of prefabricated and trained code examples from Q-engineering; all of these achieve very good results on the Raspberry Pi 4, even in livestreams [9]. Code::Blocks is used here, too, and it even provides slide shows of screenshots in the tutorials to help newcomers gain some initial experience with AI applications [10]. Instead of the sample photos and MP4

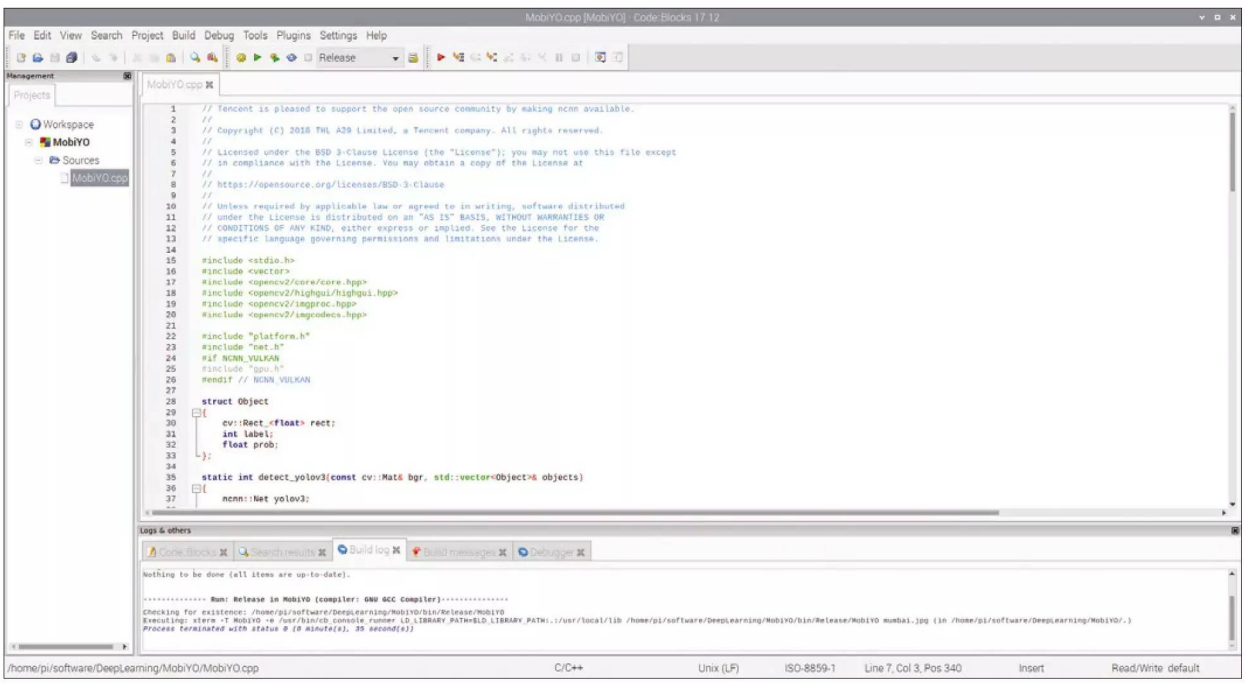


Figure 2: The Code::Blocks IDE helps you use AI models.

videos included in the bundle, you can use your own pictures or video files from the Raspberry Pi camera. All you need to do is copy them to the appropriate directories and specify them as parameters in Code::Blocks (Figure 3).

Generating Your Models

Because custom models cannot be trained on small computers, Google offers a web-based tool [11] to help in the creation of models. The tool is suitable for various model types and outputs them as files in the TensorFlow format so that you can use the models in the Lite variant after converting. Please note, however, that generating a model for object recognition (e.g., on images and photographs) means uploading several hundred sample images. The sample images also need to be high resolution to achieve high accuracy levels later. You need to schedule several hours to work with the tool (Figure 4).

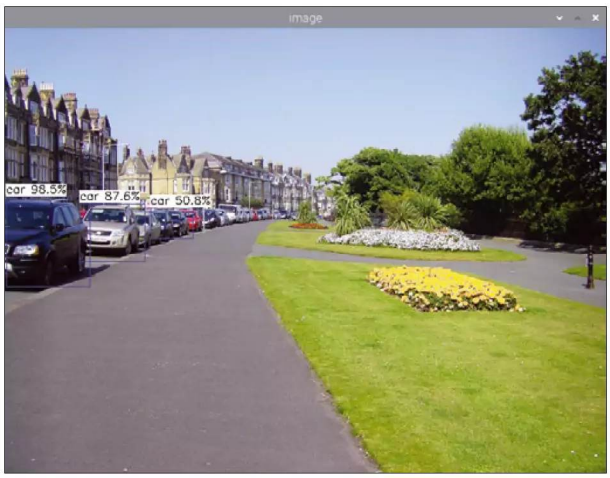


Figure 3: The object recognition elements are shown in the original image along with the percent likelihood of correct recognition.

Conclusions

AI applications with TensorFlow Lite and OpenCV have long been considered established tools and are suitable for production use. However, installing the individual libraries and frameworks on the Raspberry Pi involves a fair amount of time and overhead – especially because the documentation is often either outdated or lacking. For this reason alone, it makes sense to check out recent tutorials and examples to familiarize yourself gradually with AI applications on the Raspberry Pi. ■■■

Info

[1] TensorFlow: *https://www.tensorflow.org/*

[2] TensorFlow Lite: *https://www.tensorflow.org/lite*

[3] Raspberry Pi OS: *https://www.raspberrypi.com/software/operating-systems/#raspberry-pi-os-64-bit*

[4] FlatBuffers: *https://flatbuffers.dev/*

[5] Model conversion: *https://www.tensorflow.org/lite/models/convert*

[6] OpenCV: *https://opencv.org/*

[7] Q-engineering: *https://qengineering.eu/*

[8] Code::Blocks IDE: *https://www.codeblocks.org/*

[9] Code examples: *https://github.com/qengineering*

[10] Tutorial: *https://qengineering.eu/opencv-c-examples-on-raspberry-pi.html*

[11] Teachable Machine image model: *https://teachablemachine.withgoogle.com/train/image*

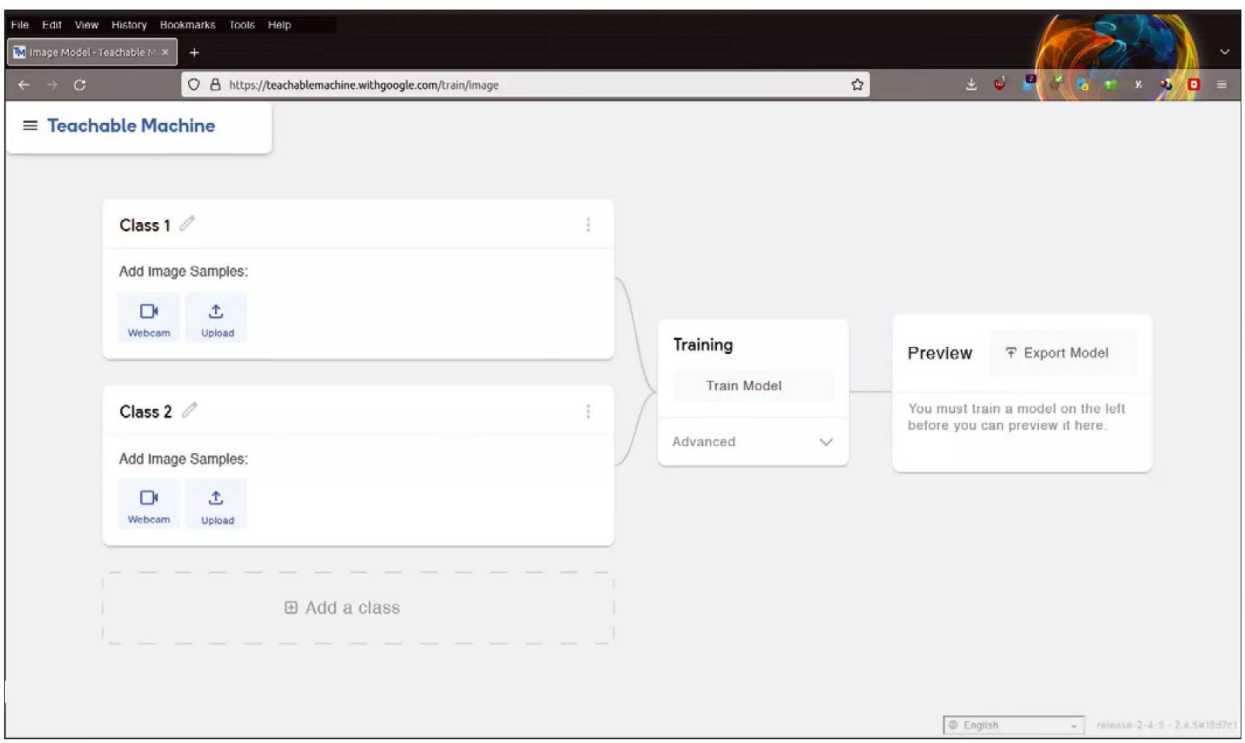


Figure 4: You can create your own models with a web-based tool.

Creating home automation devices with ESPHome

Automatic Home

With an ESP32 or Raspberry Pi Pico W microcontroller board, you can easily create your own home automation devices. Thanks to ESPHome, you don't even have to be a programmer. *By Koen Vervloesem*

Many home automation devices can be controlled through WiFi, but often these devices have limitations. For example, they might only work through the manufacturer's cloud service, they might be difficult to integrate with your own home automation system if you prefer to do everything local, they might lack advanced functionality, or they might be difficult to update.

Luckily, you can install alternative firmware on many existing or homemade devices. In this article, I introduce you to ESPHome [1], which supports numerous devices with an ESP32, ESP8266, or RP2040 microcontroller (the chip in the popular Raspberry Pi Pico W), although ESPHome support for the RP2040 is still in development. In the examples in this article, I'll use the Raspberry Pi Pico W. However, if you encounter any issues with your own projects, I recommend an ESP32 development board.

With ESPHome, you can create your own home automation devices with a supported microcontroller board that you connect to LEDs, sensors, or switches. What sets ESPHome apart from other solutions like Arduino [2] or MicroPython [3] is that you don't need to program. Instead, you configure which components are connected to which pins on the board. ESPHome then generates the necessary C++ code and compiles it into firmware that you can install on the device (see also the "Replacing Firmware on Commercial Devices" box).

Replacing Firmware on Commercial Devices

You can replace the existing firmware on commercial devices with ESPHome to gain full control over a device and use it in ways that the manufacturer hasn't anticipated. Two popular brands that have easy-to-flash devices are Shelly [4] and Sonoff [5]. A website [6] hosts more than 300 ESPHome device configuration templates that can help you get the most out of them. Note that often you'll need special hardware to flash your own firmware to these devices, at least the first time – afterward you can update them through WiFi. You'll need a USB-to-TTL adapter and to connect the pins of the adapter to the appropriate GPIO pins on the device. This isn't always a straightforward process (Figure 1).

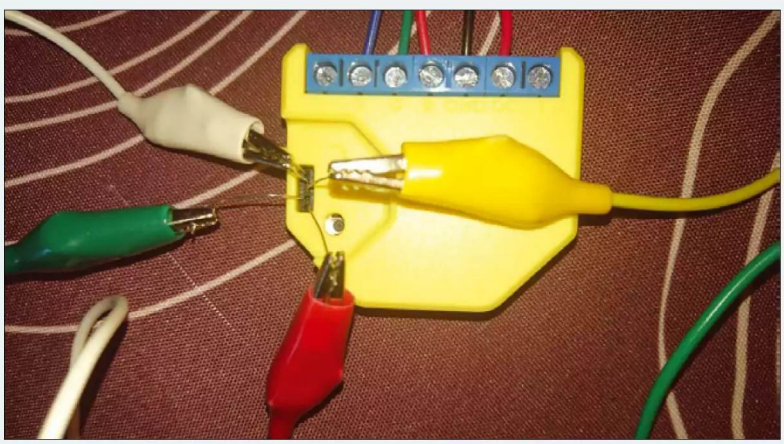


Figure 1: Crocodile clips and cut resistor leads saved the day when I wanted to flash ESPHome to this Shelly RGBW2 WiFi LED controller.

Lead Image © Valeriy Kachaev, 123RF.com

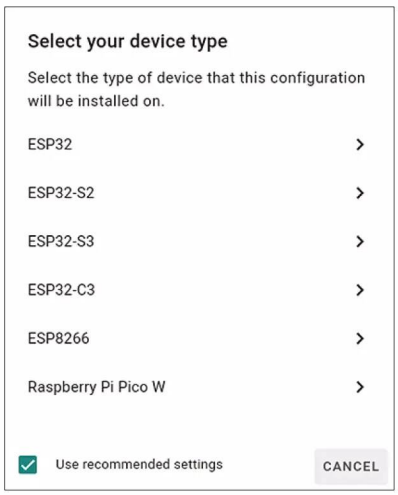


Figure 2: Select the type of device on which to install ESPHome.

Installing ESPHome

ESPHome is a Python program, and most Linux distributions already have Python installed by default. You should first confirm that you have at least version 3.9 installed, by running the command

```
$ python --version
Python 3.9.15
```

If your Python version is older, consider upgrading your distribution, or deploy the ESPHome Docker image [7].

If the Python version looks good, create a virtual environment to contain ESPHome and its dependencies:

```
$ python -m venv esphome_venv
$ source esphome_venv/bin/activate
```

Once you're in the Python virtual environment, install the ESPHome package from PyPI:

```
$ pip install esphome
```

After the installation is complete, enter

```
$ esphome version
Version: 2023.6.5
```

to confirm that ESPHome has been installed successfully.

Creating a Project with the Dashboard

A directory in which you store all of your ESPHome projects is recommended. Suppose you call this directory `config`. To start the ESPHome dashboard and point it to this directory, run:

```
$ esphome dashboard config/
```

This command starts a web server on *http://0.0.0.0:6052*, which you should be able to open in your web browser. If you already have ESPHome devices on your network, the dashboard will automatically discover them.

Next, click *New Device* at the bottom right corner, and then *Continue*. Give your device a name and enter the SSID and password for the WiFi network to which you want your device to connect, then click *Next* and select your device type (Figure 2).

In this example, choose *Raspberry Pi Pico W*; for an ESP32 or ESP8266 you also need to select the specific board. The dashboard then creates a minimal configuration and shows an encryption key that you can use to allow the ESPHome device to communicate with Home Assistant [8], a popular open source home automation gateway developed by the same team behind ESPHome. Finally, click *Install*.

You can use several methods to install ESPHome to your device, but not all of them are supported by every device. Because no ESPHome firmware is running on the device yet, the first method (over WiFi) is not possible; the *Plug into the computer running ESPHome Dashboard* choice isn't available either. You can always choose *Manual download*, which has instructions on how to accomplish the installation (Figure 3).

For the Raspberry Pi Pico W, you'll need to disconnect the board from USB, hold the *BOOTSEL* button while reconnecting the board, and then release the

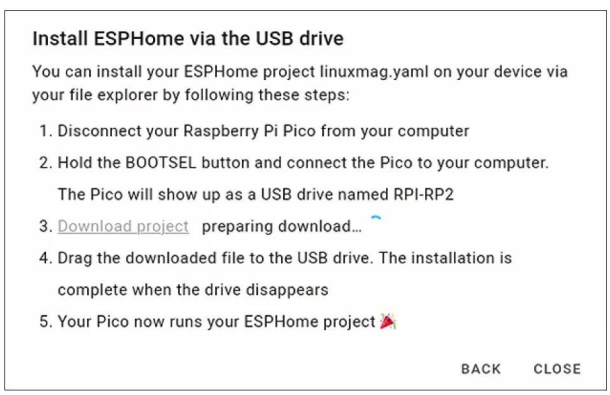


Figure 3: The ESPHome dashboard has excellent instructions for every step of the installation.

Listing 1: Pi Pico W Default Config

```
esphome:
  name: linuxmag
  friendly_name: linuxmag

rp2040:
  board: rpipicow
  framework:
    # Required until https://github.com/platformio/
      platform-raspberrypi/pull/36 is merged
    platform_version: https://github.com/maxgerhardt/
      platform-raspberrypi.git

# Enable logging
logger:

# Enable Home Assistant API
api:
  encryption:
    key: "7wFO19sSMAkGXO08l+wX4u53hBz/a1Ha+9bAdouUjo8="

ota:
  password: "20e3778465f1c5b147f8645dc237b146"

wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password

  # Enable fallback hotspot in case wifi connection fails
  ap:
    ssid: "Linuxmag Fallback Hotspot"
    password: "DVaDAPFJN5cA"
```

button, which causes a USB drive named RPI-RP2 to appear in your file manager. Now, click *Download project* and drag the `.uf2` file to the USB drive. Once the drive disappears, the board runs your ESPHome firmware, and you can click *Close*.

Default ESPHome Configuration

In the ESPHome dashboard, click *Edit* in the box representing your device to open your device configuration in a web editor. The configuration file is written in YAML [9], with various key-value pairs for different options (Listing 1).

As you can see, this configuration file sets the device name and its friendly name, as well as the platform and board. It then enables logging and the Home Assistant API, sets a password for over-the-air (OTA) updates, and configures WiFi credentials and a fallback hotspot in case the WiFi connection fails. If a failure happens, you can connect with your mobile phone to the hotspot of the device to reconfigure the network. The WiFi credentials are stored in a separate file, `secrets.yaml`, which prevents accidental exposure of sensitive information when sharing your device configuration with others.

Note that if you don't use Home Assistant, you should remove the `api` line and the two lines that follow; otherwise, your ESPHome device keeps waiting for a connection from Home Assistant. If no connection is established within 15 minutes, the device will assume that something's wrong and reboot.

Listing 2: Blinking LED

```
output:
  - platform: gpio
    pin:
      number: 32
      mode: output
    id: LED

interval:
  - interval: 1000ms
    then:
      - output.turn_on: LED
      - delay: 500ms
      - output.turn_off: LED
```

Blinking the Built-In LED

Now you can modify this configuration in the web editor or in your favourite desktop or command-line editor. The configuration file is saved in the `config` directory you created. In the next exercise, make the board's built-in LED blink by adding the configuration shown in Listing 2.

Make sure to use the correct indentation because spaces are important in YAML. This configuration adds an `output` component from the `gpio` platform on pin `32`, which corresponds to the built-in LED on the Raspberry Pi Pico W. Additionally, an `interval` component is defined that is triggered every `1000ms`. On each trigger, it turns on the output with `id: LED`, waits `500ms`, and then turns off the same output.

After saving the file (in the web editor at the top right), click *Install* in the dropdown menu of the node and choose your installation method. This time you can choose *Wirelessly*, because your device is already running ESPHome and is connected to your WiFi network. Your device doesn't even need to be connected to your computer's USB port any more. Your YAML configuration is now transformed into C++ code and compiled. If you see the message *INFO Successfully compiled program*, the dashboard will upload the new firmware. Once your device reboots, the LED starts blinking.

Adding Your Device to Home Assistant

If you're running Home Assistant on your home network, your ESPHome device will be recognized automatically. In your Home Assistant dashboard, click *Notifications* in the sidebar and then *Check it out* at the *New devices discovered* message. You will see the name you assigned to your ESPHome device (Figure 4). Click the *Configure* button

Listing 3: Remote LED Control

```
switch:
  - platform: gpio
    pin:
      number: 32
      mode: output
    name: LED
```

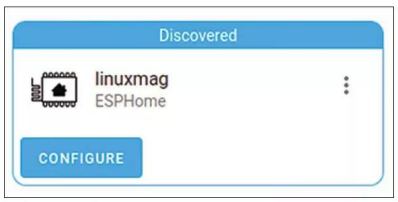


Figure 4: Home Assistant automatically discovers ESPHome devices on your network.

and then *Submit* to add the ESPHome device to Home Assistant.

You will be asked to enter the device's encryption key. Go to the ESPHome dashboard and find the key in the device's YAML code. Alternatively, click on the three dots in the box representing your device, then *Show API Key*. Next, click *Copy* and paste the key in the *Encryption key* field of Home Assistant. After clicking *Submit*, optionally choosing an area, click *Finish* to complete the process. The device is added to Home Assistant.

Remotely Control the LED

Now that you have configured your device to blink its LED and added it to Home Assistant, you might want to control it remotely. Instead of having the LED blink automatically, modify the configuration to allow you to control the LED from Home Assistant's dashboard. The required changes are simple: In the YAML configuration file, remove the entire `interval` block, change the `output` key to `switch`, and change the `id` key to `name`. You can find the modified configuration (without the defaults that you leave unchanged) in Listing 3.

After installing the firmware on your device, you can control the built-in LED from Home Assistant's dashboard (Figure 5). Because you have defined the LED as a `switch` component, you can turn it on and off.

Adding Sensors

Controlling a simple LED is relatively easy with alternatives like an Arduino sketch or some MicroPython code. However, things become more complex when you start connecting sensors. This is where ESPHome shines. The ESPHome website provides documentation for various supported sensors [13], each with simple YAML examples.

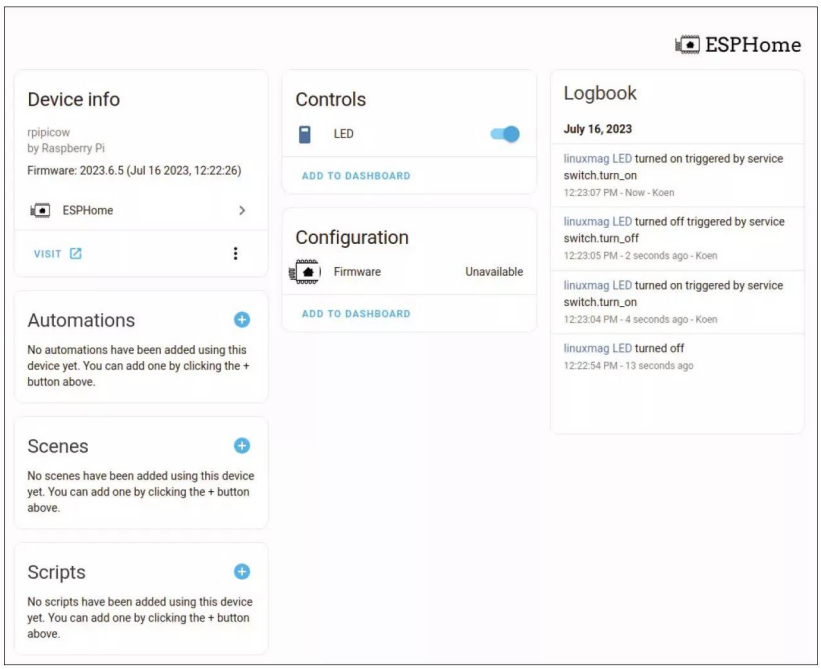


Figure 5: Control the LED on your ESPHome device from within Home Assistant.

Take the MH-Z19 CO_2 sensor, for example. This sensor is useful to have at home, especially considering the importance of ventilation in the fight against viruses. The worse the ventilation in a home, the higher the concentration of CO_2. Not only is CO_2 concentration – $[CO_2]$ – a good indication of the need for ventilation, it has various health hazards on its own.

Connecting the CO_2 Sensor

The ESPHome documentation provides instructions on how to connect and configure the MH-Z19 sensor [14]. First, you disconnect your Raspberry Pi Pico W from power and put it on a breadboard. You will be using four pins on the MH-Z19: VIN, GND, RX, and TX. Their names are listed on the bottom of the sensor board. Additionally, consult the Raspberry Pi Pico W pinout [15] or the pinout of the other microcontroller board you're using.

Connect the VIN pin of the sensor to the VBUS (which receives 5V from the USB power supply) of the Raspberry Pi Pico W, GND to GND, RX to GP4, and TX to GP5 (Figure 6). The configuration for the sensor is shown in Listing 4.

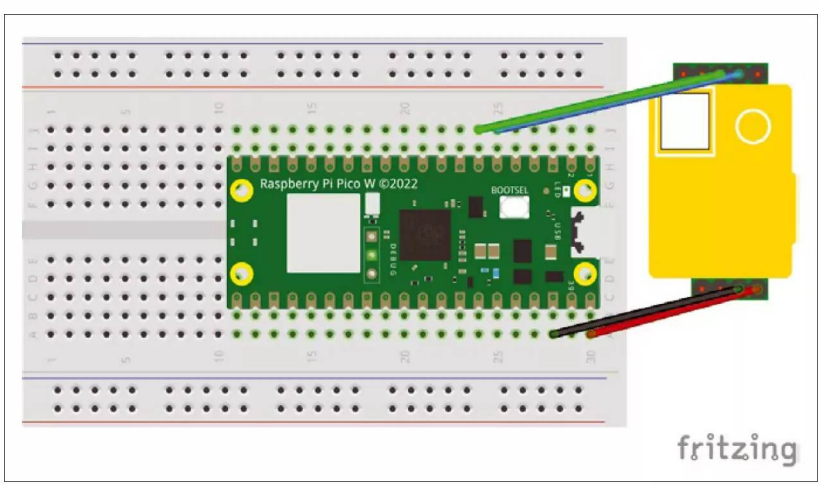


Figure 6: Connect the MH-Z19 CO_2 sensor to the Raspberry Pi Pico W.

In this configuration, you define a UART bus on pins GP4 and GP5, operating at 9600 baud. Note that the RX defined here is connected to the TX of the sensor, and vice versa. The configuration also defines the CO_2 sensor, which measures both $[CO_2]$ and temperature and sets the update interval to once per minute. You can remove the switch for the LED from the configuration because you don't need it here.

After installing this configuration, you'll see the current CO_2 concentration in parts per million (ppm) appearing in Home Assistant and in the logs on the ESPHome dashboard. Be sure to read the ESPHome documentation on calibrating the sensor to ensure accurate measurements. Note that the internal temperature sensor of the MH-Z19B isn't that accurate; it's primarily used as a reference for the CO_2 sensor.

Automated CO_2 Alarm

You can now read the CO_2 values in your Home Assistant dashboard (Figure 7), but you might not always be looking at your computer or phone screen. Fortunately, we already know how to control the built-in LED. In principle you can now add an automation in Home Assistant that turns on the LED on your Raspberry Pi Pico W when the CO_2 level is too high. However, this detour is not necessary; you can achieve the same result with ESPHome's built-in automation features. The advantage is that these automations continue to work even when your Home Assistant installation or MQTT broker (see the "MQTT Broker" box) is offline or your network connection is down.

To make your Raspberry Pi Pico W board act as a CO_2 alarm, turn on the

Listing 4: CO_2 Sensor over UART

```
uart:
  rx_pin: 5
  tx_pin: 4
  baud_rate: 9600

sensor:
  - platform: mhz19
    co2:
      name: "MH-Z19 CO2"
    temperature:
      name: "MH-Z19 Temperature"
    update_interval: 60s
```

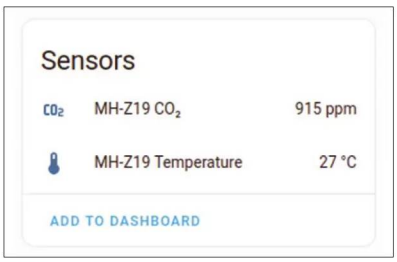


Figure 7: Your CO_2 sensor is visible in Home Assistant.

built-in LED when the CO_2 concentration exceeds 1,000ppm (Listing 5).

This code first defines the LED as an output, rather than a switch, so it's not controllable through Home Assistant or MQTT. The `id` lets you refer to the LED in the configuration of the CO_2 sensor.

Next, it modifies the `co2` section in the CO_2 sensor configuration, which configures the sensor to turn on the LED with ID `co2_alarm` when the CO_2 value exceeds 1,000ppm and turns it off when the CO_2 value drops below 1,000ppm. Your CO_2 alarm will now function even without a network connection. When you see the built-in LED turn on, you'll know that it is time to open the windows for ventilation.

Listing 5: CO_2 Sensor with LED Alarm

```
output:
  - platform: gpio
    pin: 32
    id: co2_alarm

uart:
  rx_pin: 5
  tx_pin: 4
  baud_rate: 9600

sensor:
  - platform: mhz19
    co2:
      name: "MH-Z19 CO2"
      id: co2_value
      on_value_range:
        - above: 1000
          then:
            - output.turn_on: co2_alarm
        - below: 1000
          then:
            - output.turn_off: co2_alarm
    temperature:
      name: "MH-Z19 Temperature"
    update_interval: 60s
```

Adding a Display

If you want more than just an LED on your device and prefer to see the full sensor value, you can add a display. ESPHome supports various display components [16], and this example uses an SSD1306 [17]. Start by disconnecting power from your Raspberry Pi Pico W and then connecting the display to your breadboard. Connect it as follows: the VCC of the display to 3V3 Out of the Pico W, GND to GND, SDA to GP8, and SCL to GP9 (Figure 8). Listing 6 shows the code you need to add to Listing 5 to show the sensor values on the display.

The SSD1306 display uses the I2C bus (there's also an SPI version), so first define this bus in the ESPHome configuration. If you use other pin numbers on the Raspberry Pi Pico W, make sure to use pins that are designated I2C0 in the pinout, because I2C1 isn't supported yet by ESPHome.

Because you want to show letters and numbers on the display, you also need to define a font. This example uses Roboto font size 18. The display is now defined and uses the previously defined I2C bus, specifying the model and I2C address of the display. The last line is the first real code in this article.

MQTT Broker

If you don't use Home Assistant – or if you prefer not to use the Home Assistant API – ESPHome also supports communication through the MQTT machine-to-machine messaging protocol [10]. This route requires you to set up an MQTT broker, such as Eclipse Mosquitto [11], running in a Docker container or as a Home Assistant add-on. Then you add an MQTT Client Component [12] to your ESPHome device configuration, specifying which MQTT broker to connect to, as well as the username and password for authentication. At this point, you can control the switches defined in your device by sending MQTT messages to your broker, and you can subscribe to MQTT messages from the sensors defined in your device.

This `lambda` is C++ code integrated in your ESPHome configuration. Although most ESPHome configurations can be defined without programming, displays are one of the few components that require code.

In this example, the one-liner calls the `printf` method on the display, specifying the horizontal and vertical coordinates in which to place the text, the `id` of the font, the string template to display, and the state you want to display. The value is obtained by referring

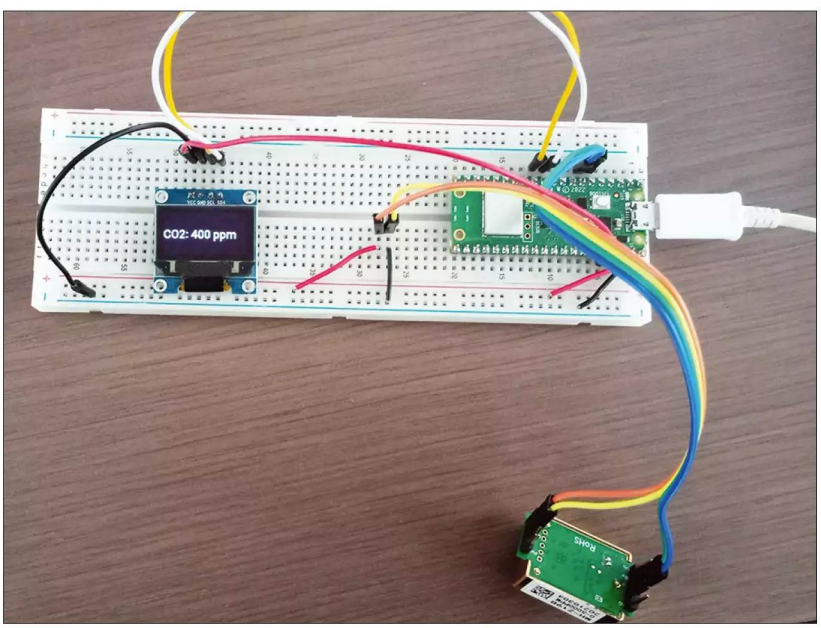


Figure 8: On a real breadboard, a circuit always looks messier than in a diagram.

to the ID of the CO_2 sensor and getting its `state` property:

```
id(co2_value).state
```

Because this is a floating-point number (e.g., 1746.00), the `%.0f` pattern shows only the integer part (i.e., 1746).

Complex Devices

The full YAML file of this CO_2 sensor device isn't very long, but you can create much more complex configurations. For example, I created an ESPHome air quality monitor [18] that combines a CO_2 sensor with a particulate matter (PM) sensor, temperature-humidity-pressure sensor, and display. I've also created an ESPHome configuration for the M5Stack PM2.5 air quality kit [19], as well as an ESPHome heart rate display [20] showing heart rate from a Bluetooth Low Energy (BLE) heart rate sensor on a display.

Finally, if you want to learn more about ESPHome and explore various examples to create your own home automation devices, read my book on the topic [21]. ■■■

Author

Koen Vervloesem has been writing about Linux and open source, computer security, privacy, programming, artificial intelligence, and the Internet of Things for more than 20 years. You can find more on his website at *koen.vervloesem.eu*.

Listing 6: Display Showing [CO_2]

```
i2c:
  sda: 8
  scl: 9

font:
  - file: "gfonts://Roboto@medium"
    id: font_roboto
    size: 18

display:
  - platform: ssd1306_i2c
    model: "SSD1306 128x64"
    address: 0x3C
    lambda: |-
      it.printf(0, 23, id(font_roboto), "CO2: %.0f ppm", id(co2_value).state);
```

Info

[1] ESPHome: *https://esphome.io*

[2] Arduino: *https://www.arduino.cc*

[3] MicroPython: *https://micropython.org*

[4] Shelly: *https://shelly.cloud*

[5] Sonoff: *https://sonoff.tech*

[6] ESPHome devices: *https://devices.esphome.io*

[7] ESPHome Docker image: *https://esphome.io/guides/getting_started_command_line.html#installation*

[8] Home Assistant: *https://home-assistant.io*

[9] YAML: *https://yaml.org*

[10] MQTT: *https://mqtt.org*

[11] Eclipse Mosquitto: *https://mosquitto.org*

[12] MQTT Client Component: *https://esphome.io/components/mqtt.html*

[13] ESPHome sensor components: *https://esphome.io/index.html#sensor-components*

[14] ESPHome MH-Z19 sensor: *https://esphome.io/components/sensor/mhz19.html*

[15] Raspberry Pi Pico W pinout: *https://picow.pinout.xyz*

[16] ESPHome display components: *https://esphome.io/index.html#display-components*

[17] ESPHome SSD1306 display: *https://esphome.io/components/display/ssd1306.html*

[18] ESPHome air quality monitor: *https://github.com/koenvervloesem/ESPHome-Air-Quality-Monitor*

[19] Config for M5Stack PM2.5 air quality kit: *https://github.com/koenvervloesem/M5Stack-Air-Quality-ESPHome*

[20] ESPHome heart rate display: *https://github.com/koenvervloesem/ESPHome-Heart-Rate-Display*

[21] Vervloesem, Koen. *Getting Started with ESPHome*. Elektor International Media B.V., 2021, *https://www.elektor.com/getting-started-with-esphome*

■■■

Manage your greenhouse with a Raspberry Pi Pico W

Sheltered Growth

You can safely assign some greenhouse tasks to a Raspberry Pi Pico W, such as controlling ventilation, automating a heater, and opening and closing windows. *By Swen Hopfe*

When implementing my greenhouse control system, I didn't have to start completely from scratch. An older control system already existed with which I had a little experience. Building on this established setup, I decided to use power windows for the hinged skylights (Figure 1) and a fan to circulate the air in the greenhouse. Also, when nighttime temperatures dropped in the spring and fall, I wanted a heater to switch on automatically. In contrast, crops needed protection against excessive heat in summer.

An intelligent control system would also be nice to reference the outside temperature, allowing it to close the windows in time for cool evenings and build up a heat reserve for young crops during the night. At the same time, a reliable clock was essential to adapt to the lighting conditions of different seasons.

All functions should be remotely accessible, with the option to intervene over the web if thresholds were exceeded. Another requirement was an activity log to collect messages from ongoing operations for remote viewing without always having to check the display in the greenhouse. To implement all of this, I used a Raspberry Pi Pico W. In addition to the essential peripheral devices, it now provides the entire logic and a web server.

Figure 1: The automatic greenhouse control system regulates when the skylights open and close.

Getting Started

Unlike the single-board computers from the Raspberry Pi family, the Pico requires very little preparation. I used the WiFi version because the controller could not be managed remotely without a connection to the home WiFi network. I also needed a USB port for the programming. In the development phase, you need to feed the commands externally from the special Python Thonny integrated development environment (IDE)

Lead Image © Ledda Maria Rita, 123RF.com

Parts List

- Raspberry Pi Pico W
- LCD panel with I2C controller
- DS1307 real-time clock with AT24C32N EEPROM
- DS18B20 temperature sensor
- 12V electric window lifters (2)
- 6x relay board
- LED board
- Transistors, resistors
- 5V and 12V supply (for top-hat rail)
- Automatic circuit breaker (for top-hat rail)
- Various empty housings (for DIN rail)
- Housing (fuse box)
- Wiring, installation material

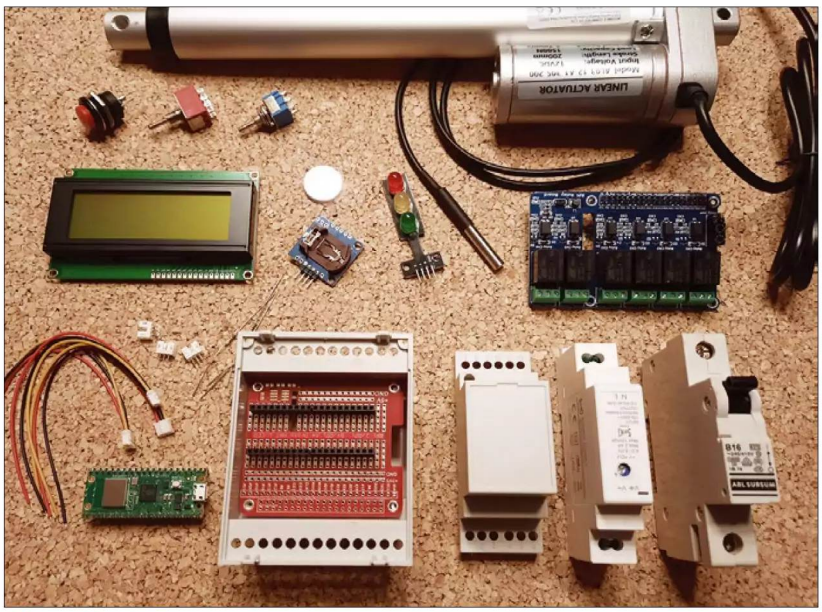


Figure 2: The components in this setup are connected to the Pico board by plug connectors.

to the controller, and to finish the job, you need to transfer your code to the module permanently.

One advantage of microcontrollers over computers with filesystems is their robustness, with no risk of the SD card or hard drive being damaged by a power failure. You do not need to shut down your controller in a defined way beforehand when switching it off and can instead simply press the power switch. In return, you have to make do with fewer resources and without a battery-buffered real-time clock in the case of the Pico. Luckily, this restriction did not matter for my control system, because I designed in a real-time clock (RTC) to exchange information with a time server on the Internet. The project also had electrically erasable programmable read-only memory (EEPROM) to store the latest setting values.

Setup

The LCD display and the RTC are connected to Pico over an I2C bus and the temperature sensors over a 1-Wire bus. The transistors are connected upstream of the LEDs that occupy three ports. To allow the Pico to control the large consumers such as the window lifters and heater (230V), a relay board with six inputs was also controlled by the Raspberry Pi GPIOs. In this use case, the Pico sits in a housing and on its own small circuit board. All other components (see the "Parts List" box) were connected by plug connectors and put in a DIN top-hat rail housing (Figure 2). The idea was to keep things manageable for any service work and to be able to disconnect all the components easily.

A 5V power supply powers the entire circuitry, and 12V is required for the window motors. In case of activity, the current there needs to be relatively high. To avoid the power supply being constantly idle, and to remove the need for DC/DC converters, the 12V motor power supply is only switched on with a relay when needed. Most of the time, I make do with a frugal 5V circuit to save money.

Most of the hardware is housed in a prefabricated enclosure – a fuse box with a weatherproof seal (Figure 3). I wanted to take advantage of a DIN top-hat rail housing to be able to arrange and exchange individual modules easily side by side. Whatever didn't fit beside the circuit breaker and power supply units was located in empty housings for the top-hat rail (e.g., LCD display, switches, and LED). Enough space was left in their housings to install the Pi, the relay board, and the rest of the electronics closer to the rear, resulting in an uncluttered front side.

I then routed the wires out to the lift motors, fan, heater, door contact,

Figure 3: All components are mounted on DIN top-hat rails in a weatherproof housing.

temperature sensors, and network connection through the enclosure, making sure that everything was watertight. All of the outdoor wires first were routed to clamp connectors on the inside so the control and switch enclosure could be tested separately from the rest of the external installation and be easily assembled.

Control

If you take a look at the few alternatives available for purchase, you'll quickly discover that various controllers are often limited to thermostats only and are difficult to expand. However, I wanted to use my own accessories and be able to program everything.

To operate the solution, I use a central control script `main.py` in the usual style for the Pico and added methods for the LCD controller and DS1307 chip to the source code.

Development occurred in the Python Thonny IDE (Figure 4), which integrates the Pico seamlessly. MicroPython was the programming language of choice. If you have a device without MicroPython, you first need to take care of loading the latest firmware. Numerous workable how-tos for this step can be found on the Internet.

After applying the operating voltage, the Raspberry Pi initially checks to see whether all of the components are in place and whether all of the peripherals

Figure 4: The Python Thonny IDE was used to develop the code for the controller.

can be accessed by I2C or the 1-Wire bus. If this is the case, the system can be fired up. In the case of less serious events, such as a missing Internet connection or an inaccessible time server, the system continues and tries to establish contact at a later time. After that, the Pico lowers the windows to the stop position to adjust the zero points in case the windows were open. The window lifters have internal stop switches, so I didn't have to connect anything to the Pi.

The current operating values are queried in an infinite loop and checked for the designated upper and lower limits for switching the actuators, followed by appropriate actions. The system is really smart in this way because a variety of indoor and outdoor conditions play a role. For example, knowing whether the access door is open or closed is important for ventilation, and different switching values for everything depend on the time of year because climate parameters such as insolation will vary. The controller also provides different hystereses depending on the time of day, to avoid repeated switching. In the evening, for example, the windows are kept closed longer to store some heat for the night.

The script is also responsible for displaying the current values on the LCD display of the control panel on the outer wall and for writing logs, which you view in the web app. Scheduled actions are reported in the message log, as well as exceptions into a separate error log, such as a sensor failure or exceeding maximum or minimum temperature limits, because errors and warnings are retained for a longer period of time. At the end of the day, then, you can find out what has been going on with a quick check of the app or the LCD display.

The corresponding switching values are defined as constants in the source code by default. If required, the constants can be customized later in the EEPROM, which you can also write to from the web app. For the app to work at all, the script keeps trying to reconnect at regular intervals after a connection loss. The exact time is also resynchronized regularly. In general, the Raspberry Pi's MicroPython interpreter should never stop, if possible; you want it to keep things under control whatever else happens.

I worked with the *uasyncio* library to allow ongoing operations and web interface updates to take place in parallel. The library provides an asynchronous scheduler that assigns application time to both tasks, which makes it a great choice for this use case: to ensure smooth operations (short response time after pressing a button in the web interface) on the one hand and smooth processing of the program (avoiding long waits for requests from the web server) on the other.

Of course, you also want to be able to control all of the functions manually as an alternative. For this to happen, a multipole switch disconnects the actuators from the control system so that each of the two windows can then be set to the desired position without conflicting with the automatic system, and the fan and heater can be set. The LCD display backlighting also is manually switched on to make it easier to read in the evenings.

Web-Based Remote Control

The control system works reliably offline, but in my opinion, remote web-app-based access (Figure 5) is a great idea because it avoids the need to check everything manually onsite. Also, it means you do not have to make the control panel on the greenhouse too fancy. In support, the Pico can run a web server, which then gives you a user interface (Figure 5). If so desired, you can additionally share the interface on the Internet, but in this case, it can only be accessed on the local network.

The app's functions are:

- Immediate display of the current values
- Minimum/maximum temperatures and assessment of the operating status
- Manual switching of windows, fan, and heater
- Resetting variables to automatic control
- Displaying and deleting messages and error logs
- Daily and monthly charts of indoor/outdoor temperatures
- Reading and writing control parameters from and to memory

Because the Pico is always within range of my home WiFi network, I have the situation in the greenhouse constantly under control and can intervene by smartphone, tablet, or PC from the garden or apartment.

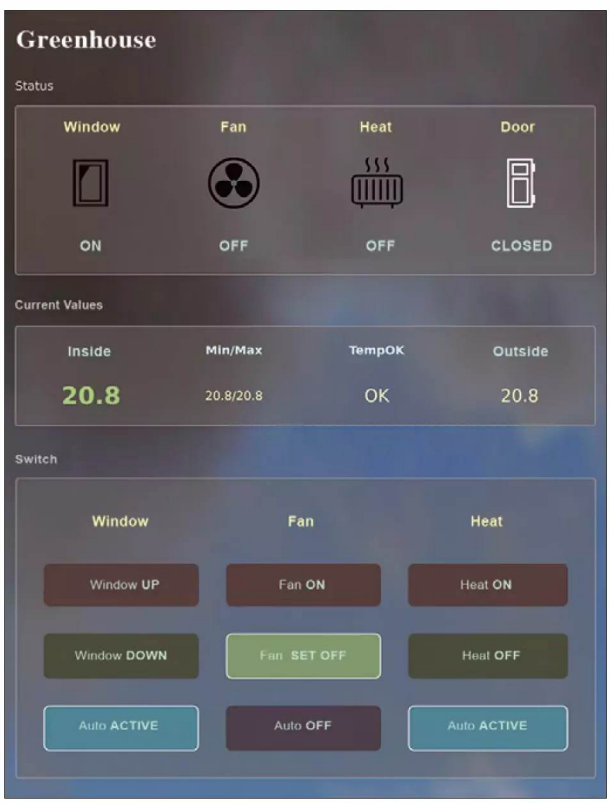


Figure 5: Thanks to a user interface provided by the web server, it's fine for the control panel to be a bit spartan.

Conclusions

The greenhouse is in constant use from April to October. Accordingly, I want the electronic control system to be not only functional, but above all reliable in terms of operation. To protect the crops, it is advisable in a project like this to test everything thoroughly, module by module, up front before putting anything into operation. Once you have installed the system outside, it can be difficult to access the individual components.

The new controller has been running for some time now and has demonstrably been a valuable asset in the greenhouse thus far (Figure 6). It is reassuring to know that everything is well taken care of in my absence. Sometimes it's the little things that matter, like the LED lights that let you know whether everything is OK as you walk past the greenhouse. In the next stage of expansion, I want to add moisture sensors so that I can also monitor the soil. You will find the software and full details of the project online [1], as well as on my GitHub site [2]. ■■■

Figure 6: The fully stocked greenhouse from the inside.

Info

[1] Greenhouse project (in English): *https://linuxnewmedia.thegood.cloud/s/XnzsiEKtagjHKr3*

[2] Greenhouse project (original code in German): *https://github.com/swenae/ghouse*

Author

Swen Hopfe works for a medium-sized company with a focus on smart cards and near-field communication (NFC). When he is not taking photos, in the great outdoors or in his garden, he focuses on topics such as the Raspberry Pi, Internet of Things, and home automation.

Control your smart home with RaspBee II and your Raspberry Pi

Home Manager

The RaspBee II module turns your Raspberry Pi into a smart control center for Zigbee devices.

By Erik Bärwaldt

New technologies and new providers are constantly expanding the range of potential smart home applications. Solutions from various manufacturers are often mutually incompatible, which means customers cannot leverage the full potential of home automation services.

But there is another approach. Zigbee is a protocol for low-power wireless communication based on the IEEE 802.15.4 standard. The Zigbee protocol is often used for home automation devices. Dresden elektronik [1] has been working on light control systems based on the Zigbee protocol for some 10 years and now offers numerous vendor-agnostic systems for the smart home. The company also distributes the RaspBee module for home automation with a Raspberry Pi. A RaspBee device with the accompanying software can help you reap the benefits of the smart home without expensive gateways and cloud connections.

The RaspBee module, which has already reached its second generation, is a piece of hardware attached on top of a Raspberry Pi (called a HAT) that helps users convert the system into a control center for smart home devices such as lights, alarms, or smart sockets and switches. The RaspBee II module [2] supports devices from various manufacturers with home automation solutions based on the Zigbee protocol. The underlying technology is a framework for wireless networks that are especially suitable for home automation due to their low data volume and low energy requirements.

The modular Zigbee protocol supports extensions, and some manufacturers try to boost customer loyalty by adding new functions that lead to incompatibility with devices by other providers, thus preventing users from switching. The Zigbee II module takes this into account by having the developers test components from a wide range of vendors and add them to continually updated compatibility lists [3].

The latest RaspBee module is far more compact compared with the first-generation model (Figure 1), and it also comes with an important innovation. Thanks to a battery-buffered RTC on the HAT, the system is now able to sync with the Raspberry Pi control center after a power failure to carry out time-critical tasks. According to the manufacturer, the battery should last for at least two years and will provide power for up to eight years if the system is used daily. It is a replaceable button cell.

In addition, the RaspBee II module comes with a power amplifier that gives users an effective range of 30 meters inside buildings. Outdoors, a range of up to 200 meters should be possible. Because Zigbee installations act as a mesh network, where devices such as lights or smart sockets play the role of repeaters, you can easily extend the range without additional hardware.

RaspBee HATs are supported by two software packages. You can use the deCONZ platform to configure the hardware. The graphical tool visualizes Zigbee networks, to which you can add devices from different vendors. deCONZ runs in the background during this process. The

Lead Image © Pavel Shlykov, 123RF.com

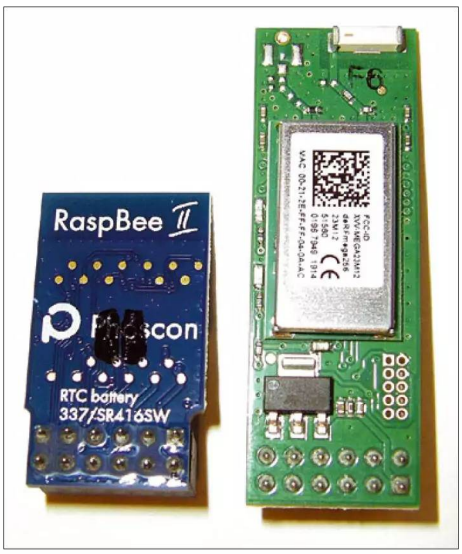


Figure 1: The Zigbee II module (left) has shrunk considerably compared with its predecessor (right).

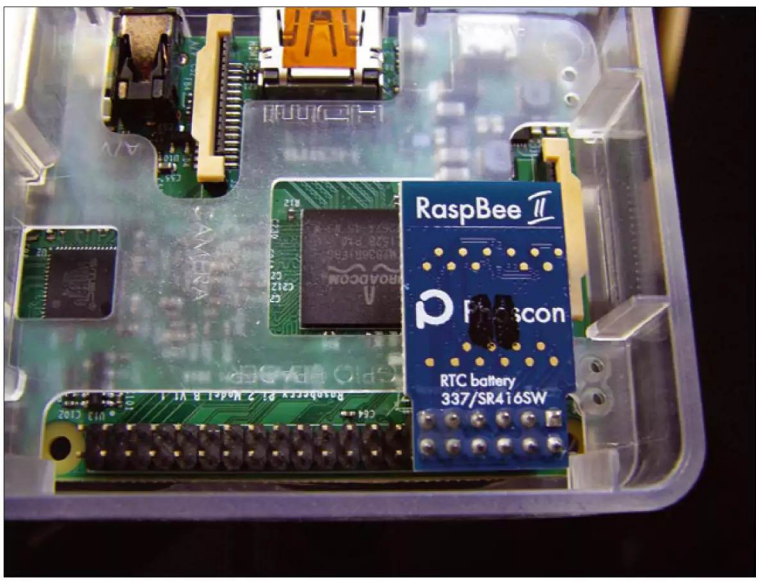


Figure 2: The Zigbee II module is plugged into the end of the Raspberry Pi's pin rail.

second component is the Phoscon app, which is also graphical and browser-based. Phoscon acts as a graphical front end for controlling lighting installations [4].

Installation

Get started by simply plugging the RaspBee II module onto the pin header of the Raspberry Pi – on the side facing the slot for the MicroSD card (Figure 2). The module supports all versions of the Raspberry Pi, so you can use an older model for home automation. You need a current version of Raspbian "Buster" or Pi OS "Bullseye" as the operating system.

The Raspbee manufacturer offers a total of four different images for MicroSD cards. These images contain a customized Raspbian "Buster" on which the deCONZ application is preinstalled. One of the images already has the Homebridge Hue plug-in, which you can use to control components of the Philips Hue lighting system. The images all have the Raspbian working environment, except for one, so you can conveniently run the configuration graphically. All images are also suitable for the first generation RaspBee module.

The images with the desktop environment each weigh in at just shy of 880MB, while the headless image without the graphical environment only takes up 395MB. The provider recommends using a fast MicroSD card with a minimum capacity of 8GB for all images. It should at least meet the Class 10 standard to avoid latencies during operation [5]. Alternatively, check out the vendor's website, where you will find detailed instructions for integrating the software packages required for the RaspBee II module with conventional Pi-OS images [6]. If you decide to use the preconfigured images, a system update and the deCONZ configuration software will launch automatically. In the graphical network representation in the deCONZ control interface, you will initially only see the existing gateway (Figure 3).

Then, working on any machine on the local network, open a web browser and type *http://RaspPi_IP_address:80* as the target address. After a short startup delay, the browser window displays the gateway. Clicking on the gateway icon takes you to a configuration dialog where you can set a password for logging into the gateway.

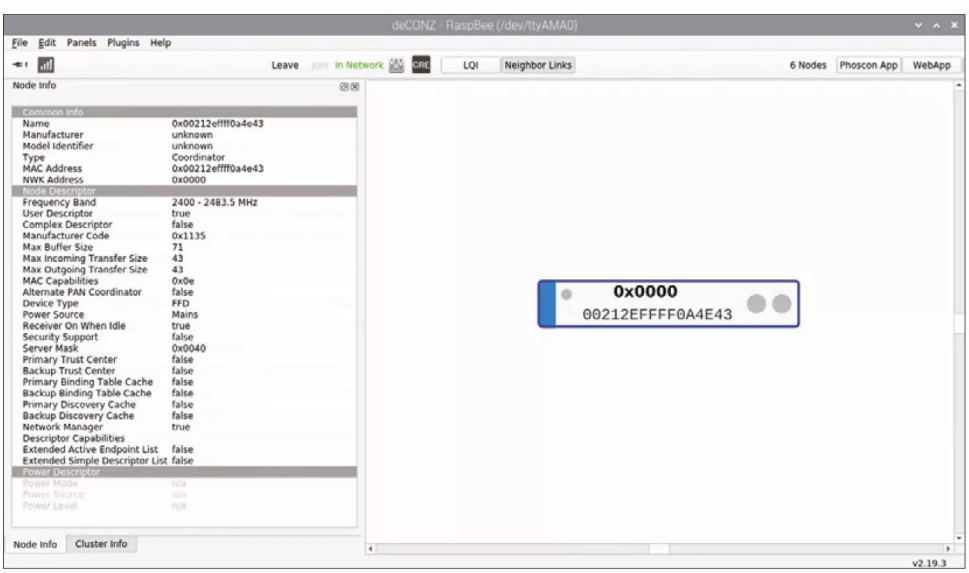


Figure 3: The main section of deCONZ shows you the gateway.

After that, future settings require logging into the gateway (Figure 4).

The routine prompts you to turn on all the lights you want to integrate with the Zigbee network. But before doing so, you have to reset the lights to the factory defaults, otherwise the RaspBee module will be unable to identify the components. Dresden elektronik supplies instructions for resetting lighting by various home automation manufacturers.

After you switch on the lights, deCONZ searches for them and lists them. The software then automatically draws connecting lines from the gateway to each device to outline the mesh network (Figure 5). The identified components appear in a table in the browser window of the Phoscon app. Click on one of the devices in deCONZ to display its technical data in a vertical pane on the left.

Because deCONZ is vendor-independent, you can add Zigbee-compatible devices from different manufacturers to your smart home setup. But before purchasing individual components, it makes sense to take a look at the compatibility list provided by the vendor to make sure your choice of device actually works with the RaspBee II system.

Grouping

Once all devices are logged onto the gateway, click *Mainpage* in the Phoscon app browser window. Then proceed to create an initial group in an overlapping dialog box. Groups are typically used to designate different rooms with smart home components in the Phoscon app.

After clicking *Create first group*, go on to define the group name in a separate dialog box. The app makes suggestions for a group name. After you *Create* the new group, you are taken to the dialog for assigning the identified components. If you click on the gear icon in the bottom right, you will come across the *Manage lights* option. In the *Available lights* dialog that appears, you need to add the desired components to the group.

Be careful: Some components, such as intermediate switches that integrate a conventional device into the smart home, are also identified by the software as lights and listed accordingly. Luckily, meaningful symbols to the left of the individual nodes reveal whether the device is a light source or some other device. Use the plus sign to the right of each

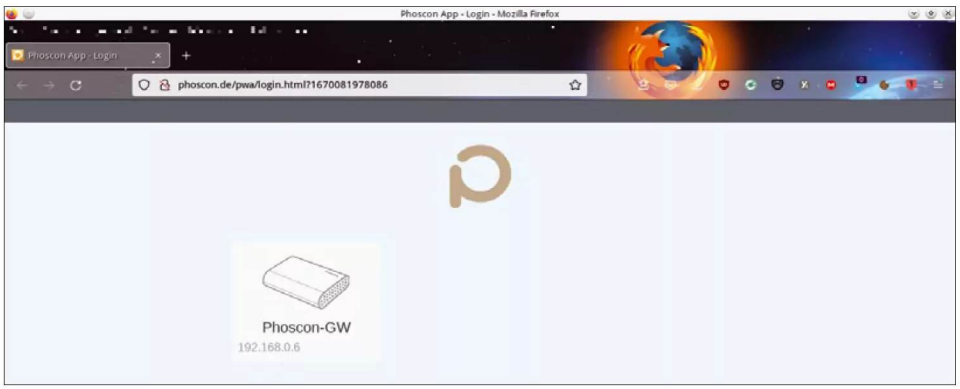


Figure 4: When you launch the Phoscon app, you initially see only the gateway.

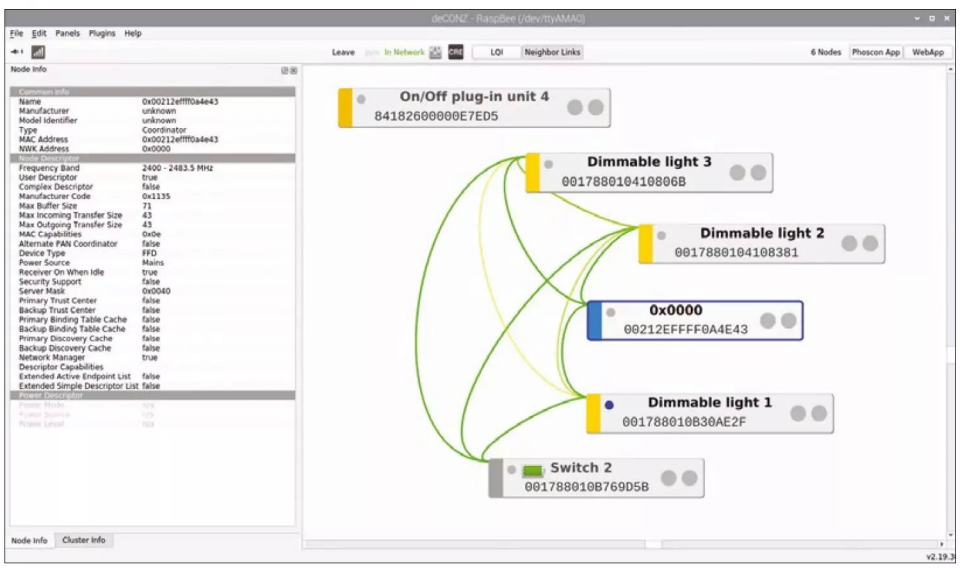


Figure 5: deCONZ draws connecting lines to outline the mesh network.

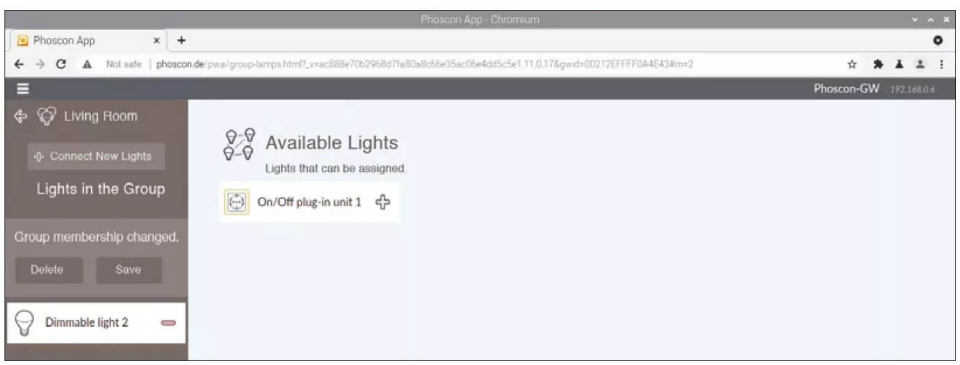


Figure 6: You can see the group members and add new ones in a graphical interface.

device to transfer it to the vertical group column on the left. Don't forget to *Save* your work after adding all the desired components to the group (Figure 6).

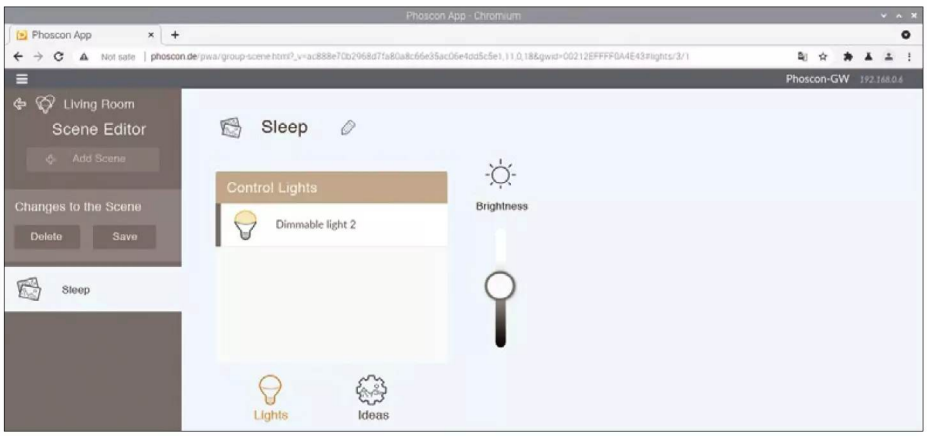


Figure 7: The scene editor lets you control the devices individually.

Making a Scene

In the next step, you add what are known as scenes to the installation and then configure the individual nodes in each scene. For example, you can turn switches or lamps on and off using timer controls or dim the lights in the scene editor.

The *Scenes editor* option lets you open the scene dialog, where you need to click on *Add Scene* and assign a meaningful name. After clicking *Create*, you can configure and save the desired settings as a function of the options supported by the enabled device (Figure 7).

If you now switch back to the *Groups* menu, you can use the *Schedules* dialog to define when you want the tool to enable a specific scene in the current group. *Schedules* | *Actions* lets you set an alarm or a timer, to which you can assign a name.

Following the *Create* step, you will find yourself in the configuration dialog for timer control. Just a few mouse clicks let you define which scene is linked to which action and at what time. You can also specify the length of the fade time before starting the timer (Figure 8).

You need to repeat this procedure for each room in which you have Zigbee network components. When you are done, you can enjoy reliable automated control of your home devices.

Conclusions

The RaspBee II module makes it easy to manage devices supporting the Zigbee protocol. The Raspbee II's control software is very flexible – you can even use it on remote machines via VNC, and the accompanying software supports most of the Zigbee devices on the market. You can even integrate older systems without any manual configuration, provided the devices are not burdened with incompatibilities caused by proprietary extensions. ■■■

Info

[1] dresden elektronik: *https://www.dresden-elektronik.com*

[2] RaspBee II module: *https://phoscon.de/en/raspbee2*

[3] Compatibility lists: *https://phoscon.de/en/conbee2/compatible*

[4] Software descriptions: *https://phoscon.de/en/raspbee2/software*

[5] Downloads: *https://phoscon.de/en/raspbee2/sdcard*

[6] How-tos: *https://phoscon.de/en/raspbee2/install*

Author

Erik Bärwaldt has been working as an IT consultant for several decades and contributes to many IT magazines and websites.

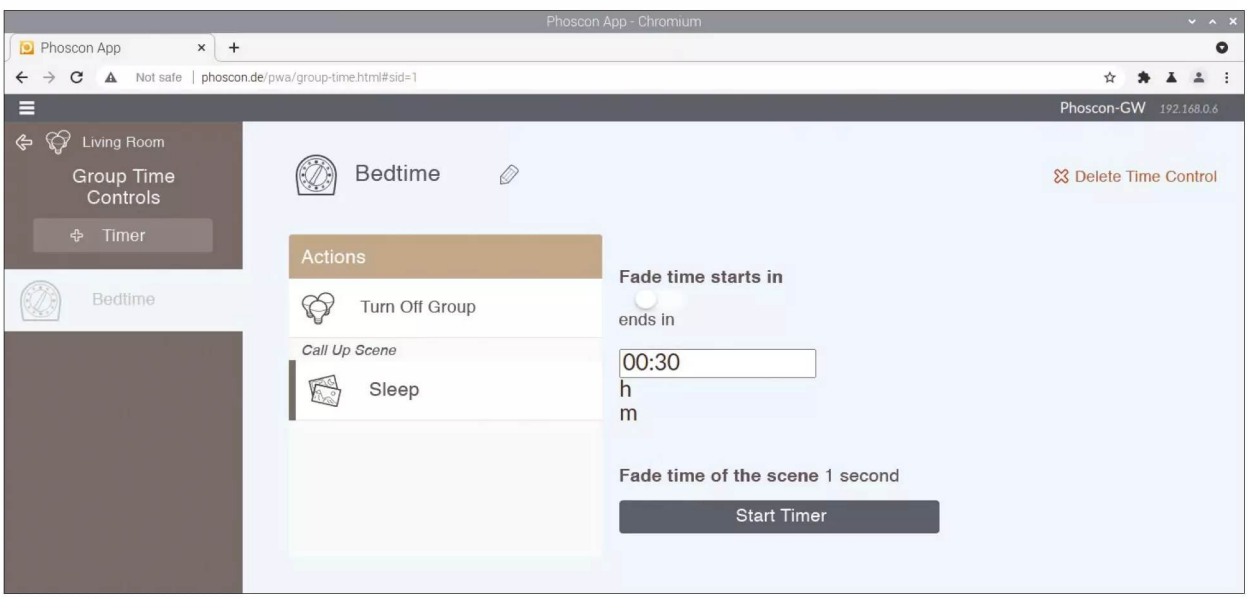


Figure 8: The timer lets you automate the control of individual devices.

Home Assistant controls microcontrollers over MQTT

Home Sweet Home

Automating your four walls does not necessarily require commercial solutions. With a little skill, you can develop your own projects on a low budget. *By Gerhard Schauer*

Home automation offers many opportunities, but equally harbors many risks. If you succeed in becoming independent of commercial products, you can save money while retaining control over what data flows where. In this case, the Message Queuing Telemetry Transport (MQTT) protocol proves to be very useful.

A previous article on Z-Wave [1] showed how you can bring the Raspberry Pi up to speed with Home Assistant and components available on the market to attain the goal of achieving automation magic in your home without human interference. Plenty of components can be addressed by Home Assistant, even without the cloud. The only limits are your wallet and your imagination.

This sequel explains how you tweak both the price and the DIY factors of the components. The Raspberry Pi from model 3 onward comes with a wireless interface that is also available in many microcontroller modules and is likely to open many doors. The glue that connects the whole thing to the Home Assistant environment looked at in the previous article is an IP-capable protocol that saw the light of day long before any Internet of Things (IoT) hype did: MQTT.

The MQTT protocol was introduced in 1999 and is a text-based protocol that runs over TCP on any IP network. Transport Layer Security (TLS) can optionally be used for encryption. The sensors and actuators communicate as clients with the broker, which acts as the communications hub. Addressing relies on what are known as "topics." Transmitters are referred to as publishers, and receivers as subscribers.

Testing MQTT

To implement the protocol on the Raspberry Pi, you first need the *mosquitto* package. For the small implementation tests that follow, you will also need the matching client. You can set up both components on the Raspberry Pi with the commands:

```
$ sudo apt-get install mosquitto
$ apt-get mosquitto-clients
```

If the broker is running after the install, you can log in to the individual topics with the client and retrieve a list of topics with a command-line call that specifies the host on which the broker is running with the `-h` option:

```
$ mosquitto_sub -h localhost -t "#"
```

In this case, it is localhost. For example, to send the value *10* to a topic

Lead Image © shamain, 123RF.com

named *test/test1* and receive the value as a subscriber, you would use the commands:

```
$ mosquitto_pub -V mqttv311 ➥
  -h 192.168.3.7 -t test/test1 -m 10
$ mosquitto_sub -h 192.168.3.7 ➥
  -t test/test1
```

For a more practical example, I'll use MQTT to transfer the measured values of a one-wire temperature sensor of the DS18x20 type connected to a Raspberry Pi to the broker listening on another Raspberry Pi. To do this, you need to enable one-wire support up front with

```
dtoverlay=w1-gpio
```

in the `/boot/config.txt` file.

After connecting the sensor to the standard one-wire GPIO port, the temperature values can be queried as four-digit integers in `/sys/bus/w1/devices/28-0417c1b1f7ff/w1_slave`. The value `28-0417c1b1f7ff` is the unique ID of the sensor on the bus. The complete command sequence is:

```
$ mosquitto_pub -h 192.168.3.7 ➥
  -t wohn/temp/28-0417c1b1f7ff ➥
  -m $value
$ mosquitto_sub -h localhost ➥
  -t wohn/temp/28-0417c1b1f7ff
```

To transfer the values to the Home Assistant environment, you need to enter the broker you are using in the `configuration.yaml` file; in this case, it is running on the same host (Listing 1, lines 1-2). The entries in the lines that follow take care of subscribing to the appropriate topic and fielding the sensor data in Home Assistant; the last line also specifies degrees Celsius as the unit.

ESP8266 Controller

The theoretical side is now covered, but to use a separate Raspberry Pi for each remote actuator or sensor seems a little unrealistic in practice. The ESP8266 controller will come in handy here. On the one hand, you can source low-priced and useful developer boards that use this controller; on the other hand, the chip is used in various products from the Far East, and you can take control of many of these simply by flashing them with freely available firmware.

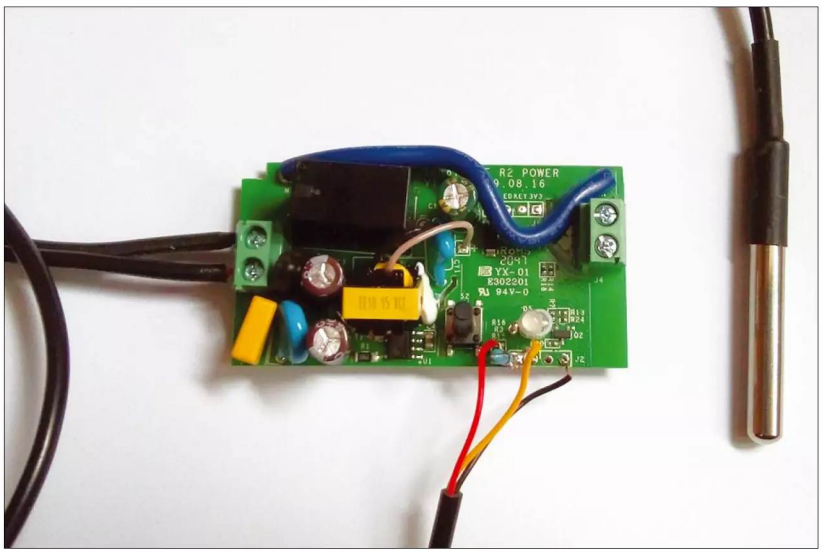


Figure 1: The Sonoff Basic WiFi smart switch with temperature sensor.

For the tests here, I used a Sonoff Basic WiFi smart switch [2] (Figure 1). This controllable relay switch has a screw connection for a 220V mains supply with switchable output.

CAUTION: There is a danger to life if this hardware is not used correctly. In this article, I am restricting the supply voltage to the module to 3.3V.

The firmware is Tasmota. The software lets you configure a variety of supported sensors and components on the existing GPIO ports of the controller, such as the temperature sensor I referred to earlier. Tasmota uses MQTT to transmit the data from the sensors.

Listing 1: configuration.yaml

```
01 mqtt:
02   broker: localhost
03
04 sensor:
05   - platform: mqtt
06     name: temp1
07     state_topic: "living/temp/28-0417c1b1f7ff"
08     unit_of_measurement: '°C'
```

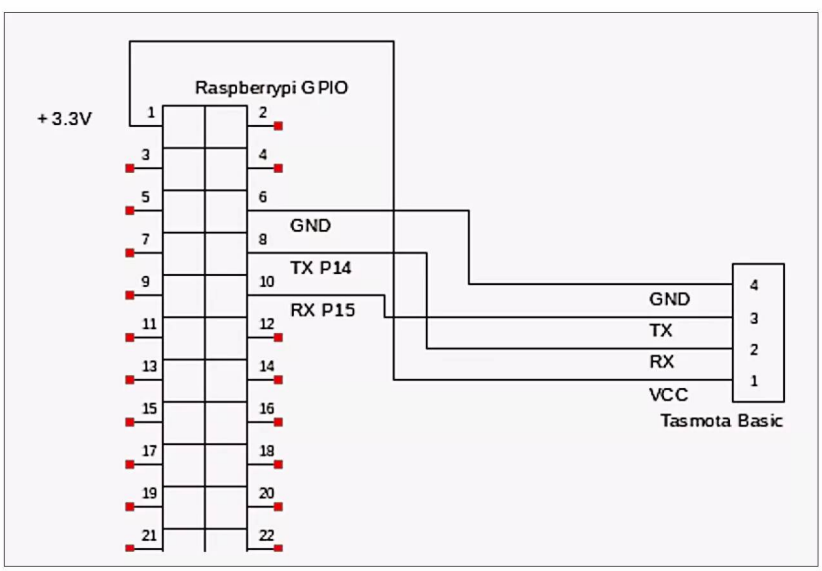


Figure 2: Wiring the Sonoff Basic to the Raspberry Pi.

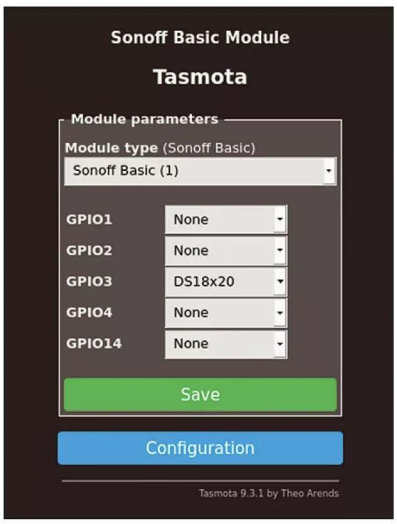


Figure 3: A convenient web GUI helps you set up the temperature sensor.

Flashing Tasmota

Among the several ways to flash Tasmota, one popular approach is to use a USB serial adapter. A description can be found on the Getting Started page of the project [3]. Because the Raspberry Pi comes with a serial interface, I will do the flashing directly through the GPIO.

Figure 4: Just a few parameters let you prepare Tasmota for transferring data to Home Assistant.

The wiring required for this is shown in Figures 1 and 2. The `esptool.py` utility from the PiOS repository provides the software.

The Tasmota firmware, along with a massive collection of information, can be found in the project's GitHub repository [4]. I downloaded the sensor variant for the tests because I wanted to integrate the temperature sensor:

```
$ wget -c ↵
  http://ota.tasmota.com/tasmota/↵
  release-9.4.0/tasmota-sensors.bin
```

After connecting the module as shown, you have to hold down the button on the Sonoff module while powering it on; the LED does not light up. Next, use the command

```
$ esptool --port /dev/ttyAMA0 chip_id
```

to check whether you can access the Sonoff. An ID must be shown.

Finally, flash Tasmota with the command:

```
$ esptool ↵
  --port /dev/ttyAMA0 write_flash ↵
  -fm dout 0x0 tasmota-sensors.bin
```

This operation should take about a minute. When the module is restarted, it reports for duty as the new WiFi access point, which you can reach at *http://192.168.4.1* and on which you configure the WiFi network to be used. All further setup steps are then performed with the IP address assigned by your own WiFI network.

Setting Up Tasmota

Before proceeding, you first need to connect the temperature sensor to one of the GPIOs used for flashing (RX/TX), which is now free. You can do this with a pull-up resistor and the 3.3V and GND connections [5]. After wiring the elements, it's time to configure the sensor; its name will have already appeared in the GUI (Figure 3).

You also need to assign a password for the web GUI and activate MQTT (Figure 4), which you will be using to deliver the data to Home Assistant. The broker and topic are configured in the corresponding entries in the web interface, where you will also find the Tasmota console, which lets you issue commands directly and set up the parameterization.

At this point at the latest, I recommend studying the very good documentation from Tasmota's GitHub repository. For this project, I used a command that makes the later integration with Home Assistant by autodiscovery very easy by telling Tasmota to announce all available topics (`setoption19 on`). At the end of the day, Tasmota advertises itself in the GUI as a web switch with a temperature sensor (Figure 5). You can also

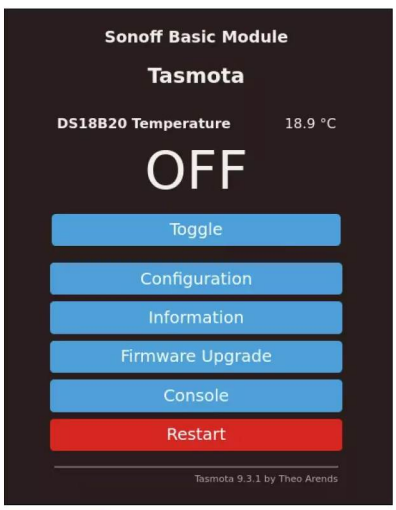


Figure 5: Hardly distinguishable from commercial products: the web-based user interface of Tasmota.

Figure 6: A few simple steps are all it takes to integrate the MQTT protocol into Home Assistant.

view the MQTT publishing in the Console, if needed.

Integration

To make the components available in Home Assistant, you now need to set up MQTT integration in the GUI. To do so, all you need is the IP address and port of the broker you are using; by default, this is set to 1883. After the setup, the new MQTT devices should be displayed (Figure 6).

After clicking on the *entities* URL, both the switch and the temperature sensor are shown as available Home Assistant entities, and you can now use them in the usual way (Figure 7). In combination with the relay in the Sonoff module, the familiar automation features of Home Assistant can now be used to implement, say, a simple thermostat control.

Conclusions

The Tasmota option associated with Home Assistant's autodiscovery of MQTT integration easily makes this DIY solution as convenient as the commercial counterparts described in the previous article. Homebrewing turns out to be worthwhile – and great fun, too. ■■■

Author

Gerhard Schauer is a self-employed electronics engineer living in the southern part of Germany. He writes maker articles because learning by doing and maintaining control of technology is the "right" way.

Info

[1] "Z-Wave Home Assistant" by Gerhard Schauer, *MakerSpace 03*, 2023, pg.53, *https://linuxnewmedia.thegood.cloud/s/XnzsiEKtagjHKr3*

[2] Sonoff Basic R2 Power: *https://tasmota.github.io/docs/devices/Sonoff-Basic/#sonoff-basic-r2*

[3] Getting Started: *https://tasmota.github.io/docs/Getting-Started/*

[4] Tasmota repository: *https://tasmota.github.io*

[5] Wiring Tasmota: *https://tasmota.github.io/docs/DS18x20/*

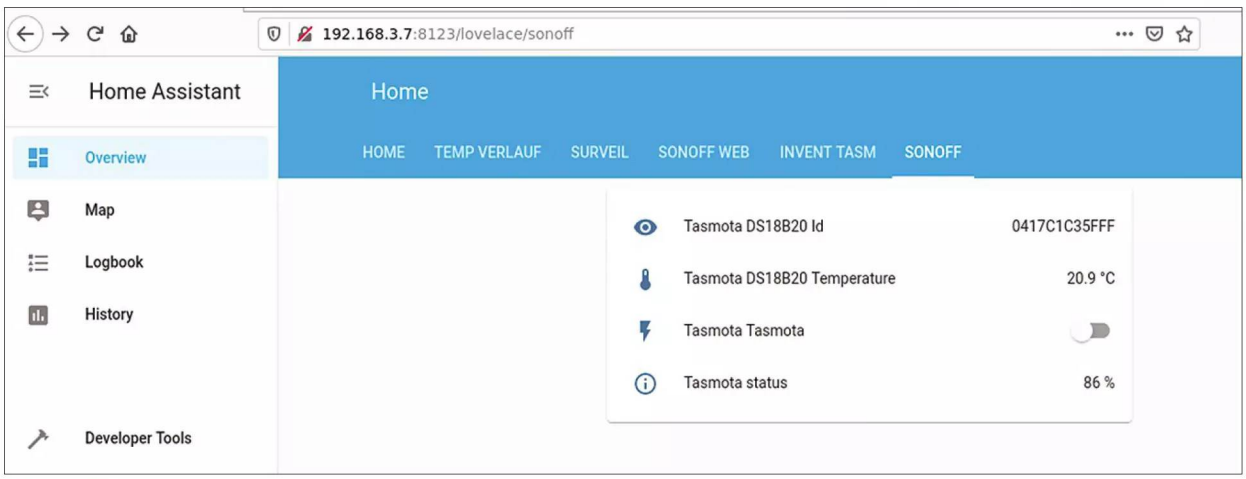
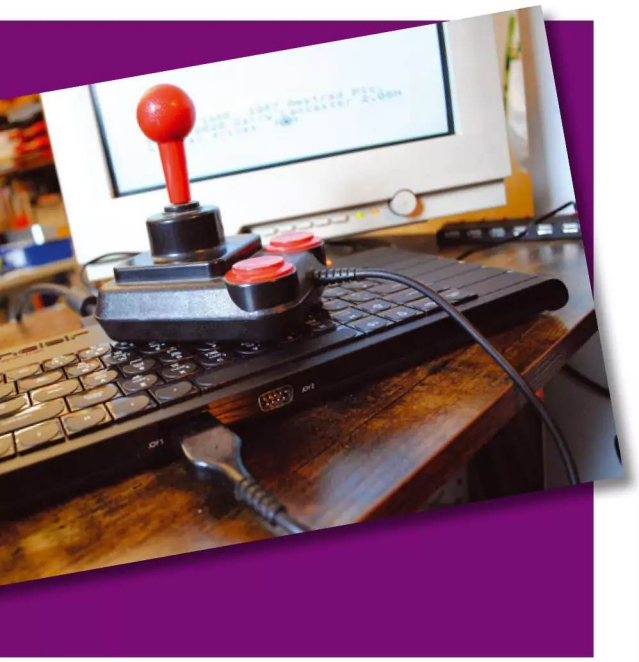


Figure 7: Home Assistant presents the integrated sensors in a clear-cut and neat way.



Second Edition of the FPGA-based Sinclair ZX Spectrum Next

The Next Spectrum

They've done it again: After the ZX Spectrum Next development team at SpecNext ended their second successful Kickstarter campaign in 2020, backers had to wait until Christmas 2023 when they were finally able to put a new 8-bit computer under the tree. Was it worth the wait?. *By Hans-Georg Eßer*

The Sinclair ZX Spectrum [1] was one of the great home computers of the 1980s: Its first models from 1982 had a rubber keyboard and only 16KB or 48KB of RAM. Driven by a Z80 CPU that ran at 3.5MHz, owners would typically connect a cassette recorder and load games from tapes. Over the years, Sinclair made some improvements: For example, the ZX Spectrum + had a better keyboard, and the culmination of the development process was the ZX Spectrum 128K with lots of RAM (also known as the "Toast Rack" model). After that, Amstrad bought Sinclair and released new Spectrum models with built-in cassette or disk drives, but those machines looked nothing like the old Speccies, and in 1992 the product line was discontinued.

That is, discontinued until 2016 when the Spectrum was revived by a team which included Rick Dickinson, the industrial designer who had in the past worked on several Sinclair machines. After some prototype work, SpecNext [2] started a Kickstarter campaign for a Spectrum successor that they named "Spectrum Next": The campaign ended in May 2017 with more than 3,000 backers. During production, the team had to overcome many obstacles, and in February 2020 they started shipping the machines. The new home computers have not only found lots of happy users, but also spawned a community of game developers. There are a number of excellent 8-bit games and other software titles [3] that you can buy for the Spectrum Next, some of them are even available as physical media. The DVD boxes typically contain a booklet and an SD card that can be directly inserted into the Spectrum Next.

The SpecNext team then started a second Kickstarter, and when the pledging period ended in September 2020, they had attracted more than 5,000 backers. Again there were problems, and this time the situation was even worse than with the first Kickstarter: The main processor was no longer available in sufficient numbers, and so they had to do a complete redesign and switch to a different chip. In the end, all problems were solved, and in December 2023 the team started shipping the second generation of the ZX Spectrum Next. We're looking at one of those new machines in this article. If you've missed out on backing the project, you might see a third Kickstarter in the near future, and some machines will be for sale on eBay, but you can also get a clone or play with an emulator.

Lead Image courtesy of Hans-Georg Eßer

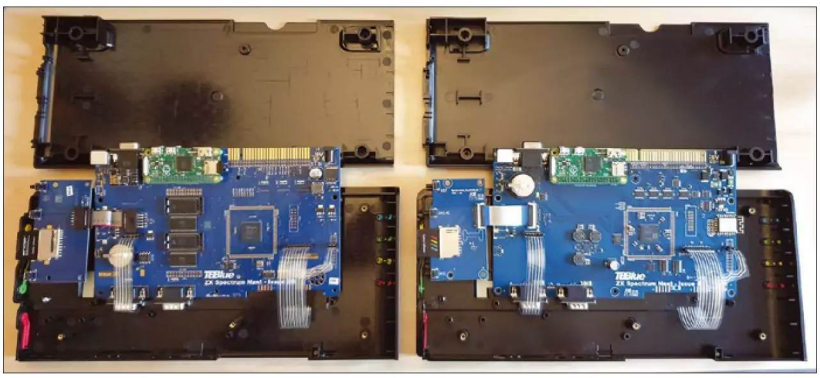


Figure 1: The KS1 Spectrum Next (left, Issue 2B) and the KS2 (right, Issue 4) have the same dimensions, and both get some help from a Raspberry Pi Zero.

Spectrum Next KS2

Many backers received their Spectrum Next just in time for Christmas – my own machine arrived early in December. Unpacking started with a small joke as there was an "R Tape cutting error, 0:1 - Do not open with a sharp instrument" warning in the old Spectrum screen font printed on the outer box – reminiscent of the Spectrum's "R Tape loading error" message that appeared when a parity error occurred while loading a program from a tape.

From the outside, the KS1 (Kickstarter 1) and KS2 models look almost identical. If you want to quickly decide which model you're looking at, check the screws: KS1 Spectrums came with silver screws, while the KS2 machines have black screws.

What's Inside?

All members of the Spectrum family of computers have used a Zilog Z80 processor, and the Spectrum Next is no different. However, instead of a real Z80, the Next machines contain an FPGA chip. That's a special kind of chip that can be programmed in a hardware description language and will then behave like the chip it is asked to emulate. This is not software emulation; an FPGA-based computer exactly replicates the functionality of classic processors, I/O devices, and other chips in hardware. In very simplified terms, this corresponds to emulation at the hardware level, and there are no restrictions as far as parallel processing on multiple components is concerned. That makes this approach much better than software emulation where concurrency often leads to timing problems.

The KS1 Spectrum Next used a Xilinx Spartan-6 XC6SLX16 FPGA, and that's the chip that was no longer available when the second Kickstarter ended. The team switched to another Xilinx chip, an Artix-7 XC7A15T. The new chip is similar to the old one but a bit more powerful.

The KS2 mainboard has been improved in many ways; there is a long video in which Mike Cadwallader describes some of the changes [4]. For example, if a KS1 machine was connected to an HDMI monitor, in some situations it would not shut off fully when its power was cut, because the monitor supplied some power over the HDMI connection. This has been resolved by adding an extra chip. Another problem with KS1 machines occurred when users connected incompatible Amstrad joysticks which could cause a short-circuit and reset the computer. The KS2 adds a fuse that prevents this reset.

The main features of the new Spectrum Next are the same as for the KS1 model (Figure 1). The FPGA is configured to provide a Z80N chip (basically a Z80 with some extra instructions) that can run at 3.5MHz, 7MHz, 14MHz, and 28MHz. Every machine comes with 2MB of RAM, and you can connect the Spectrum Next to both old and new monitors (RGB, VGA, HDMI in 50Hz and 60Hz), plug in two classic joysticks with 9-pin connectors, add a PS/2 mouse or keyboard, and load programs from a real tape recorder. There's WiFi connectivity, so you can download files from the Internet, and a Raspberry Pi Zero provides some extra functionality such as unpacking and playing back TZX archives – TZX is a popular archive format for ZX Spectrum software on cassettes [5]. The machines also have an expansion bus which is compatible with the one in the classic Spectrum computers, so you can use an old floppy disk controller or a ZX Printer.

The KS2 Next has an updated BASIC editor that displays the code with syntax highlighting (Figure 2), and the power cable now has a switch with Spectrum-colored stripes that match the colors on the keyboard.

Software

When you start the Spectrum Next and quickly press Space, you can configure the video mode and choose the Spectrum model that the FPGA will emulate – this way you can turn the machine into a Spectrum 48K or 128K. Merely rebooting the machine does not let you enter this menu; you need to power the computer off and on to get back here. In

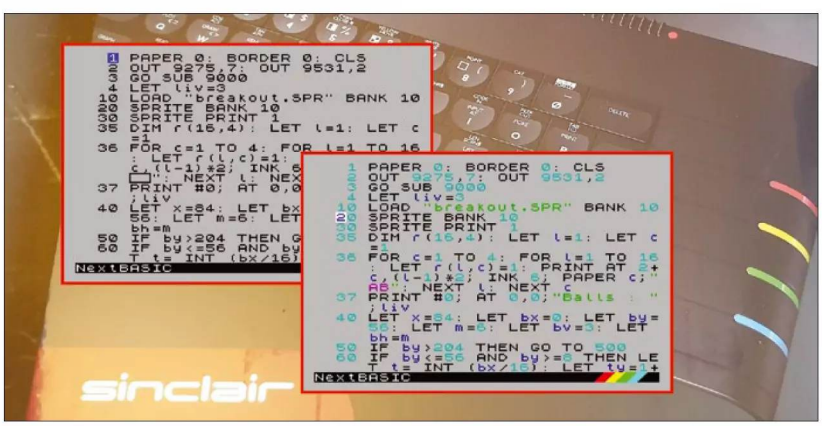


Figure 2: The BASIC editor has learned syntax highlighting: The screenshot in the back shows the colorless KS1 editor.

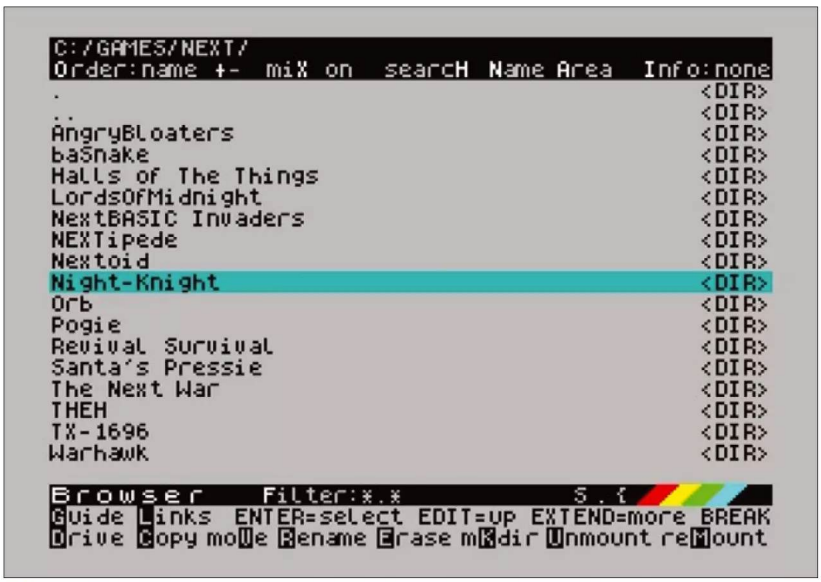


Figure 3: The file manager is called "Browser." This is where most sessions start.

most cases you will want to stick with the default settings, *ZX Spectrum Next (standard)*.

Next, the machine will display some welcome screens (which you can disable) and then show a menu. The first entry, *Browser*, launches the file manager which lets you navigate the folders on the SD card (Figure 3). Use the cursor keys to go from one entry to another and from one page to the next; select entries with Enter, go back to the parent directory with Edit. (Keys on the Spectrum have interesting names, for example, the modifier keys are called Symbol Shift and Caps Shift, and there is a Break key where you'd expect an Escape key.)

Native Next programs have a *.nex* extension; when you select such a file, the program will start after a few seconds. The list of native Next programs is limited, though it's steadily growing. You're not limited to native applications, though. Classic Spectrum programs work well with the modern machine, too. From the browser you can launch games and demos stored in *.tzx*, *.tap*, *.sna*, and *.z80* formats. Some of these store the audio data from old software tapes, and when you select such a file, it will load in the classic way which can take several minutes. You also get the flickering border that you may remember from the old days.

The browser also supports many other file types; for example, it will open text files, images, and videos in appropriate viewers. Yes, the Spectrum Next can do video playback, though the quality is not very high.

A special folder, *KS2Extras*, contains two brand-new games that have been developed for this Kickstarter campaign: Crowley World Tour 2 by Rusty Pixels and Night Knight. Backers can also download a brand-new version of the classic Head over Heels (Figure 4) from the Rusty Pixels website. That game was not finished in time to put it on the SD cards.

CP/M

When MS-DOS became the default operating system for IBM-compatible personal computers, it had not been invented from scratch. Instead, it had copied (and improved) many ideas from CP/M, an older system that was primarily used on computers with an Intel 8080 or a Zilog Z80 chip. With CP/M it was possible to use standardized, professional software, such as Wordstar or Turbo Pascal.

CP/M requires a floppy drive, and on the Spectrum Next you can emulate several such drives by assigning drive letters to disk image files. Using CP/M on the Spectrum Next has been simplified in comparison to the KS1 model, because the owners of the CP/M intellectual property have granted the right to use and distribute CP/M source code and binaries in 2022 [6]. Choose *More | CP/M* in the menu to auto-install a minimal CP/M system, and then do it again to boot the fresh installation. Via special `import` and `export` commands it is possible to transfer files between CP/M's disk images and the regular FAT32 filesystem on the SD card: CP/M programs do not understand the FAT32 format.

While CP/M apps should run on any CP/M machine, there are a lot of incompatibilities that can make it tough to run a program. For example, floppy disk formats are not standardized, which means that many disk images that are available on Archive.org and other websites will

Figure 4: Rusty Pixels has ported the classic "Head over Heels" to the Spectrum Next.

Figure 5: On the Spectrum Next you can install lots of games via downloader tools, such as GetIt.

be in a wrong format and CP/M on the Spectrum Next will not be able to access them. If you do manage to mount a disk image and access its files, programs may start but create garbled screen output, or your keyboard may lack an important key. I tried playing around with Turbo Pascal 3.0 [7], and while I was able to compile and run a simple Pascal program, the application occasionally asked me to press the Escape key – which does not exist on the Spectrum Next keyboard.

WiFi and Downloaders

Every KS2 Spectrum Next has built-in WiFi, and it's easy to connect to your local WLAN. A helpful application that uses the connectivity is the ZXDB Downloader [8]: It is pre-installed (*Apps* | *Wifi* | *zxdv-dl*), connects to a program database, and lets you enter a search term. It then shows you all matching entries, and if you've found something you want to try out, just download and run it. GetIt [9] is a similar tool, but it looks even more modern and displays low-resolution screenshots of the apps (Figure 5). GetIt is not pre-installed; the project website has installation instructions.

Modern Times

If you're interested and want to dig deeper into modern Spectrum machines, check out clones such as the N-Go [10] and more general FPGA systems such as the MiSTer FPGA on which you can also install a Spectrum Next core [11]. Those are options for getting Next-like hardware without waiting for the next Kickstarter campaign, but then you won't get the pretty keyboard.

Also, have a look at the SymbOS operating system [12]. It is available for several Z80-based computers, and the project website offers a preview package for the Spectrum Next that brings multi-tasking and a Windows-95-like desktop to the 8-bit computer (Figure 6). The responsiveness of the system is pretty amazing.

If you need some more reading material, have a look at the new *Next Magazine* [13], presented by Crash – it's a bi-monthly A5 publication by FusionRetroBooks that discusses old and new software titles for the Next and helps you learn programming. The very first issue has been published at the end of March.

It's amazing what 8-bit computers can do. Of course, at 28MHz the Spectrum Next is much faster than its predecessors, but only by a factor of 8. ■■■

Figure 6: SymbOS brings a Windows-like multi-tasking environment to the Spectrum Next.

Info

[1] Wikipedia, ZX Spectrum: *https://en.wikipedia.org/wiki/ZX_Spectrum*

[2] SpecNext: *https://www.specnext.com/*

[3] Software catalog: *https://www.specnext.com/software-directory/*

[4] Board changes: *https://www.youtube.com/watch?v=-q4FZhIH_SY*

[5] TZX: *http://k1.spdns.de/Develop/Projects/zasm/Info/TZX%20format.html*

[6] CP/M license: *http://www.cpm.z80.de/license.html*

[7] Turbo Pascal 3.0: *https://archive.org/details/Turbo_Pascal_v3.00a_1985_Borland_cpm_version*

[8] ZXDB Downloader: *https://github.com/em00k/next-zxdb-downloader*

[9] GetIt: *https://zxnext.uk/nextbuild/getit-released/*

[10] N-Go: *https://manuferhi.com/c/n-go*

[11] Spectrum Next Core for MiSTer FPGA: *https://github.com/MiSTer-devel/ZXNext_MISTer*

[12] SymbOS preview: *http://www.symbos.de/download.htm#marke3*

[13] *Next Magazine*: *https://fusionretrobooks.memberful.com/*

Lead Image courtesy of Hans-Georg Eßer

Run your Commodore emulators on a C64-look-alike PC

Keeping Up with the Commodore

Commodore is back: First a computer case via Kickstarter brings back the "bread box" form factor but lets you put a Mini-ITX PC mainboard inside, and now there's a new Linux distribution that fits that setup perfectly. *By Hans-Georg Eßer*

There have been many attempts to bring back the Commodore C64 or some machine that looks like it: You can get replacement mainboards that work in the original case, you can buy a THEC64 or its mini version from Retro Games Ltd, and there's also the MEGA65 project which created a modern FPGA-based version of the Commodore C65 – a successor to the C64 that never reached the manufacturing stage. All those machines have one thing in common: Their main goal is to let you run original C64 programs, either with an emulator or with an FPGA-based reimplementation of the original hardware.

For those of you who only care about the look and the unique bread-box form factor of the C64, the Commodore 64x (with an extra "x") is a modern computer in a C64-like case with a keyboard that resembles the C64 keyboard while offering a layout that works with PC operating systems. That machine was originally available from 2011 to 2012, it came with its own Commodore-branded Linux, and it was revived in a Kickstarter campaign in 2022 [1]. My Retro Computer Ltd. is currently offering a C64x barebone case (without a mainboard) for $215 or EUR198, whereas Kickstarter backers had options for fully pre-built setups (C64x Extreme and C64x Ultimate). The website [2] still lists those models, but they are out of stock.

In addition to the hardware, the 2011/2012 Linux distribution has also been updated: Commodore OS Vision 2.0 [3] is a lovingly customized version of MX Linux 21.3 (which itself is based on Debian GNU/Linux 11), and while it is intended for use with the C64x machines, there's no stopping you from running it on any modern computer (or in a virtual machine). We'll look at the software first, because that's what everyone will have access to. Then in the second part of the article we'll talk about the C64x case and its keyboard.

Commodore OS Vision

The 6GB ISO image (*CommodoreOS-20231213.iso*) boots into a live system that lets you try out all parts of the distribution. Turn on your speakers before the desktop appears, because you'll be treated to some computer voice output if you increase the volume: It's initially set to zero. When the desktop appears, Commodore OS first displays a welcome screen and then asks you to agree to terms and conditions – don't just close that window: If you do, you'll get a notification that the system cannot be used without agreeing and "will now self destruct," and it reboots after closing that note.

If you get past the terms and conditions, the MX Linux installer (Figure 1) starts. Close it if you simply want to play around with the live system. Otherwise, the installer lets you pick the right keyboard settings and partition your hard disk, then it copies the files to the disk. While that's happening, you can set

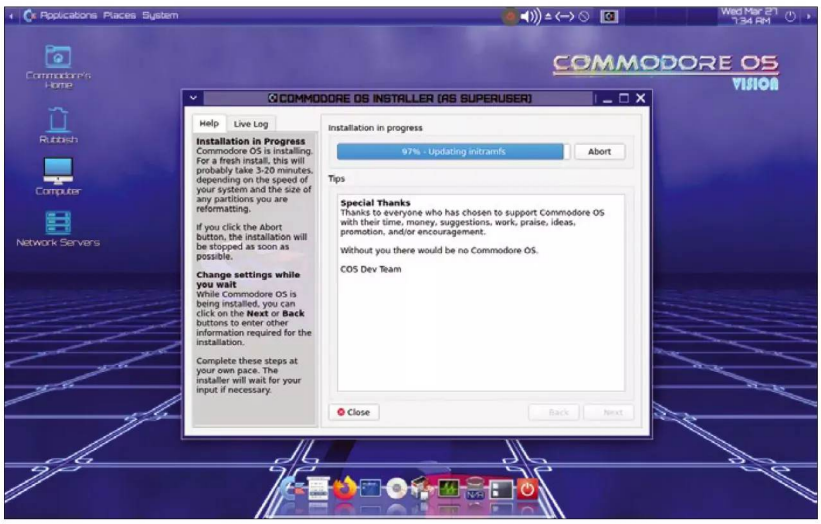


Figure 1: Commodore OS Vision uses the MX Linux installer, but without its user account setup function.

computer and domain names, change the workgroup for the Samba (SMB) server (or disable it), and select locales and the time zone (with both defaulting to Australia) – that's it. While the regular MX installer will also let you configure a user account and set a root password, Commodore OS skips that step. Instead it automatically creates a default user called *Commodore* with the password *C=*, and you can later auto-login to the installed system and become *root* via `sudo su` without entering a password. Once the installation is done, you can reboot. When you start Commodore OS Vision from the hard disk for the first time, it opens a series of information dialogs. Close one window and the next one appears.

The pre-installed software is a mix of pretty current and older software. You get Kernel 6.5.0 (from August 2023), an older Mate 1.24.1 desktop (2020) with wobbly window movement and a cube animation for desktop switching, LibreOffice 7.0.4 (again, from 2020) and some of the other applications that are part of Debian 11 (Bullseye). One of the additional APT repositories is misconfigured. An update via `apt update; apt upgrade` downloaded and installed 780MB of software. While the project website states that Commodore OS Vision 2.0 was released on December 13, 2023, the software identifies as "v2.0 (Beta 3)."

Emulators

The Linux distribution features a selection of home computer emulators. There are two sub-menus named *Commodore Emulators* and *Emulators* in the Applications menu. The Commodore section contains entries for the 8-bit machines (which all launch VICE 3.5.0) and the Amiga computers (using FS-UAE 3.0.5), but they all lack the required ROMs. For the C64, the emulator includes open source ROM replacements, so it will launch, but it cannot handle any of the included C64 games. In order to get the emulators to run properly you need to obtain official ROMs from Cloanto or extract them from an original machine's ROM chips.

Via the *System* | *Commodore OS* | *Cloanto Commodore ROM Setup* menu entry, you can download the "Free Express Version" from the C64 Forever website (Figure 2). Once that process has completed, you can run the C64 emulator with its proper ROMs. Note that it is preconfigured to simulate two joysticks when pressing any of the WASD (joystick #2) or cursor keys (joystick #1). If you need to enter text, double-click to switch from full-screen to window mode, then press Alt + O (or select *Settings* | *Settings* from the menu) to open the settings, navigate to *Input devices* | *Joystick* and disable the option *Enable keyboard joysticks* (Figure 3). Also, if you want to use a USB joystick, for example, the classic Competition Pro (which is period-correct for a C64), you need to open the drop-down list below *Joystick #2* and select the joystick (which will be the last entry in the list). If you later notice that a game expects a joystick in the first joystick port, open the dialog again and click on *Swap joysticks*. That will swap the settings for *Joystick #1* and *Joystick #2*.

Inside the C64 emulator, press Alt + W to enter Warp mode. The system will run

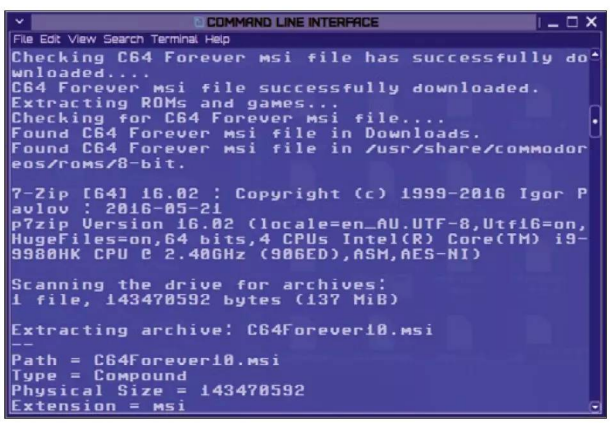


Figure 2: A setup script downloads and unpacks C64Forever.msi from Cloanto's server [4].

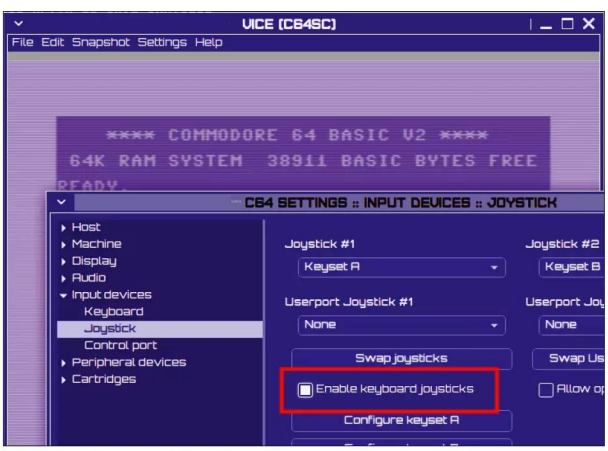


Figure 3: Disable this setting if you need to type text inside the emulator; otherwise, the WASD keys are interpreted as joystick movement.

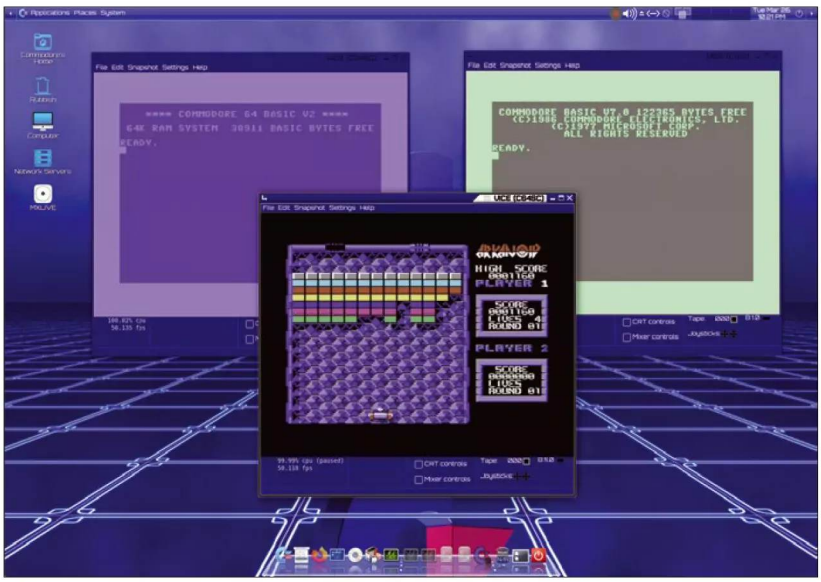


Figure 4: Let's play Arkanoid! In the background there are two more Vice windows showing the BASIC screens of the C64 and C128.

much faster, which is especially useful when you're loading a game from a virtual cassette tape or floppy. Press the same hotkey again to switch back to regular mode when the game is done loading – otherwise it might be unplayable. The distribution has a lot of games, demos, and audio files pre-installed: You can find them in the file manager by following the *Classic Software* link in your home directory. There are 88 C64 games in the *Commodore/C64/Games/Original_Commodore_Games* subfolder, and when you run the Cloanto installer, you get 100 additional games. Double-clicking one of the files in the file manager should automatically load them in the Vice emulator. If that does not work, right-click the file and select *Open With RunC64* in the context menu.

If you're not happy with the selection of C64 games, you can visit the C64 Software Library of the Internet Archive [5]; they've got everything. When I played around with this system, I had to try out Arkanoid, one of the classics (Figure 4).

The Amiga emulator (FS-UAE) requires ROMS, too. The Cloanto Commodore ROM setup tool will also help you with those, but they are not available for free. You can buy the Amiga Forever 10 Value Edition for EUR20 at Cloanto's AmigaForever store [6]. In the *Emulators* sub-menu you find Atari 2600, Atari Jaguar, DOS, Atari 16/32 Bit, MAME Arcade, MSX, NES, Playstation 2, Sega Genesis, Sega Saturn, ZX Spectrum, and Atari 8-bit emulators, but many of them will not start without providing ROMs or other media.

In addition to the emulators, you can also play modern games: Under *Applications | Gaming Services and Tools* you will find menu entries for Steam, GOG, Lutris, the Heroic Games Launcher, Wine, and PlayOnLinux.

Desktop

If you want to use Commodore OS Vision for regular tasks as well, you will eventually work with the shell. The terminal program has been configured to use a C64-like font, and the shell prompt ends in *READY.* on a separate line, mimicking the C64 BASIC prompt. It also features a blinking cursor. Run a terminal and the C64 emulator side-by-side, adjust the font size in the terminal, and you will see that they are lookalikes (Figure 5).

The desktop theme with its various shades of blue is pretty unique, too, and the system is noisy, both audibly and visually: Many actions, such as closing or resizing a window, are accompanied by system sounds, and the wobbling windows and animated dock icons create a restless atmosphere. Of course, you can change all those settings and get rid of elements you don't like: This is a Linux distribution, after all.

The Commodore 64x

The intended use of Commodore OS Vision is to install it on a modern Commodore-look-alike. I have pledged for the 2022 Kickstarter campaign by My Retro Computer, and as is custom with retrocomputing projects, it was more than a year and a half until a pretty red Commodore 64x case was delivered to my door. The barebone case does not contain a mainboard or a power adapter, so I bought an Asus Prime J4005I-C Mini-ITX board with a built-in Intel Celeron SoC, a Pico PSU (internal power supply), a universal (external) power supply, a small SSD drive, and an 8GB DDR4 memory stick. All of that fits nicely inside the case (Figure 6); there was only a minor problem with connecting the internal and external power supplies. I removed the USB hub on the right side of the case and put the Pico PSU's power plug in its place. The suggested solution is to drill a hole into the case, but I did not want to risk ruining it. I had opted for a hard-drive dock add-on which lets you insert and remove an SSD or hard disk without opening the case. However, that requires screwing a lever to the drive, and the eject procedure is difficult

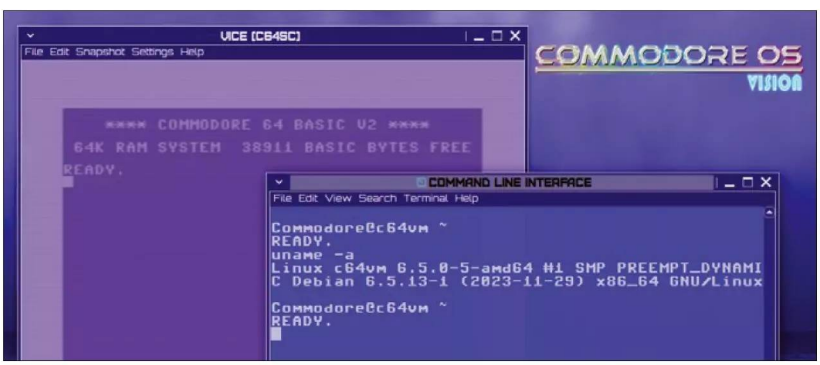


Figure 5: The shell uses a Commodore-C64-like font and prints "READY." at the end of its prompt.

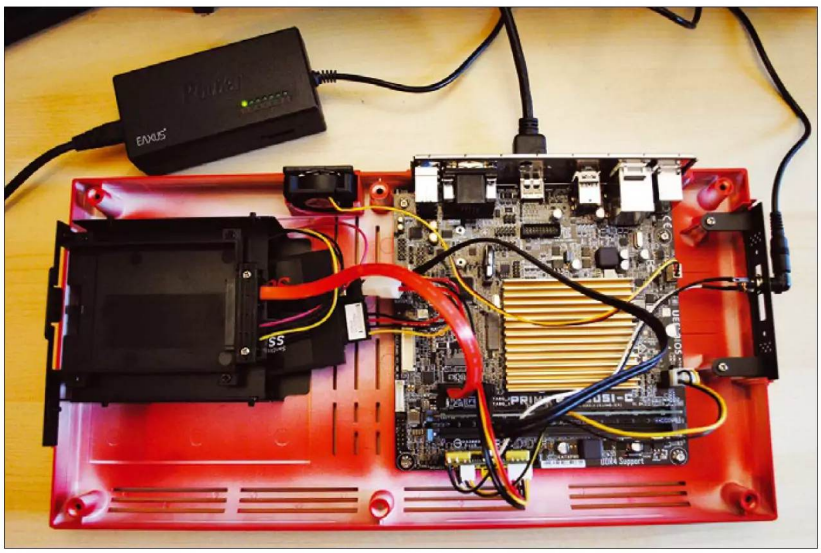


Figure 6: The Commodore 64x lets you install a Mini-ITX PC mainboard.

and might break the dock if done too often. There are better solutions for hot-swapping disks.

Whereas RetroGames' THEC64 uses the authentic keyboard layout of the original machines, the Commodore 64x comes with a somewhat similar but modernized layout (Figure 7). It lets you use the internal computer with a modern operating system, such as Windows or Linux. The keyboard has no function keys, but you can hold the *Fn* key and press *1* to *0*, - and = to generate *F1* to *F12* keystrokes. However, hotkeys such as Alt + F4 require some skillful finger placement, and you need both hands.

Figure 7: Same form factor, but very different – top to bottom: the Commodore 64x (2024), RetroGames THEC64 (2020), and Commodore C16 (1984).

It makes no sense that the keyboard has no function keys, because there are five wide special keys on the right (where a classic C64 has its four function keys). These keys would have been good candidates for function keys, but on the C64x they are labeled *Internet*, *E-Mail*, *Files*, and *Volume up/down*. When used with *Fn* they turn into media control keys.

Final Thoughts

While I've been critical of some of the design choices in both the Commodore OS Vision Linux distribution and the Commodore 64x case and keyboard, it is important to remember that these are both projects created by Commodore fans for Commodore fans. You cannot at the same time cram a full standard keyboard into such a case and preserve the look of the original machine, and it also makes sense that a dedicated Commodore-styled Linux distribution differs a lot from our standard systems because what would be the point if it didn't?

Mixing current technology and the classic case lets you build a pretty gaming rig that plays both the classic games of the old home computers and – depending on what kind of hardware you put inside – current games. I would not recommend the machine for daily work if it involves typing a lot: The keyboard layout is just too different, and while you could connect a regular keyboard, that would look a little silly. But for gaming, web browsing, watching videos, or consuming other media, it is a good alternative to a boring standard computer box, and it will be an eye-catcher in the living room if you place it next to your TV. For maximum nostalgia, connect an older monitor with a 4:3 aspect ratio (ideally a CRT) and run the C64 emulator in full screen.

The 1980s brand still exists in 2024, and several projects are keeping it alive. As they said in the ads: "Are you keeping up with the Commodore? 'Cause the Commodore is keeping up with you!" [7]. ■■■

Info

[1] Commodore 64x 2022 Kickstarter: *https://www.kickstarter.com/projects/myretrocomputer/the-commodore-64-its-back-and-better-than-ever/*

[2] MyRetroComputer shop: *https://myretrocomputer.com/order-now/*

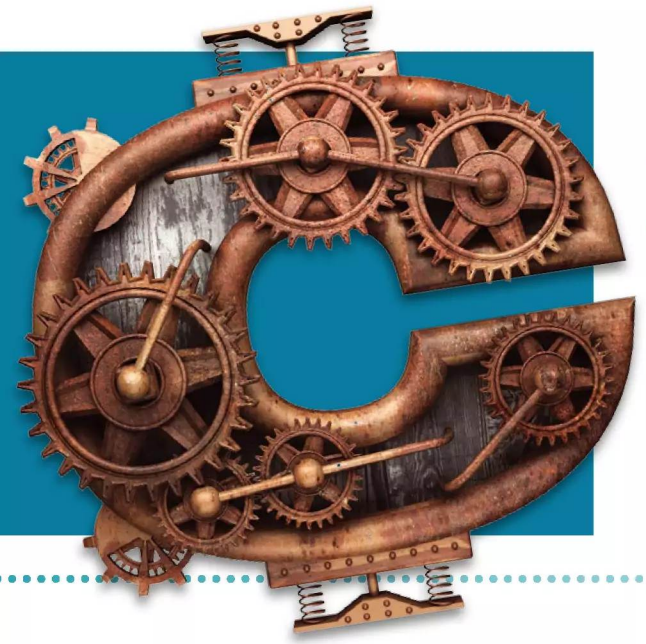
[3] Commodore OS Vision: *https://www.commodoreos.net/*

[4] Cloanto, C64 Forever: *https://www.c64forever.com/*

[5] Internet Archive, Software Library: C64: *https://archive.org/details/softwarelibrary_c64*

[6] Cloanto, Amiga Forever: *https://www.amigaforever.com/*

[7] Commodore tv commercial: *https://www.youtube.com/watch?v=95cGh9EeMIY*



BCPL for the Raspberry Pi

Before C

The venerable BCPL procedural structured programming language is fast to compile, is reliable and efficient, offers a wide range of software libraries and system functions, and is available on several platforms, including the Raspberry Pi. *By Dave Allerton*

In the 1960s, the main high-level programming languages were Fortran, Basic, Algol 60, and COBOL. To optimize code or to provide low-level operations, assembler programming offered the only means to access registers and execute specific machine instructions. BCPL, which was used as a teaching language in many universities, provided a language with a rich syntax, addressed the scoping limitations of the other languages, and had low-level operations such as bit manipulation and computation of variable addresses.

Where BCPL differs from the other languages is that it is typeless; all variables are considered to be a word, typically 16 or 32 bits. Programmers can access individual bits and bytes of a word, perform both arithmetic and logical operations on words, compute the address of a word, or use a word as a pointer to another word. One further novel aspect of BCPL is that the compiler is small and written in BCPL, producing intermediate code for a virtual machine and simplifying the development of the compiler for a wide range of computers. BCPL was used on mainframe computers and minicomputers in the 1970s and microprocessors in the 1980s.

The early developers of Unix were influenced by, and many aspects of C were adopted directly from, BCPL. Although BCPL also supported characters and bytes, the lack of richer types was addressed in C, which became the programming language of choice for Unix (and subsequently Linux), leaving BCPL mostly for academic applications. Several groups developed compilers, operating systems, software utilities, commercial packages, and even flight simulation software in BCPL, but for the most part, BCPL has been forgotten.

The demise of BCPL in both academia and industry is disappointing, particularly because it is a powerful teaching language, introducing students to algorithms, software design, and compiler design. Later, languages such as Pascal and Modula-2 became popular languages to introduce concepts in computer science but have been superseded by Java, Python, and C++. Whereas the learning curve for BCPL is small, enabling students to become productive in a short time, the complexity of languages such as C++ can be a barrier to students learning their first programming language.

The BCPL Language

The example in Listing 1 of a small BCPL program computes factorial values from 1! to 5!. Because C was developed from BCPL, the syntax of both languages is similar. The `include` directive in C is a `GET` directive in BCPL, the assignment operator = in C is := in BCPL, and the fences (curly brackets) { and } are identical. In C the address of a variable `a` is denoted by `&a`, whereas in BCPL it is

Lead Image © videodoctor, 123RF.com

given by `@a`. Indirection, or the use of pointers, is given by `*a` in C or `!a` in BCPL. Arrays are organized so that `a!b` in BCPL corresponds to `a[b]` in C.

The `GET` directive includes the common procedures and definitions needed in the compilation of a program. The procedure `start` is similar to `main` in C, where the `VALOF` keyword denotes that `start` is a function with the result returned by the `RESULTIS` keyword. The variable `i`, a local variable of the procedure `start`, is implicitly defined at the start of the `FOR` loop, which is executed five times. The `writef` function is similar to `printf` in C. The recursive function `fact` tests whether `n` is zero and returns either `1` or `n*(n-1)!`, where the parameter `n` is a local variable of the procedure `fact`.

In BCPL, a variable is defined as a word that can represent an integer, a bit pattern, a character, a pointer to a string of characters, a floating-point number, or an address. A programmer can apply arithmetic operators, logical operators, shift operators, an address operator, or indirection to a variable – the compiler assumes that the programmer knows what they are doing and, subject to syntactic and sematic compilation checks, places very few constraints on programming constructions. Arguably, C and BCPL fall into the category of languages that provide almost unlimited power for a programmer with very few checks on their intention.

Both C and BCPL allow sections of a program to be compiled separately (e.g., to provide a library of functions). Global variables and procedures in BCPL, which are similar to external variables and functions in C, can be accessed by all sections of a program, whereas static variables are only accessible from the section in which they are declared. The other category of variables is local or dynamic variables, which are declared and used in the same way as C. When a local variable is declared, space is allocated on a stack, which grows and shrinks dynamically, typically on entry to and exit from a procedure, respectively, enabling procedures to be called recursively.

Portability

BCPL was developed by Martin Richards in the Computer Laboratory at the University of Cambridge. His more recent Cintcode implementation is extensive and provides numerous examples of coding, mathematical algorithms, and even operating system functions. The advantages of this implementation are considerable: It is fast to compile, is reliable and efficient, and offers a wide range of software libraries and system functions. It is also available on several platforms, including the PC and the Raspberry Pi. The only drawback is the loss of speed from interpreting the compiled code.

I refer you to Martin Richard's textbook [1], and his website [2] which includes a version of Cintcode, that is straightforward to download and implement on an RPi. Also, a guide directed at young people programming a Raspberry Pi [3] provides an extensive description of BCPL and the Cintcode implementation and numerous examples of BCPL programs.

For the programmer intending to write applications in BCPL that exploit the processing power of the ARM cores of a Raspberry Pi, a BCPL compiler generating ARM instructions directly is likely to produce code which runs considerably faster than interpreted code. For other users less concerned with processing speed, the tools and support provided by the Cintcode implementation of BCPL offer a stable and reliable platform.

BCPL for the Raspberry Pi

The arrival of the Raspberry Pi with its ARM cores, network connection, sound and video outputs, USB ports, and I/O interface running under the Linux operating system has encouraged the development of a range of programming languages for this platform. A code generator for BCPL that I developed compiles BCPL directly to ARM machine code, which can be linked with the standard Linux `gcc` toolset. The compiler (7,000 lines) compiles itself in less than 0.2 seconds on a Raspberry Pi 4B.

This 32-bit implementation of BCPL compiles a BCPL program `prog.b` to `prog.o`, where `prog.o` is a Linux object module linked with two libraries – `blib.o` and `alib.o` – by the `gcc` linker to produce an executable ELF module, `prog`. The library `blib.b` is written in BCPL and contains the common BCPL library functions. A small library `alib.s` is written in Linux assembler and contains low-level functions to access the Linux runtime environment.

Although the `gcc` linker builds the executable program, the object code produced by the compiler contains only blocks of position-independent code, requiring no relocation. At runtime, `alib` initializes the BCPL environment, setting up the workspace for the stack and global and static variables. Strictly, `gcc` is only used to generate a Linux-compatible module that can be loaded, whereas the linking of a BCPL program and libraries is performed by `alib`.

Notes for Developers

The compiler uses registers *r0* to *r9* for arithmetic operations, logic operations, and procedure calls. The code generator attempts to optimize the code by keeping variables in registers, minimizing the number of memory accesses.

Register *rg* points to the global vector, and register *rp* is the BCPL stack pointer or frame pointer. Procedure linkage, procedure arguments, and local variables are allocated space in the current frame. Stack space is claimed on entry to a procedure and released on return from a procedure. The link register *lr* holds the return address on entry to a procedure and can also be used as a temporary register within a procedure. The system stack pointer *sp* is not used by the BCPL compiler, so it can be used to push and pop temporary variables. The compiler uses the BCPL stack for procedure linkage and the storage of local variables. It should be noted that the ARM core is a pipelined processor and reference to *pc* during an instruction implies the address of the

Listing 1: 1! to 5! in BCPL

```
01 GET "libhdr"
02
03 LET start() = VALOF
04 {
05     FOR i = 1 TO 5 DO
06       writef("fact(%n) = %i4*n", i, fact(i))
07     RESULTIS 0
08 }
09
10 AND fact(n) = n=0 -> 1, n*fact(n-1)
```

current instruction + 8 for most instructions. The program counter *pc* is used in the code generation of relative addresses used for procedure calls and branches and also in `switchon` expressions in BCPL.

Although Linux libraries are not explicitly linked, the *libc* library is available to BCPL programs. Fortunately, the register calling mechanisms of the GNU `gcc` tool chain and BCPL are distinct and independent. The BCPL stack grows upward, with no access or modification to the system stack. In C, the stack grows downward, and local variables are stored relative to the system stack pointer *sp*. Consequently, it is possible to call C functions from BCPL.

In the ARM Procedure Call Standard (APCS), the first four arguments are loaded into registers *r0*, *r1*, *r2*, and *r3*, respectively, and a result is returned in register *r0*. The address of the procedure is computed, and the procedure is called by an appropriate branch and link (*bl*) instruction or a branch, link, and exchange instruction (*blx*).

However, C and BCPL have two important differences: (1) BCPL strings are defined by the string size in the first byte followed by the 8-bit characters of the string, whereas strings in C are arrays of 8-bit characters terminated with a zero byte. BCPL strings must be converted to C strings, if calling C. (2) Addresses of variables, vectors, and strings in BCPL are word addresses, whereas they are machine addresses in C. Passing an address from BCPL to C requires a logical left shift of two places, and passing an address from C to BCPL requires a logical right shift of two places. Care is needed with strings in C because they are not necessary aligned on 32-bit word boundaries.

Table 1: BCPL Registers

Register	Name	Function
0	*r0*	Data register 0
1	*r1*	Data register 1
2	*r2*	Data register 2
3	*r3*	Data register 3
4	*r4*	Data register 4
5	*r5*	Data register 5
6	*r6*	Data register 6
7	*r7*	Data register 7
8	*r8*	Data register 8
9	*r9*	Data register 9
10	*rg*	Global vector
11	*rp*	BCPL stack
12	*ip*	Unused
13	*lr*	Link register
14	*sp*	System stack pointer
15	*pc*	Program counter

In both C and BCPL, the registers *r0*-*r9* are not preserved across procedure calls. Additionally, the BCPL registers *rp*, *rg*, and *lr* cannot be guaranteed to be preserved in C, and it is advisable to store these registers before calling a C procedure. In practice, they can be pushed onto the system stack and popped on return by:

```
push {rg, rp, lr}
pop {rg, rp, lr}
```

The code produced by the code generator for the factorial example is shown in Listing 2 with comments to explain specific instructions. Note that register *r0* is reloaded at location 0x38 because it is reached by code from locations 0x34 and 0x74; consequently, the content of register *r0* is not assured. Additionally, the reference to the string

```
"fact(%n) = %i4*n"
```

is not known at location 0x4C when the instruction is generated; therefore, a full static reference is generated with the offset 0x00000028 stored at location 0x90.

Listing 2: Code Generator Output

```
 0:   0000003c      data                          Section size (words)
 4:   0000fddf      data                          Section identifier
 8:   6361660b      data                          Section name "fact"
 c:   20202074      data
10:   20202020      data
14:   0000dfdf      data                          Entry identifier
18:   6174730b      data                          Procedure name "start"
1c:   20207472      data
20:   20202020      data
24:   e8a4c800      stmia   r4!,{fp,lr,pc}        Standard procedure entry
28:   e884000f      stm     r4,{r0,r1,r2,r3}
2c:   e244b00c      sub     fp,r4,#12
30:   e3a00001      mov     r0,#1                 Initial value i
34:   e58b000c      str     r0,[fp,#12]           Save i
38:   e59b000c      ldr     r0,[fp,#12]           Load i
3c:   e28b4024      add     r4,fp,#36             Set new stack frame
40:   eb000017      bl      0xa4                  Call f(i)
44:   e1a02000      mov     r2,r0                 Arg 3 = f(i)
48:   e59b100c      ldr     r1,[fp,#12]           Arg 2 = i
4c:   e59fe03c      ldr     lr,[pc,#60]           Arg 1 = "fact(%n) = %i4*n"
50:   e08f000e      add     r0,pc,lr              pc offset
54:   e1a00120      lsr     r0,r0,#2              BCPL address
58:   e28b4010      add     r4,fp,#16             Set new stack frame
5c:   e59ae178      ldr     lr,[sl,#376]          Global writef
60:   e12fff3e      blx     lr                    Call writef()
64:   e59b000c      ldr     r0,[fp,#12]           Load i
68:   e2800001      add     r0,r0,#1              Increment by 1
6c:   e58b000c      str     r0,[fp,#12]           Store i
70:   e3500005      cmp     r0,#5                 Check end of for-loop
74:   daffffef      ble     0x38                  Continue for-loop
78:   e3a00000      mov     r0,#0                 Return 0
7c:   e89b8800      ldm     fp,{fp,pc}            Standard procedure return
80:   6361660f      data                          String "fact(%n) = %i4*n"
84:   6e252874      data
88:   203d2029      data
8c:   0a346925      data
```

Installation

The file `bcpl_distribution` [4] contains the files shown in Table 2. The object

Listing 2: Code Generator Output (continued)

```
90:   00000028   data
94:   0000dfdf   data                              Entry identifier
98:   6361660b   data                              String "fact"
9c:   20202074   data
a0:   20202020   data
a4:   e8a4c800   stmia   r4!,{fp,lr,pc}           Standard procedure entry
a8:   e884000f   stm     r4,{r0,r1,r2,r3}
ac:   e244b00c   sub     fp,r4,#12
b0:   e3500000   cmp     r0,#0                    Test n=0
b4:   1a000001   bne     0xc0                     Skip if not
b8:   e3a00001   mov     r0,#1                    Return 1
bc:   e89b8800   ldm     fp,{fp,pc}               Standard procedure return
c0:   e59b000c   ldr     r0,[fp,#12]              Load n
c4:   e2400001   sub     r0,r0,#1                 Decrement n
c8:   e28b4010   add     r4,fp,#16                Set new stack frame
cc:   ebffffff4  bl      0xa4                     Call f(n-1)
d0:   e59b100c   ldr     r1,[fp,#12]              Get n
d4:   e0000190   mul     r0,r0,r1                 Return n*(n-1)
d8:   e89b8800   ldm     fp,{fp,pc}               Standard procedure return
dc:   00000000   data                              No statics
e0:   00000000   data                              Start of global vector
e4:   00000001   data                              Global 1 (start)
e8:   00000024   data                              Offset to global 1
ec:   0000005e   data                              Maximum global of the section
```

files `bcpl.o` and `blib.o` each contain a block of position-independent code. The assembler module `leader.s` provides a means of identifying the start of a BCPL program. The runtime library `alib.s` is written in assembler code and includes data regions for the global variables and static variables and is linked to the GNU C runtime library `libc`. Note that the files `bcpl.b` and `bcplfecg.h` are only needed to rebuild the compiler and are not required for user applications.

The distribution also includes several BCPL examples and a user guide (Table 3). The programs `queens.b` and `primes.b` are described in Martin Richard's excellent notes to young people interested in programming the Raspberry Pi [3].

To install BCPL on a Raspberry Pi Model 3 or 4, create a new directory and copy the distribution files in `bcpl-distribution` to this directory. Alternatively, to install BCPL on a Raspberry Pi Model 2, copy the distribution files in `bcpl-distribution-rpi2`. In a terminal shell, enter the commands

```
unzip bcpl-distribution.zip
as leader.s -o leader.o
as alib.s -o alib.o
gcc leader.o bcpl.o blib.o alib.o -o bcpl
```

to build and test the compiler (> denotes the Linux prompt).

For a first compiler test, compile and run the program `fact.b`, which prints the factorial numbers from 1! to 5!:

```
./bcpl fact.b -o fact
./fact
```

Further confidence tests rebuild the BCPL compiler `bcpl.b` with the BCPL compiler and build the library `blib.b`:

```
./bcpl bcpl.b -o bcpl
./bcpl -c blib.b
```

The BCPL library files and the compiler can then be copied to the appropriate Linux shared directories:

```
sudo mkdir /usr/include/BCPL
sudo cp libhdr.h /usr/include/BCPL/
sudo cp bcpl /usr/bin/
sudo cp leader.o /usr/lib/
sudo cp blib.o /usr/lib/
sudo cp alib.o /usr/lib/
```

The remaining BCPL programs can now be compiled and run with the command `bcpl` rather than `./bcpl`. The compiler searches for library files in the working directory before searching the directories `/usr/include/BCPL` and `/usr/lib`.

Nostalgia

The influence of BCPL on the development of C and its later variants cannot be overstated. The availability of BCPL for the Raspberry Pi allows old computer science students to dust off copies of their programs, which should run directly on the Raspberry Pi. BCPL was used extensively in many UK university computer science departments. The portable multi-tasking operating system Tripos was written entirely in BCPL in the Computer Laboratory at the University of Cambridge and used in early versions of the Commodore Amiga, in the automotive industry, and in financial applications.

The logic simulator HILO-2 (the forerunner of Verilog) was developed in BCPL. Numerous utilities, including the early word processor `roff` were written in BCPL. Before the availability of floating-point hardware, I adapted BCPL compilers for the Motorola 6809 and 68000 processors to use scaled fixed-point arithmetic in real-time flight simulation. ■■■

Table 2: bcpl_distribution

File Name	Function
alib.s	A runtime library written in GNU ARM assembler
blib.b	The BCPL runtime library, written in BCPL
blib.o	A precompiled version of the BCPL runtime library blib.b
bcpl.b	The BCPL compiler and code generator to run under Linux
bcpl.o	A precompiled version of the BCPL compiler and code generator
bcplcg.b	The code generator used by the BCPL compiler for the ARM processor
bcplfecg.h	A header file used by the code generator
leader.s	A small assembler program only used to locate the start of a BCPL program
libhdr.h	The standard BCPL header

Table 3: BCPL Examples and User Guide

File	Content
bench.b	A small program to time the execution of a small fragment of BCPL
fact.b	A small program to print the factorial numbers from 1! to 5!
primes.b	A small program to print the prime numbers less than 1,000
queens.b	An implementation of the "Queens" problem for 1 to 16 pieces
guide.pdf	A guide to BCPL for the Raspberry Pi, including installation notes

Info

[1] Richards, Martin. *BCPL: The Language and its Compiler*, revised ed. Cambridge Univ Press, 2009: *https://www.amazon.com/BCPL-Language-Compiler-Martin-Richards/dp/0521286816*

[2] Martin Richards: *https://www.cl.cam.ac.uk/~mr10/*

[3] Richards, M., *Young Persons Guide to BCPL Programming on the Raspberry Pi Part 1*. Cambridge (UK): Computer Laboratory, University of Cambridge, revised 23 Oct 2018: *https://www.cl.cam.ac.uk/~mr10/bcpl4raspi.pdf*

[4] Code for this article: *https://linuxnewmedia.thegood.cloud/s/XnzsiEKtagjHKr3*

Author

Dave Allerton obtained a PhD from the University of Cambridge in 1977 and worked in the defense industry before spending 10 years at the University of Southampton as a lecturer in computing. He was the Professor of Avionics at Cranfield University before moving to the University of Sheffield as Professor of Computer Systems Engineering, where he is currently an Emeritus Professor. He is also a Visiting Professor at Cranfield University and at Queen Mary University of London. His research activities include flight simulation, computer graphics and real-time computing. He is author of two textbooks, *Principles of Flight Simulation* (Wiley, 2009, ISBN 978-0-470-75436-8) and *Flight Simulation Software: Design, Development and Testing* (Wiley, 2022, ISBN 978-1-11973-767-4).

Hone Your Skills
– with –
Special Issues!

Get to know Shell, LibreOffice, Linux, and more from our Special Issues library.

The *Linux Magazine* team has created a series of single volumes that give you a deep-dive into the topics you want.

Available in print or digital format

background image © roystudio, 123RF.com

Check out the full library!

shop.linuxnewmedia.com

Flashing and programming an LED display

Project Blinking Lights

The Ulanzi TC001 is a low-budget LED display that lets you customize the firmware and add some homemade scripts. *By Mike Schilli*

External displays that continuously show data without a real screen, even when the computer is taking a nap, are a genuine upgrade to any office. Of course, they can be used to display the time or weather, but they can also perform unusual tasks tailored to your needs. The reasonably priced Ulanzi TC001 [1] ended up on my doorstep within a week for around $60, after traveling all the way from China to the USA. My original idea was to use it to build a "Wealth Clock" that shows the current gold level in all my money stores so that I know how wealthy I am at any given time.

Flashing Custom Firmware

The LED display has a retro feel. Of course, there are higher-resolution displays available today, but the LED display is definitely suitable for displaying short character strings and gives you a sort of cozy Tetris feeling at the same time. The included firmware can only do mundane tasks such as displaying the time, the date, and the battery level, but the Awtrix [2] project offers open source firmware including a browser-based instant flashing tool that turns the device into a Jack of all trades in next to no time. Figure 1 shows how the new firmware boots up.

The device does not offer much RAM, and the processor is a modest ESP32. Although this microcontroller can handle WiFi and Bluetooth, its performance cannot be compared to that of a modern CPU. This is why more demanding applications aren't running directly on the Ulanzi. Instead, they are chugging along on an external computer with more power, which then uses an API command to periodically tell Awtrix what to

Lead Image © greenflame, 123RF.com

Figure 1: Booting the Ulanzi after flashing with the Awtrix firmware.

Author

Mike Schilli works as a software engineer in the San Francisco Bay Area, California. Each month in his *Linux Magazine* column, which has been running since 1997, he researches practical applications of various programming languages. If you email him at *mschilli@perlmeister.com* he will gladly answer any questions.

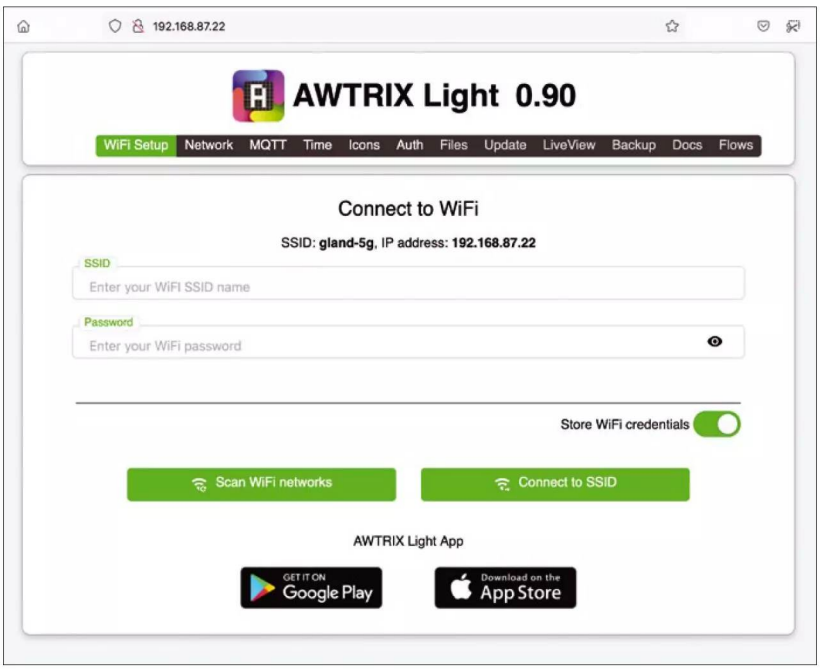


Figure 2: The Awtrix admin interface in the web browser.

display. After completing the boot process, the firmware rotates through all of its configured standard apps: time/date, temperature, humidity provided by its internal sensors, and current battery strength. But that's not the objective here. Instead, we will be disabling the standard apps one by one in order to upload our own custom apps.

Figure 3: The display counts the days, hours, and minutes until a birthday.

Figure 4: Followers and uploads on my YouTube channel.

Perpetual Cycle

To do this, you need to press and hold the center button with the circle at the top of the Ulanzi for about two seconds; this will force Awtrix to jump into the admin console. One of the submenus there is named *Apps*.

Another short press on the circle button shows the status of the first app (e.g., the remaining capacity of the built-in battery). The display can be operated for around five hours without a power cable using the built-in battery – but this is unlikely to be useful to anyone, because it definitely requires a power socket for continuous operation.

Pressing the arrow buttons to the left or right now reveals additional apps such as the temperature or humidity display, or the time and date. Briefly pressing the circle button switches the displayed app off or back on again; the firmware acknowledges this by displaying *off* or *on*.

A long press on the circle button causes the console to jump back up to the next level and ultimately return to the infinite app cycle. Once you have disabled all the default apps, you will now see nothing but a dark display.

Meaningless Password

The Awtrix firmware's web UI and API can be protected with a username and password using the Admin console (Figure 2). However, the mini web server on the device then expects login credentials for each request via basic authorization using unprotected HTTP. That is not exactly state-of-the-art: Anyone listening in on the WiFi network can sniff the password.

To integrate new custom apps into the firmware display loop, clients either can use the MQTT interface, which is particularly popular for home automation systems, or send

Listing 1: countdown.go

```
package main
import (
  "fmt"
  "time"
)
func DHMUntil(until time.Time) string {
  dur := time.Until(until)
  days := int(dur.Hours() / 24)
  hours := int(dur.Hours()) % 24
  mins := int(dur.Minutes()) % 60

  return fmt.Sprintf("%02d:%02d:%02d", days, hours, mins)
}
```

commands via the web API. The latter is not well-documented on GitHub, but, ultimately, a `POST` request to the Ulanzi's IP on the WiFi network is all it takes. After flashing with the new firmware, the device starts in AP mode. If you select the new `awtrix_XXX` WiFi network on a laptop or smartphone, you can send the WiFi access credentials for the home network to the Ulanzi in the browser that then opens. After a reboot, the Ulanzi then connects to the WiFi network and grabs an IP, which it shows on the display when booting up.

API calls for setting up new apps will be sent to this IP and the path `/api/custom`; they also require a (freely selectable) name for the app and a JSON blob with the desired display content.

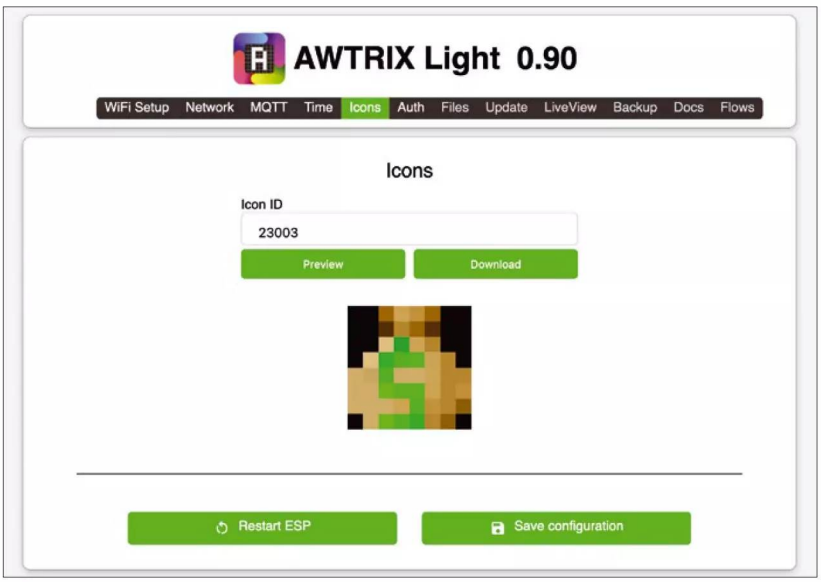


Figure 5: A money bag as a symbol for the wealth clock.

Listing 2: api.go

```
01 package main
02 import (
03   "bytes"
04   "encoding/json"
05   "fmt"
06   "net/http"
07 )
08
09 const baseURL = "http://192.168.87.22/api/custom"
10
11 type apiPayload struct {
12   Text     string `json:"text"`
13   Rainbow  bool   `json:"rainbow"`
14   Duration int    `json:"duration"`
15   Icon     int    `json:"icon"`
16 }
17
18 func postToAPI(name string, p apiPayload) error {
19   url := baseURL + "?name=" + name
20   jsonBytes, err := json.Marshal(p)
21   if err != nil {
22     return 0, err
23   }
24
25   resp, err := http.Post(url, "application/json", bytes.
       NewBuffer(jsonBytes))
26   if err != nil {
27     return 0, err
28   }
29   defer resp.Body.Close()
30
31   if resp.StatusCode != http.StatusOK {
32     return fmt.Errorf("%v", resp.StatusCode)
33   }
34   return nil
35 }
```

Listing 3: youtube.go

```
01 package main
02
03 import (
04   "context"
05   "google.golang.org/api/option"
06   "google.golang.org/api/youtube/v3"
07   "log"
08 )
09
10 const ChannelID = "UC4UlBOISsNy4HcQFWSrnV5Q"
11 const ApiKey = "AIzaSyZmOrarSDWqrnAwIKkWGzjOvaVQtyvPokB"
12
13 func youtubeStats() (uint64, uint64, error) {
14   ctx := context.Background()
15   service, err := youtube.NewService(ctx, option.
       WithAPIKey(ApiKey))
16   resp, err := service.Channels.List([]
       string{"statistics"}).Id(ChannelID).Do()
17   if err != nil {
18     log.Fatalf("%v", err)
19   }
20
21   if len(resp.Items) == 0 {
22     log.Fatal("Channel not found")
23   }
24
25   stat := resp.Items[0].Statistics
26   return stat.SubscriberCount, stat.VideoCount, nil
27 }
```

Birthday Countdown

First of all, I decided to add a new app to the display that counts down the days, hours, and minutes until a specified date, for example, a birthday (Figure 3). Listing 1 uses the `DHMUntil()` function to calculate the time span between the current time and the corresponding date. It then divides the resulting number of hours by 24 to compute the number of days. A Mod 24 operation extracts the remaining hours from this, and a Mod 60 on the minutes extracts the remaining minutes.

What you get back is a string in a `DD:HH:MM` format, which the API call in Listing 2 shows on the display. It is up to the control computer how often the countdown is refreshed. If, for example, a cron job only starts every 15 minutes, the counter will lag behind by a quarter of an hour worst case.

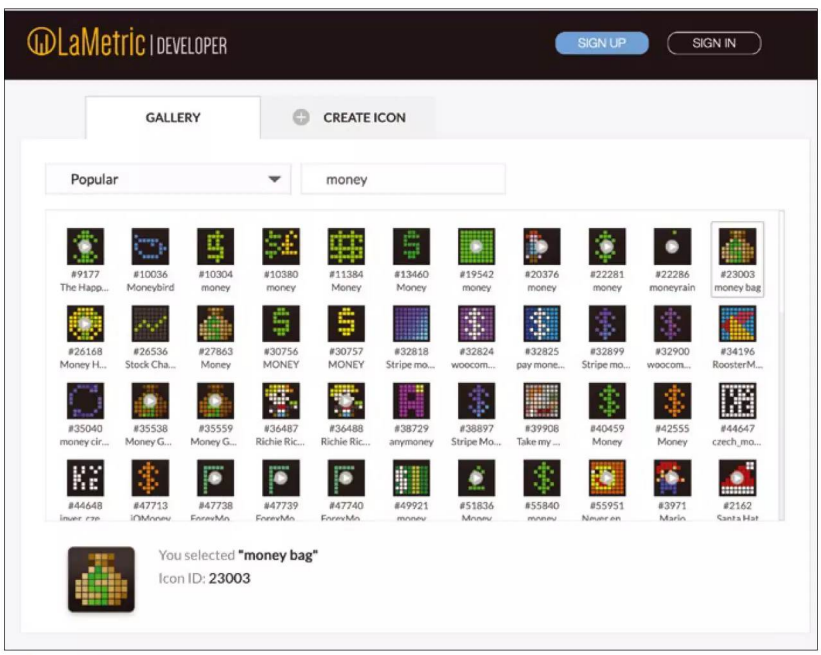


Figure 6: Icons are available for download from the LaMetric developer page.

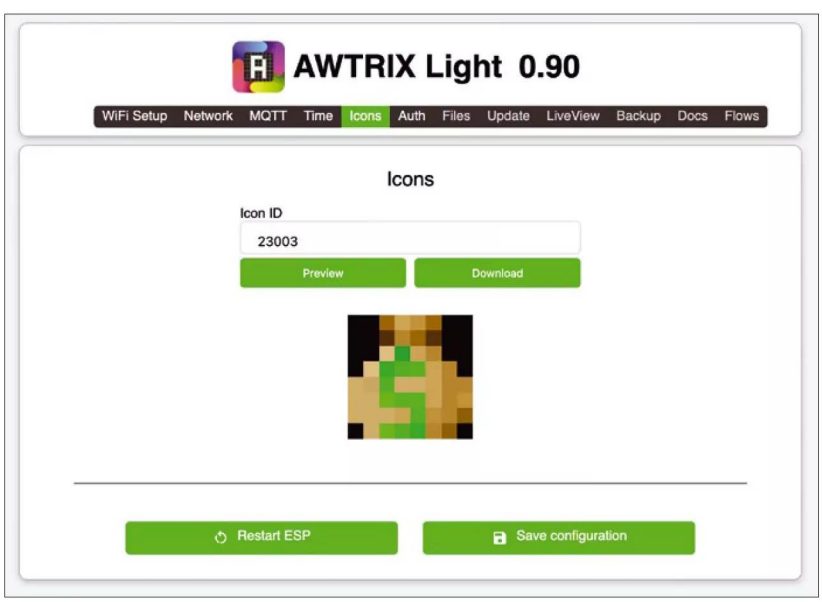


Figure 7: Awtrix downloads and displays these icons by reference to their numeric IDs.

Communication with the Awtrix firmware's web server API is handled by Listing 2 using the `apiPayload` type structure from line 11. The `json.Marshal()` packer converts the structure into JSON format in line 20 referencing the back-quoted instructions in the structure to do so. For example, the content of the Go `Text` attribute, which holds the character string to be displayed, is `text` (i.e., lowercase) in JSON by convention, because JSON fields traditionally start with lowercase letters, whereas public Go structure fields start with capital letters.

The `postToAPI()` function from line 18 expects two parameters from the caller: the name of the application and an `apiPayload` type structure. The `apiPayload` type structure contains the text to be displayed (in `Text`), the `Rainbow` flag (true value for a colorful display), and the display duration in seconds in `Duration`. You can optionally add an icon so that the viewer can visually determine which app the displayed value is associated with.

The `Post()` function from the Go *net/http* standard package then sends the JSON blob to the web server, specifying the `application/json` MIME type. The MIME type is mandatory; otherwise, the server will not route the call correctly. After checking the HTTP response for errors, the function finally returns.

Like and Subscribe

In another app, I wanted the Ulanzi to display the number of subscribers to my YouTube channel and the number of videos uploaded to date (Figure 4). Listing 3 illustrates how the control computer retrieves the desired numerical values from YouTube. Google requires a valid API key to access the data; you can obtain this from the Cloud Console as shown in my Programming Snapshot column in the March 2024 issue of *Linux Magazine* [3].

The official YouTube API client Go library used in the listing makes it easy to obtain statistics for a channel. On top of that, it removes the need for developers to extract the desired values from the mess of JSON in the server response. The channel ID for identifying the desired channel is hard-coded in line 10 and the API key in Line 11.

The code calls `NewService()` to create a service object in line 15. It then invokes the API client's `List()` function with the `statistics` parameter to extract the channel's statistics metadata. The return value is a list containing exactly one match, which line 25 drills down on. Line 26 then extracts the desired values for `SubscriberCount` and `VideoCount` from the data structure.

Pixelated Icons

If you install multiple apps, and the display constantly toggles between them, icons are a great way to show users which app generated the text currently on display. Having said this, it is not so easy to create a meaningful graphic on a mini matrix of 8x8 pixels on the display to leave room for the actual data.

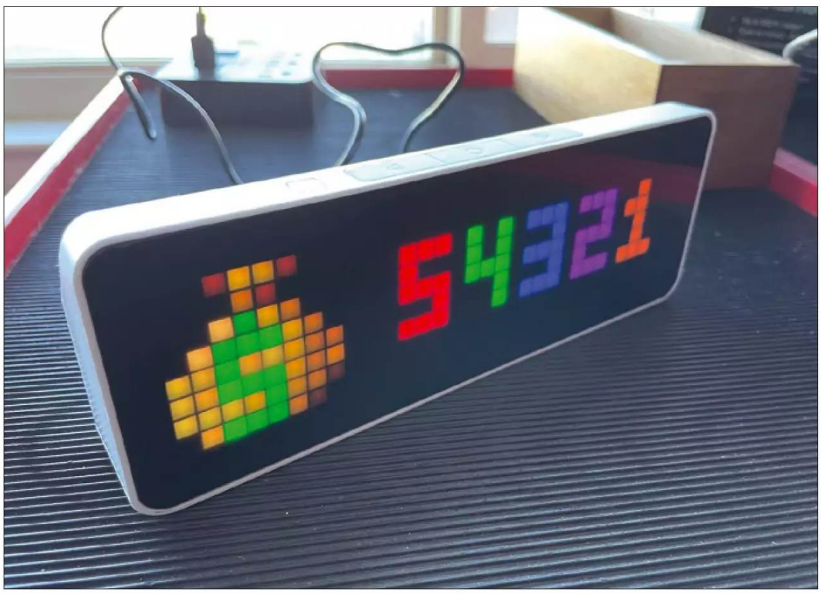


Figure 8: Symbolic display of the author's personal wealth.

Interestingly, the Ulanzi TC001 with Awtrix works around this by using predefined icons (Figure 5) from the developer site of the more expensive competitor product LaMetric [4]. You can search for suitable icons there using keywords (Figure 6) and write down their IDs. Later, on the Awtrix admin page, the small pixel artworks can be referenced by this numerical value in the *Icons* tab (Figure 7). At the touch of a button, Awtrix then downloads the respective icon to the firmware and displays it in the first field of an app whenever the JSON data of an app sent to the display references the corresponding numerical icon ID in the `icon` field.

After calling the API from the compiled Go program, the display will later show a YouTube-style red play button as an icon, as you can see in Figure 4. It told me that my personal channel on the platform now has 290 subscribers and that I have uploaded no fewer than 85 videos on cooking and car repairs.

Uncle Scrooge's Monitor

As for my personal wealth clock, I can't publish details, so Figure 8 only shows a symbolic cash balance. In reality, a Go program runs the control

Listing 4: dago.go

```
01 package main
02
03 import (
04   "bufio"
05   "golang.org/x/text/language"
06   "golang.org/x/text/message"
07   "os"
08   "os/user"
09   "path"
10   "regexp"
11   "strconv"
12 )
13
14 func mon() string {
15   usr, err := user.Current()
16   if err != nil {
17     panic(err)
18   }
19
20   logf := path.Join(usr.HomeDir, "data/monlog.txt")
21   file, err := os.Open(logf)
22   if err != nil {
23     panic(err)
24   }
25   defer file.Close()
26
27   scanner := bufio.NewScanner(file)
28   var lastLine string
29   for scanner.Scan() {
30     lastLine = scanner.Text()
31   }
32   if err := scanner.Err(); err != nil {
33     panic(err)
34   }
35
36   re := regexp.MustCompile(`\d+`)
37   match := re.FindString(lastLine)
38   n, err := strconv.ParseInt(match, 10, 64)
39   if err != nil {
40     panic(err)
41   }
42
43   n = n / 1000
44   p := message.NewPrinter(language.English)
45   return p.Sprintf("%d", n)
46 }
```

Listing 5: ulanzi.go

```
01 package main
02
03 import (
04   "fmt"
05   "time"
06 )
07
08 func main() {
09   // Youtube
10   f, v, err := youtubeStats()
11   if err != nil {
12     panic(err)
13   }
14   p := apiPayload{Text: fmt.Sprintf("%d/%d", f, v), Icon:
         974, Duration: 4, Rainbow: true}
15   err = postToAPI("youtube", p)
16   if err != nil {
17     panic(err)
18   }
19
20   // Countdown
21   loc, err := time.LoadLocation("America/Los_Angeles")
22   if err != nil {
23     panic(err)
24   }
25   timerVal := DHMUntil(time.Date(2024, time.August, 1, 0,
                0, 0, 0, loc))
26   p = apiPayload{Text: timerVal, Duration: 4, Rainbow: true}
27   err = postToAPI("countdown", p)
28   if err != nil {
29     panic(err)
30   }
31
32   // Dago
33   p = apiPayload{Text: mon(), Icon: 23003, Duration: 4,
         Rainbow: true}
34   err = postToAPI("dago", p)
35   if err != nil {
36     panic(err)
37   }
38 }
```

computer every day, evaluating all my cash deposits and investment values, adding them up, and attaching them to the end of a logfile as a numerical value. This means that the `mon()` function in Listing 4 only needs to navigate to the end of the logfile, extract the first numerical value, and return it to the caller to determine my total wealth.

Because the bytes of a file are stored sequentially on the hard disk and a line is implemented under Unix in such a way that there is a newline character at the end, reading the last line of a file is by no means trivial. The simplest method: Tell the program to read the bytes of the file line by line up to the next newline character until it reaches the end of the file; it then just needs to remember the content of the last line it processed.

However, this is very inefficient, especially with longer files, because reading out unnecessary data can take a long time. For greater efficiency, you can use the Unix `fseek()` function to tell the operating system to work its way to the end of the file without much delay and search backwards from there for the beginning of the last line. However, because the logfile processed by Listing 4 isn't excessively long, it uses the first, simpler method.

To make long numbers easier to read, the US and UK comma-separate groups of digits ("10,000"); some other countries, such as Germany, for example, use dots ("10.000") instead. The standard `text/message` Go library takes care of this in Listing 4, loading the `language` library in line 5 and initializing it for the English-language area in line 44. This means that the `mon()` function returns the correctly formatted string for the money store status to the main program.

Starting Signal

The main program in Listing 5 finally lumps it all together. It calls the helper functions of the three defined apps in sequence and sends the corresponding JSON data to the display each time. To compile the Go program, the three standard commands in Listing 6 process all five source files discussed so far and create the `ulanzi` binary. To keep the display up to date, a cron job on the control computer needs to call the binary at regular intervals (e.g., hourly). This requires a working WiFi connection to the display.

If Awtrix restarts, for example, because the device was unplugged and the battery is exhausted, like in the case of a prolonged power outage or following a manual restart due to a configuration change, the Ulanzi forgets the manually edited code and only plays the preconfigured apps (unless you disabled them in advance). Things stay this way until the next API command comes from the control computer setting the latest values for all custom apps. Then the cycle starts all over again for your viewing pleasure. ■■■

Listing 6: build.sh

```
$ go mod init ulanzi
$ go mod tidy
$ go build ulanzi.go api.go countdown.go youtube.go dago.go
```

Info

[1] Ulanzi TC001 on AliExpress: *https://www.aliexpress.us/item/3256804848125097.html*

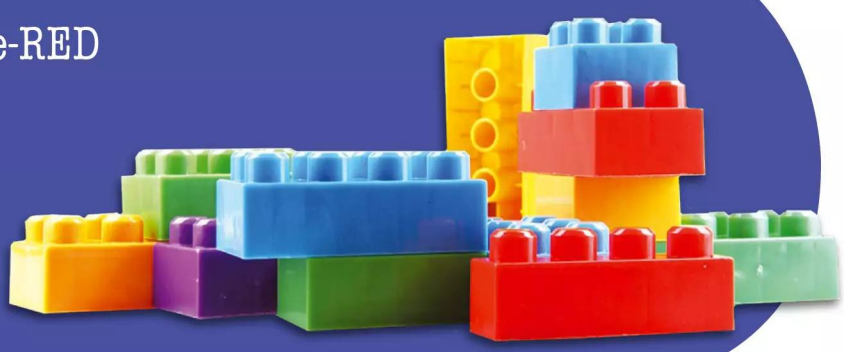
[2] Awtrix custom firmware for the Ulanzi TC001: *https://blueforcer.github.io/awtrix-light/#/*

[3] "Programming Snapshot: Process YouTube View Counts in Go" by Mike Schilli, *Linux Magazine*, issue 280, March 2024, pp. 44-49 *https://www.linux-magazine.com/Issues/2024/280/Stay-Tuned*

[4] LaMetric icons: *https://developer.lametric.com/icons*

Getting started with Node-RED

The LEGO Principle



Node-RED lets you connect ready-made code building blocks to create event-driven applications with little or no code writing. *By Udo Brandes*

Imagine you come home and the blinds come down in the living room, subtle lighting turns on, Alexa says hello, and the coffee maker prepares a cup of coffee. You can arrange all this magic at the push of a button without staying up all night writing code thanks to the low-code Node-RED platform [1].

With Node-RED, a few elements control the flow. Each element represents a whole block of instructions that would require a large number of lines of code in conventional programming (Figure 1).

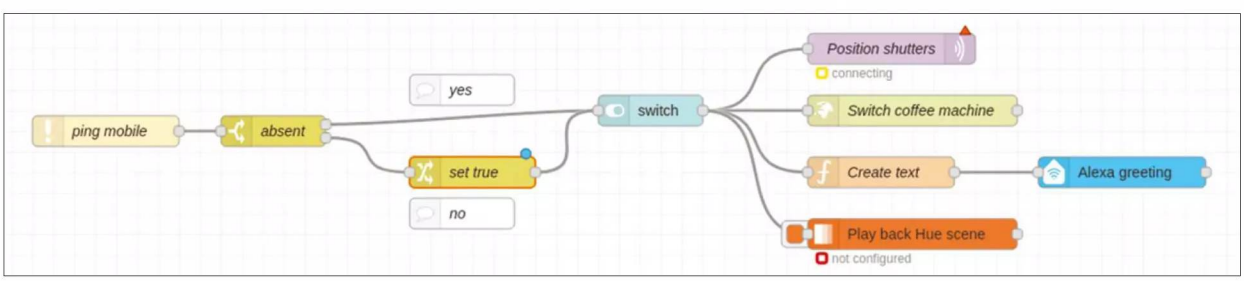
Every Node-RED flow follows the same programming principle. First, you need a trigger. In the example shown in Figure 1, the trigger evaluates a ping signal from a cell phone. You could also use geodata as the trigger by replacing the ping evaluation with a geofencing element. In either case, the event triggers the data processing, which includes queries and involves modeling data or, as an example, setting a dashboard display. All of this can be arranged visually, and it takes place – unlike many other home automation systems – in a unified user interface.

Open Source from the Start

Long considered an insider tip in the maker scene, Node-RED has become a popular and widely-used tool for writing simple programs and setting up complex applications, such as smart home systems, as well as a way to try something out to save time.

Still relatively young, Node-RED was developed in late 2013 by Nick O'Leary and Dave Conway-Jones, both scientists with the IBM Engineering Technologies Group, as an open source side project. The goal was to provide a simple way to quickly connect devices with web services and programming interfaces.

Node-RED is based on the Node.js programming language, which is why you find Node in the name. Dave



Lead Image © nenovbrothers, 123RF.com

Figure 1: This Node-RED flow controls a sequence of events to welcome you home at the end of the day.

Listing 1: Starting the Script

```
bash <(curl -sL https://raw.githubusercontent.com/node-red/
  linux-installers/master/deb/update-nodejs-and-nodered)
```

Conway-Jones suggested the RED bit because it sounds like Code Red, an alarm code as well as a family of computer viruses. Node-RED has gained popularity in a short period of time and is now a widely used tool for the Internet of Things (IoT) and industrial application prototyping.

The Basic Principle

Node-RED is a platform with a steadily growing user base and an active developer community that continuously contributes new nodes. The range of applications for Node-RED is broad and still growing, with the ability to use social media channels (X, formerly known as Twitter; email) as well as databases (MariaDB or InfluxDB). In addition to simple tasks (ad-hoc evaluations, prototyping), Node-RED can also be used to solve complex problems in an elegant way. However, not all tasks are equally well suited for Node-RED. For example, Node-RED has difficulties with loops, and processing large tables is often better off left to traditional programming languages.

Basically, Node-RED provides predefined code blocks for executing tasks. These code blocks are referred to as nodes. Nodes can be connected and networked with each other. The data processing starts in the input nodes, which are followed by the processing nodes and the output nodes. The complete structure is known as the flow.

Flow-based programming is based on an idea by J. Paul Morrison. Back in the early 1970s, Morrison considered an application to be more than just a single, sequential process with a start, a multitude of processing steps, and an end after processing the tasks. Instead, he was interested in a network of asynchronous processes that communicate via streams of structured data blocks (information packets).

Installation

You can host Node-RED on any operating system or even in the cloud. In my examples in this article, I use a Raspberry Pi, which is popular due to its frugal energy requirements. To get started, open a terminal and update the system with

```
sudo apt-get update
```

On the Raspberry Pi, Node-RED provides a script for installation and updates, which you start with the code in Listing 1.

The script installs Node.js, npm, and Node-RED. It works on all Debian-based Linux systems, such as Ubuntu. However, you may find the current Node-RED v3.0.2 requires a newer Node.js than found in the latest Ubuntu LTS v22.04. In this case, you have several options: You can install a recent Node.js from the source code, download the installer script and tell it to install an older Node-RED when called, install the Snap version, or switch to Docker. Regardless, make sure that all the required packages are in place with

```
sudo apt install build-essential git curl
```

After the install, you can typically launch Node-RED either at the terminal with

```
node-red-start
```

or as a system process at boot time with

```
sudo systemctl enable nodered.service
```

Essential Files

All of the important Node-RED files reside in `~/.node-red`, a hidden folder in your home directory. The most important files are in the `node_modules` folder, which contains the installed nodes, and the `flows.json` file, which contains all the flows in JSON format. Backing up `flows.json` saves all of your flows. The `settings.js` file defines the Node-RED configuration and is where you set up important things, such as the password security level and encrypted connection features (HTTPS). The length of the debug output (1,000 characters by default) or the logging level can also be defined in `settings.js`.

Node-RED Editor

The Node-RED editor is the core element for development with Node-RED. Its strengths lie in visual programming, a powerful debugger, and a linter (a tool for checking the code for problems). In addition, some system administration functions can be called in the editor. The Node-RED developers have implemented many new features in recent years, making programming with Node-RED even more intuitive.

The Node-RED editor runs in a browser. In a local installation, the Node-RED server listens on port 1880 by default. In a browser running on the same host as the Node-RED server (such as Chromium on the Raspberry Pi), the URL is *http://localhost:1880/*. Be careful: The Node-RED installation process leaves you with an unprotected system! Securing access with a username and password is highly recommended. A secure HTTPS connection is

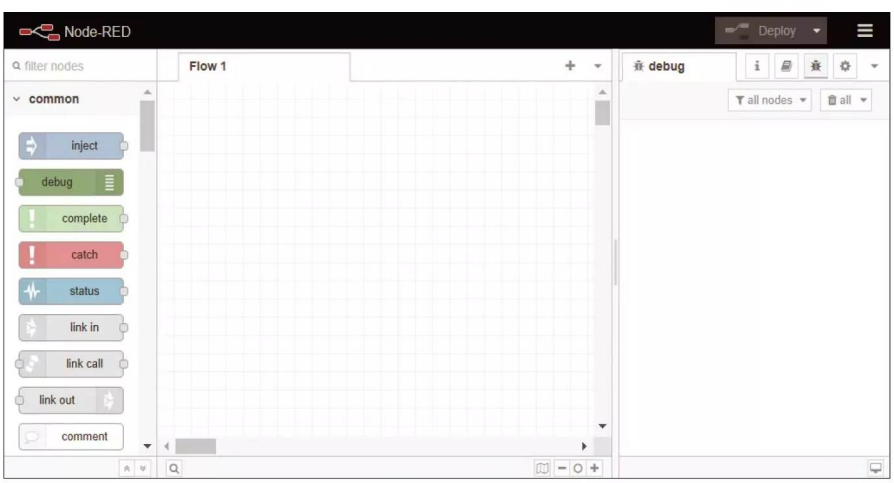


Figure 2: The Node-RED editor in a browser window.

also essential if you want to support access from outside the local network.

The Node-RED editor's browser window (Figure 2) is divided into four sections:

1. The header (at the top) contains the *Deploy* button for activating flows and a menu icon. From the menu icon, you can access various system settings, as well as some helpful functions that go beyond plain-vanilla editing (e.g., importing or exporting flows).
2. The node palette (on the left) contains the deployable nodes. The nodes are arranged by topic, with the ability to expand or collapse individual topics.
3. The workspace (in the middle) is where flow development takes place. It contains a management bar for the flows, the flow design area, and a footer.
4. The sidebar (on the right) provides useful information (e.g., help, debug) and management options (e.g., configuration nodes, which are nodes for settings such as an email account's user data). The nodes in the sidebar are divided into categories and can be opened by pressing buttons.

Message Strategy

Following in the tradition of using a Hello World program to explain data structures, language elements, and relationships in a given program language, my first Node-RED example uses a Hello World flow consisting of an Inject node that executes a program flow and a Debug node that displays the results (Figure 3).

Flows can be created by dragging nodes from the node palette into the workspace's flow design area. Each node has at least one input or output port, symbolized by the small gray squares. Some nodes also have multiple output ports (e.g., one port for normal data output and another for error messages), but a node can have only one input port. Individual nodes are connected by wires, which you create by dragging a connection from port to port while holding down the left mouse button.

Above the nodes, symbols provide information about the processing status. A blue dot indicates that changes have not yet been applied, while a red triangle indicates that configurations still need to be made. Information can also appear below the node in the node decoration (e.g., a Switch node's status). In most cases, you will need to modify the nodes you use. To do this, double-click on the node to modify the node properties (Figure 4).

The Properties dialog is roughly the same for all nodes; the difference is in the specific properties, which can be extensive in some cases. In my example, I want the Inject node to inject a message where `msg.payload` equals "Hello World" and `msg.topic` equals `first flow`. After accepting the changes (using the *Deploy* button in the header), clicking on the larger gray square to the left of the Inject node starts the flow. It sends a message to the Debug node. The debug output in Figure 5 shows the entire message object.

You can clearly see the message format: a JSON object. JSON in particular meets modern requirements for object-oriented programming with regard to the exchange of data between one or more systems. Pushbullet, Telegram, the Hue lighting system, and many other programs use JSON. It builds on a structure with a name-value pair (`"topic": "first flow"`) and on an ordered list (table) of values (`"colors": ["blue", "red", "yellow"]`).

Each message consists of at least one message ID (`_msgid`) and the message body (`payload`). The message ID remains the same throughout the entire message processing process, even if

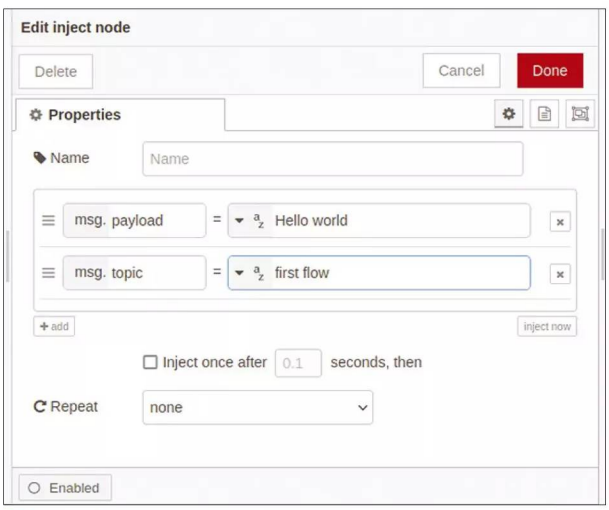


Figure 4: When you double-click on a node, the node Properties dialog box opens to let you make modifications.

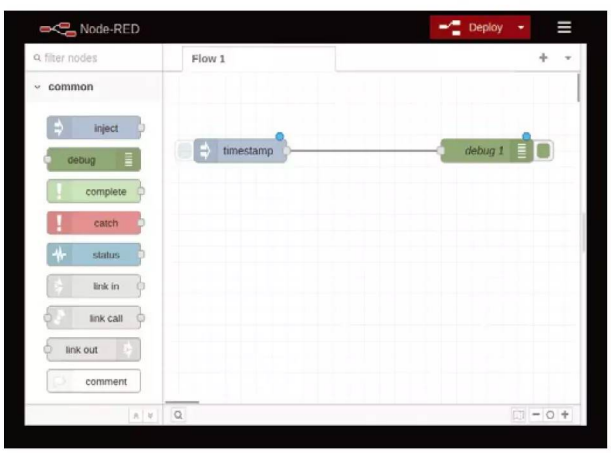


Figure 3: A simple Hello World flow in the Node-RED editor.

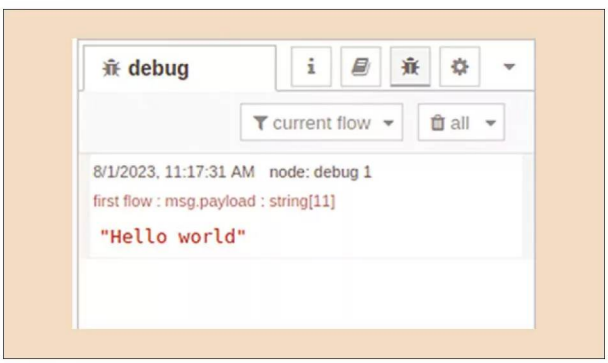


Figure 5: The message object's debug output for the Hello World example.

the message flow branches. The message body contains the message text in JSON format. Deep nesting is not uncommon. The debug output helps to keep track of the object by collapsing or expanding the object's individual components. Besides this, objects such as `"topic"` can be added to the first object level of the message – the level then depends on the circumstances.

You'll also find the JSON format in the `settings.js` configuration file and as a format for storing flows, as shown in Listing 2 in the Hello World flow.

Processing Data

My next not-too-complex example demonstrates how easily and powerfully Node-RED supports programming. This example feeds data from a soil moisture sensor via USB to Node-RED. Of course, you can change the trigger element to suit your needs. In other use cases, the trigger could just as easily be incoming mail, Telegram messages, or TCP requests. The same applies to the results of data processing (debug output, monitoring on a dashboard, or storing data in a file or database). The principle is always the same.

As the first step, I need to measure soil moisture with a capacitive soil moisture sensor (Figure 6). An Arduino Nano converts the analog measured value (voltage) into a numerical value and outputs the value via the serial interface every 10 seconds. The Arduino code only consists of a few lines (Listing 3).

Node-RED has a special node for receiving data from a local serial port: the Serial-In node from the *Network* section of the node palette. It replaces the Inject node from the Hello World example. The red triangle above the Serial-In node (Figure 7) indicates that general configuration settings need to be made in addition to configuring the node properties. Double-click to open the node properties window. You need to specify the port for the serial interface. The magnifying glass symbol (located at the bottom of the workspace) makes it easier for Node-RED to find available ports and to specify syntactically correct parameters. You can select the correct

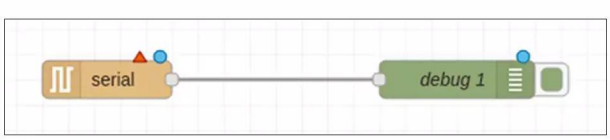


Figure 7: A serial port is connected to a Debug node.

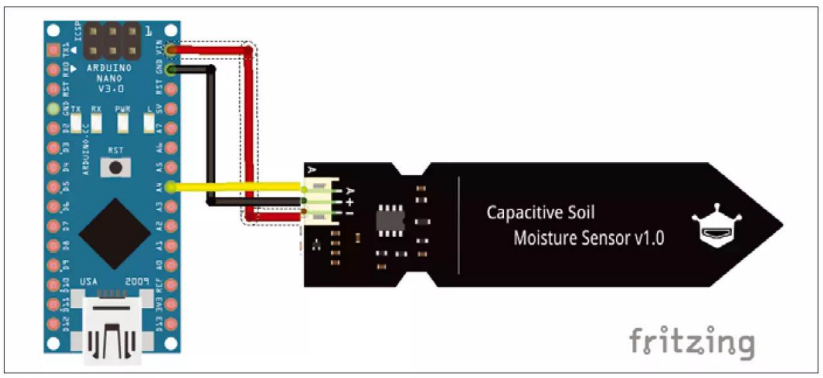


Figure 6: Using an Arduino microcontroller, a capacitive sensor measures soil moisture.

Listing 2: Flow Saved in JSON Format

```
[
  {
    "id": "b4d449fdd05e5ebd",
    "type": "tab",
    "label": "Flow 1",
    "disabled": false,
    "info": "",
    "env": []
  },
  {
    "id": "4ef230ee3cdeb12d",
    "type": "inject",
    "z": "b4d449fdd05e5ebd",
    "name": "Hello World",
    "props": [
      {
        "p": "payload"
      },
      {
        "p": "topic",
        "vt": "str"
      }
    ],
    "repeat": "",
    "crontab": "",
    "once": false,
    "onceDelay": 0.1,
    "topic": "first flow",
    "payload": "Hello World",
    "payloadType": "str",
    "x": 100,
    "y": 40,
    "wires": [
      [
        "1aba6a2edc504015"
      ]
    ]
  },
  {
    "id": "1aba6a2edc504015",
    "type": "debug",
    "z": "b4d449fdd05e5ebd",
    "name": "debug 1",
    "active": true,
    "tosidebar": true,
    "console": false,
    "tostatus": false,
    "complete": "true",
    "targetType": "full",
    "statusVal": "",
    "statusType": "auto",
    "x": 300,
    "y": 40,
    "wires": []
  }
]
```

Listing 3: Arduino Code

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  Serial.println (map(analogRead(4), 0, 1023, 0, 100));
  delay(5000);
}
```

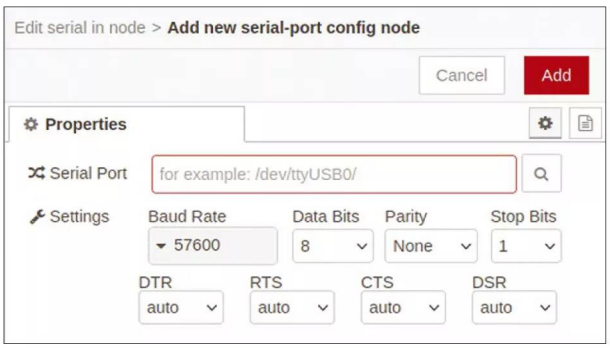


Figure 8: Properties such as the baud rate need to be set for the serial port.

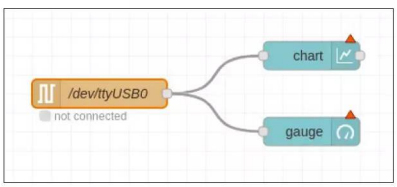


Figure 9: The measured values received via the serial port are forwarded for display as a chart or gauge.

baud rate (9600 in my example) from a drop-down list (Figure 8).

After the flow has been accepted, the received values can be taken from the debug output. The entire message object has three name-value pairs: `payload`, `port`, and `_msgid`. The content of `payload` includes the carriage return line feed character (`\n`) in addition to the numeric value; it is used to split up the data stream received via the serial interface. Other splitting options are by fixed length or after a time interval.

Dashboard

Because visualizing the measured values in a Debug node is not very user friendly for long-term operation, Node-RED offers graphical user interface (GUI) packages. A search for "Dashboard" in the Palette Manager (*Main Menu | Manage Palette | Installation*) yields more than 40 hits. While no package can currently compete with Grafana, these GUIs offer sufficient performance for most use cases. For instance, the *uibuilder* package gives users the ability to create web interfaces dynamically. For this example, I will be using the *node-red-dashboard* package, which includes nodes for typical use cases and is the most widespread, solid, and mature package.

You can use *node-red-dashboard* to create a very versatile dashboard. It can have multiple tabs to organize information by topic. A tab holds one or more vertical columns (groups) that, in turn, store the widgets. The default column width is six squares, and the default size of a square is 48x48 pixels. All this can be changed, as can the dashboard theme. Applying this to the soil moisture sensor example, the measured data can be visualized in the form of a line diagram or a pointer gauge like a speedometer. Nodes for this then replace the Debug node (Figure 9).

The red triangles again indicate that some configuration work is required (assigning the widget to a tab or group that may still need to be created), and you need to define the node properties again. Figure 10 shows how the incoming measured values are displayed without making use of the extensive configuration options of the Chart or Gauge nodes.

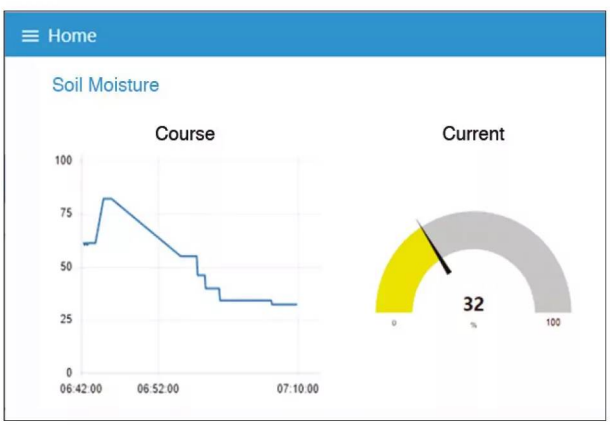


Figure 10: The Chart and Gauge nodes are used on a single dashboard here.

Creating Flows

Node-RED can help you design program-controlled processes. Usually, programming means writing smaller or larger amounts of code, which often becomes unmanageable unless you exercise self-discipline. Depending on the choice of language, in-depth programming knowledge is required. If you want to test the code, you have to repeatedly compile and bind it to create an executable module; this can require constant switching between different windows (editor, terminal). Node-RED simplifies this procedure.

As another example of Node-RED's capabilities, I will activate an irrigation system via a switch (a Shelly relay in my example) if the soil is too dry. This results in different blocks in the program flow:

- Receiving the measured values
- Querying for a threshold value that is used to decide whether the soil is too dry
- Triggering the irrigation (switching on)
- Switching off the irrigation after a certain time

Basically, Node-RED solves this problem with just four nodes. The final Debug node is for control purposes only and has nothing to do with the core task (Figure 11). The measured values are transferred to the system in the usual way via the USB interface. A Switch node checks whether a defined threshold value (an arbitrary value of `400` in my example) is exceeded. In other words, it acts like an `if` query.

In most cases, the query references a message object that returns a number or a string as a comparison value. This does not work in this example because `msg.payload` contains the CR-LF characters in addition to the measured value. This is an essential

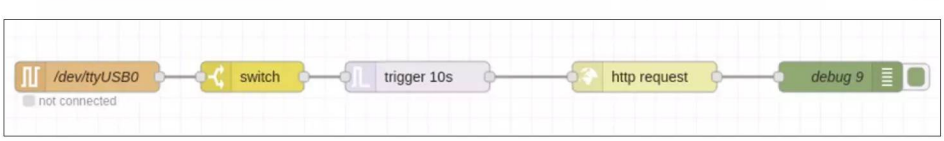


Figure 11: Four nodes form the complete irrigation control (the final Debug node is for control purposes).

requirement if you want to identify the transmitted data packets individually. Thus, the object to be compared needs to be modified using a JSONata expression (Figure 12), which is a lightweight query and transformation language for JSON data that can be used to apply simple functions directly to values. The `$substring` function is used to extract all the digits from `msg.payload` starting at the first position. The number of digits is the length of `msg.payload` minus the two characters for `CR` and `LF`. The Trigger node handles switching on and switching off the irrigation system.

When a (below threshold) message is received, the node sends a JSON message to turn on the irrigation system. It then waits 15 seconds (i.e., slightly longer than the time at which the next measured value arrives) and then sends the switch-off message. The highlight is that the output of the `Off` command is held back if further messages arrive. Irrigation then continues until the soil has a sufficient moisture content (Figure 13).

Shelly devices can easily be controlled using HTTP commands. This task is handled by an HTTP Request node at this point. The HTTP Request node's `GET` method references the URL by which the Shelly device is accessible on the local network. The action results from the `msg.payload` object. The HTTP Request node converts this object and adds the result to the URL as a query string parameter. Figure 14 shows the debug output after a power-on operation. All in all, the task can be solved quickly and compactly with Node-RED.

Programming Functions

Node-RED has a wide range of versatile nodes that cover most requirements. However, situations can arise where the default nodes are not up to the task. The Function node is often the method of choice in this case. The Function node uses JavaScript to process messages. JavaScript is a powerful programming language and can delve deeply into the system. For this reason, Node-RED needs to be secured, especially if you allow access from outside your local network.

The Function node's role as an all-rounder is already clear from its node properties. There is a separate option for:

- setting the number of output ports in the `SETUP`;
- programming the code that will be executed when the node is deployed in `START`;
- saving the code for each message input in `FUNCTION`; and
- storing the code that will be executed when Node-RED stops, or before the node is deployed again, in `STOP`.

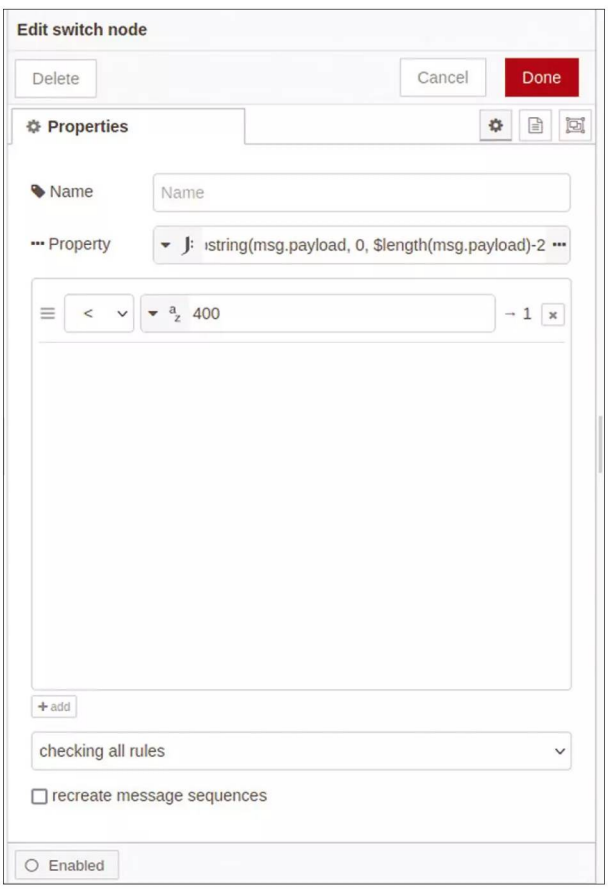


Figure 12: The numerical measured value is extracted via a `substr` function.

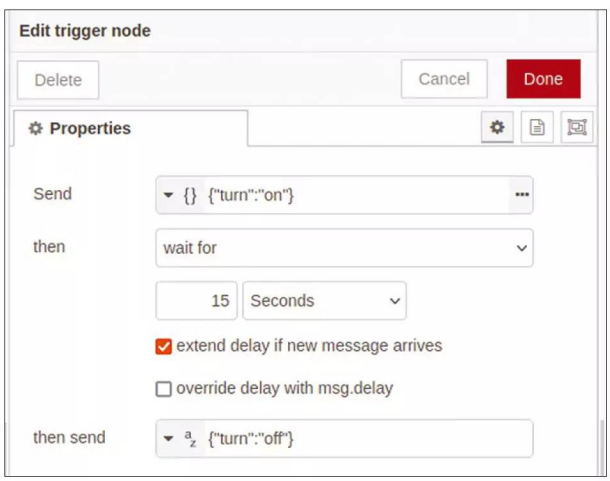


Figure 13: The properties of the Trigger node define the irrigation time.

Figure 14: The debug output following the HTTP Request node shows how to turn on the irrigation system.

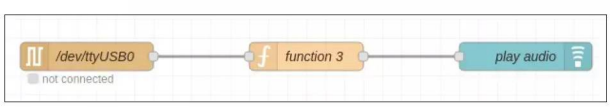


Figure 15: A Function node controls the voice output.

To demonstrate this principle, my example uses voice output (Figure 15) to inform the user of the soil moisture status once a minute. The Function node prepares the announcement. Listing 4 shows the code to be stored in the Function node for this task.

The variable `value` stores the content of `msg.payload` (i.e., the measured value). Then `msg.payload` is rebuilt. The contents depend on the measured value (set arbitrarily here). The variable `sec` contains the seconds part of the current time. Because a new value only arrives once every 10 seconds, the action of forwarding the newly created message is based on a time period and not on an exact value in seconds. The Play Audio node, which is also available for the dashboard, completes the flow with a voice output. It actually expects `msg.payload` to contain a buffer with a WAV file. If the browser provides native support for text-to-speech, `msg.payload` can (like in my example) also contain a string that is read aloud. You can choose from various female and male voices.

Listing 4: Controlling Voice Output

```
let value = msg.payload
let now = new Date();
let sek = now.getSeconds();
msg.payload = 'The soil moisture has a value of ' + value
if (value < '400') {
  msg.payload = msg.payload + "  Irrigation urgently required";
} else {
  msg.payload = msg.payload + " Everything is fine";
}
if (sec < 10) {
  return msg;
}
```

Conclusions

Node-RED is far from being just a click-and-enjoy tool. Instead, it can provide elegant solutions to complex and multi-layered tasks even in the industrial sector. It impresses with its variety of available building blocks (nodes), the clear (visual) representation of the program flow, and the intuitive user interface. The dashboard option is also of great value. ■■■

Info

[1] Node-RED: *https://nodered.org*

Author

Udo Brandes has worked as a programmer and in-house consultant for various public service providers. Today he works as an author with a focus on the Internet of Things/microcontrollers and has written a book, *Node-RED* (only in German), a comprehensive guide published by Rheinwerk-Verlag that covers Node-RED's many capabilities with a large number of hands-on examples.

Discover the past and invest in a new year of IT solutions at Linux New Media's online store.

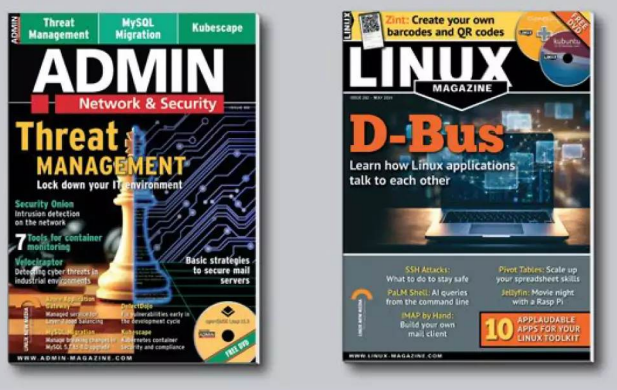
Want to subscribe?

Searching for that back issue you really wish you'd picked up at the newsstand?

shop.linuxnewmedia.com

shop.linuxnewmedia.com

DIGITAL & PRINT SUBSCRIPTIONS



SPECIAL EDITIONS

Mix low-code Node-RED with Python

Snake Senses

Adding Python to your Node-RED arsenal lets you create easy Raspberry Pi robotic and IoT projects. *By Pete Metcalfe*

If you want to build some fun Pi projects but are still working on your Python skills, then mixing low-code Node-RED with Python might be an option for you. As we've discussed in the previous article, Node-RED [1] is a low-code drag-and-drop interface that is extremely powerful for the creation of Raspberry Pi robotic and Internet of Things (IoT) projects.

Node-RED's custom scripting is JavaScript; however, you can also use Python, which offers a platform to play and learn Python basics for high-level tasks such as scheduling and web dashboards while taking advantage of Node-RED's low-code interface.

In many cases, Raspberry Pi features are only available in Python, so even die-hard Node-RED users could benefit from knowing how to integrate Python into their projects. In this article, I look at two examples that mix Python and Node-RED. The first creates a web dashboard to drive a Raspberry Pi rover; the entire project only requires two Node-RED widgets. The second project creates an IoT page that shows temperature and humidity data from a BME280 sensor.

Getting Started

Depending on your Raspberry Pi image, Node-RED may already be installed. If not, see the Node-RED documentation [2] or the previous article for custom installation directions.

Some excellent dashboard components can be used to create lightweight web interfaces. A great widget to include in your toolset is the *Button State* flow for creating an array of buttons. To install this component select the Node-RED *Menu* | *Manage Palette* item, click the *Install* tab, and search for *ui-button* (Figure 1).

The next important step is to add a Python-enabled widget. Among the various choices, I chose the *python-function-ps* component (Figure 2) because it was recently updated; however, the other choices worked on my test projects, as well. The ability to use Python instead of JavaScript in Node-RED is an extremely useful feature; however, it's not bulletproof, so some care may be needed when you're using advanced Python

Figure 1: Add a button array into Node-RED.

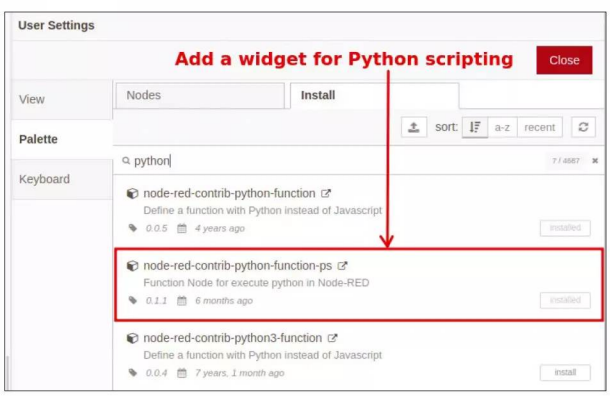


Figure 2: Add Python scripting into Node-RED.

Photo by David Clode on Unsplash

libraries. In the next section, I use these two widgets to control a Raspberry Pi rover.

Raspberry Pi Rover

Many approaches that use a Raspberry Pi can lead to a car or rover. For this project I used:

- A two-motor car chassis (~ $15)
- A portable battery (5V, 3A output, ~ $30)
- A Raspberry Pi with a motor shield
- Four alligator clips and four jumper wires
- Elastic bands and duct tape

For this project I wanted to ensure that Python scripting with Node-RED could be used on a variety of Pi models. I tested on a 1 B+, 3, and 4. The PI 1/2 are old and slow but they have the advantage of lower power. For a Raspberry Pi 3 and 4, the portable battery needs to output 3A. If you are using a Pi 1 or 2 you can use a standard 2.1A phone charger.

Because of the power draw, connecting motors directly to a Raspberry Pi is not recommended; luckily, some good motor or automation shields are available for around $25. If you're feeling adventurous, you can build your own motor shield with a L293D chip (16-pin motor driver integrated circuit) for about $4. On this project, I used an older PiFace Digital module, which has good Python support but weak Node-RED functionality.

The two-motor car chassis usually comes without any wiring on the motors. For a quick setup, I use a combination of alligator clips and jumper wires to connect the motor terminals to the Pi motor shield. A couple of strips of duct tape are useful for holding the wires in place. Finally, elastic bands keep the portable battery and the Raspberry Pi attached to the chassis.

To test the hardware setup, I found it best to keep the car chassis raised with the wheels off the ground. This step allowed me to use a standard power plug without killing the battery before I was ready to play. You might have to adjust the wiring to ensure the motors are both turning in the required direction.

The first software step is to install your motor's Python library. (Note: This step will vary depending on your motor shield.) For my hardware, I installed the PiFace library with:

```
pip install pifaceio
```

At this point, you should test the hardware directly with Python. Check your hardware documentation for some sample code to turn the motor on and off.

To test a single motor with Python within Node-RED, four flows are used: two *inject*, one *python-function-ps*, and one *debug* (Figure 3). A *debug* flow isn't required, but it's useful to verify that the Python code runs cleanly. The *inject* flows create a message payload with either a numeric 0 or 1 to stop or start the motor.

In the *python-function-ps* flow, the incoming Node-RED message (`msg`) is accessed as a Python dictionary variable. The following Python examples read, set, and clear the Node-RED message:

```
# get the message payload
themsg = msg['payload']
# set the payload
msg['payload'] = "Good Status"
# create a new message item
msg['temperature'] = 23.5
# clear the entire message
msg.clear()
```

For the PiFace library, my code needed to do a `write_pin` command to set a specific pin. A `write` command then outputs the request states for all the pins. To set pin 0 to the incoming payload message, use:

```
pin = 0
# Pass the msg payload as the
  pin state
pf.write_pin(pin,msg["payload"])
pf.write()
```

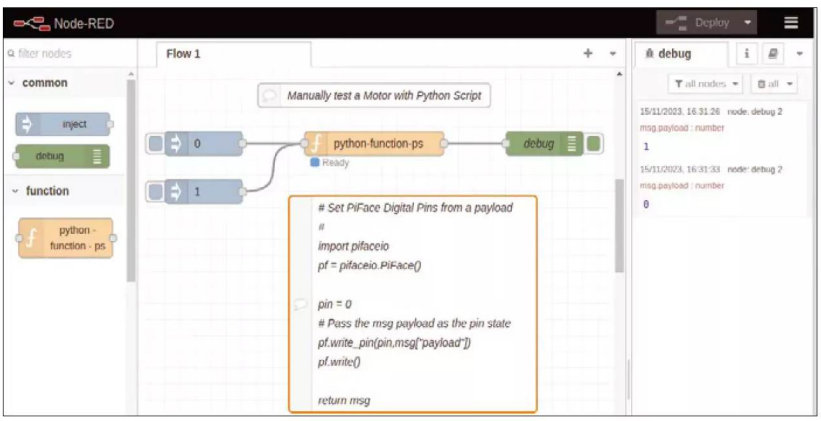


Figure 3: Node-RED test logic to control a motor.

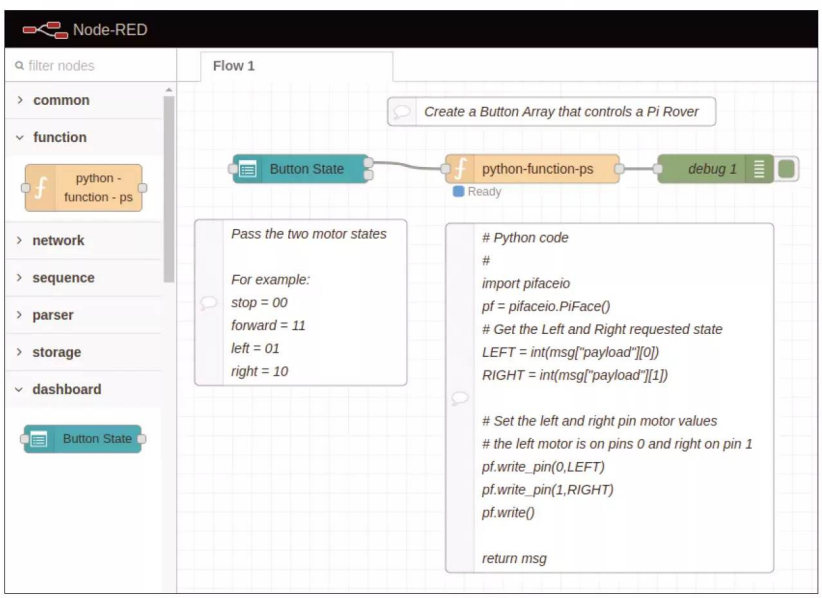


Figure 4: Node-RED logic to control a Raspberry Pi rover.

Listing 1: Python Control Code

```
01 #
02 # Set PiFace Digital Pins
03 #
04 import pifaceio
05 pf = pifaceio.PiFace()
06
07 # Get the Left and Right requested state
08 LEFT = int(msg["payload"][0])
09 RIGHT = int(msg["payload"][1])
10
11 # Set the left and right pin motor values
12 #   the left motor is on pins 0 and right is on pin 1
13 pf.write_pin(0,LEFT)
14 pf.write_pin(1,RIGHT)
15 pf.write()
16
17 return msg
```

Once the basic testing is complete, the next step is to define a Node-RED dashboard with buttons to control the rover.

The final Node-RED logic for this project only requires two widgets: The *Button State* component creates an array of user buttons, and *python-function-ps* runs the Python code to control the motors (Figure 4).

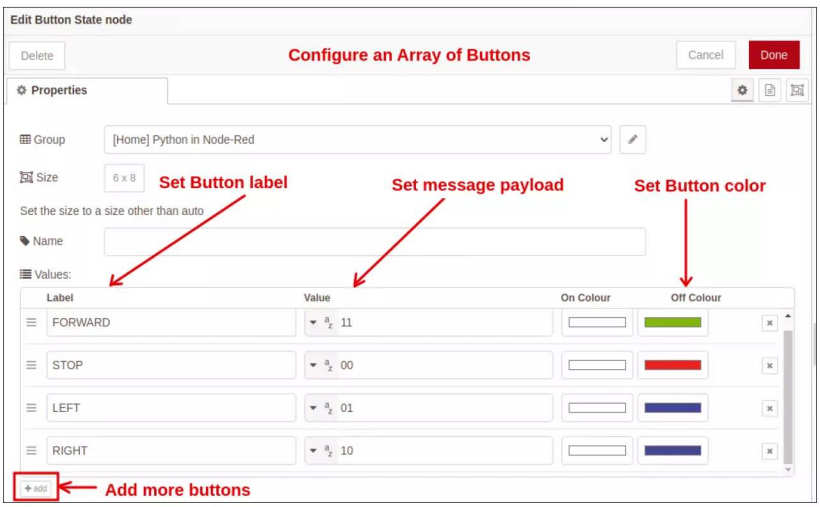


Figure 5: Configure a Node-RED button array.

The *Button State* widget is edited with a double-click. Multiple buttons can be added with custom labels, payloads, and colors (Figure 5). A simple two-character string is used for the buttons' message payloads, with the first character being the *LEFT* motor state, and the second being the *RIGHT* motor state. A *FORWARD* command sets both the *LEFT* and *RIGHT* motors to *1*, with a payload of *11*. A *STOP* command sets both motors off with a *00* payload. It's important to note, that to turn left, the left motor needs to be turned off and the right motor needs to run – and vice versa for turning right.

The *python-function-ps* flow (Listing 1) imports the Python *pifaceio* library (line 4) and creates a `pf` object (line 5). Next, the button payload passed in is parsed to make two variables: the `LEFT` and `RIGHT` requested motor state (lines 8 and 9). Lines 13-15 write the motor states.

Figure 6 shows the Node-RED dashboard and the rover with a PiFace Digital module mounted on a Pi 1. Future enhancements to this project could take advantage of motor shields that support reverse motor directions or variable-speed motor settings.

Sensor Project

You have the choice of an excellent selection of Raspberry Pi Python starter projects, but communicating with sensors and I/O are usually good places to start for people interesting in building IoT projects.

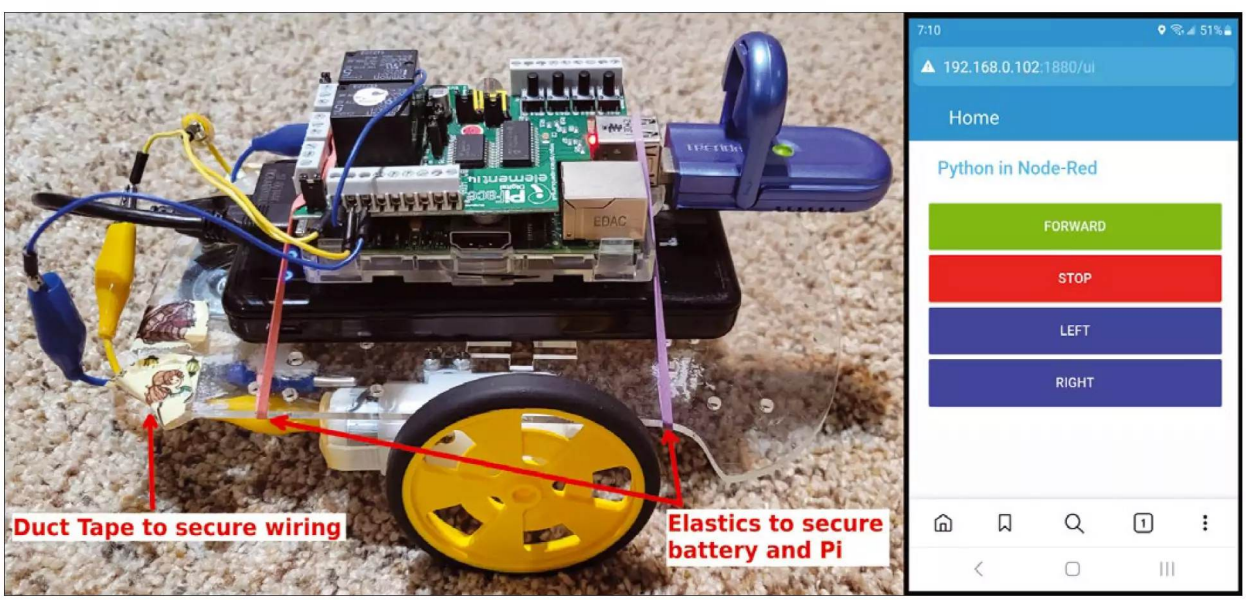


Figure 6: Pi Rover with Node-RED dashboard.

In this second project, I look at acquiring temperature and humidity data from a BME280 sensor (~$5); however, if you have a different sensor, you should be able to adapt this project to your needs. For the programming, you gather the sensor data in Python, and the real-time scheduling and the web dashboard are created in Node-RED. The BME280 sensor is connected to the Pi over inter-integrated circuit (I2C) connections. The serial data (SDA) and serial clock (SCL) are on Raspberry Pi pins 3 and 5 (Figure 7).

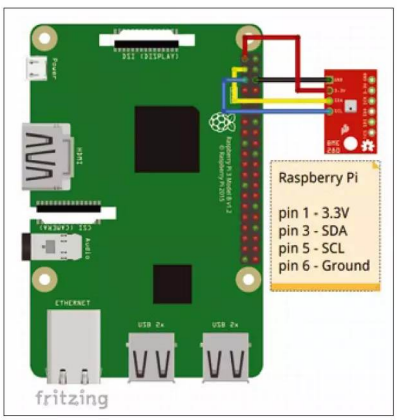


Figure 7: Pi wiring for a BME280 sensor.

Listing 2: Test BME280 Sensor

```
01 # bme_test.py - Show values from a
     BME280 sensor
02 #
03 import smbus2
04 import bme280
05
06 # BME280 sensor address (default
     address could be:  0x76)
07 address = 0x77
08
09 # Initialize I2C bus
10 bus = smbus2.SMBus(1)
11
12 # Load calibration parameters
13 calibration_params = bme280.load_
     calibration_params(bus, address)
14
15 # Get sampled data
16 data = bme280.sample(bus, address,
     calibration_params)
17
18 print("Temperature: ", data.
     temperature)
19 print("Pressure: ", data.pressure)
20 print("Humidity: ", data.humidity)
```

The first step in this project is to enable I2C communications and then install a Python BME280 library:

```
# Enable I2C, 0 = enable, 1=disable
sudo raspi-config nonint do_i2c 0
# Install Python BME280 library
pip install RPI.BME280
```

BME280 sensors are typically on addresses 0x76 or 0x77. To verify the address, use the `i2cdetect` command-line tool:

```
# Scan for I2C devices
$ i2cdetect -y 1
```

To ensure that the sensor, I2C communications, and Python library are all working, create a Python test program (Listing 2). If everything is hooked up and working correctly, some values should appear:

```
# Check BME280 setup with a Python
  test app
#
$ python3 bme_test.py
Temperature:  20.943249713495607
Pressure:  996.5068353240587
Humidity:  52.84257199879564
```

This Python code can be moved and tested in the Node-RED environment with *inject* and *debug* flows (Figure 8). A slight modification to the code in Figure 8 (lines 17-21) passes the sensor results to the dictionary `msg` variable instead of doing a `print` statement as in Listing 2. The *debug* flow is defined

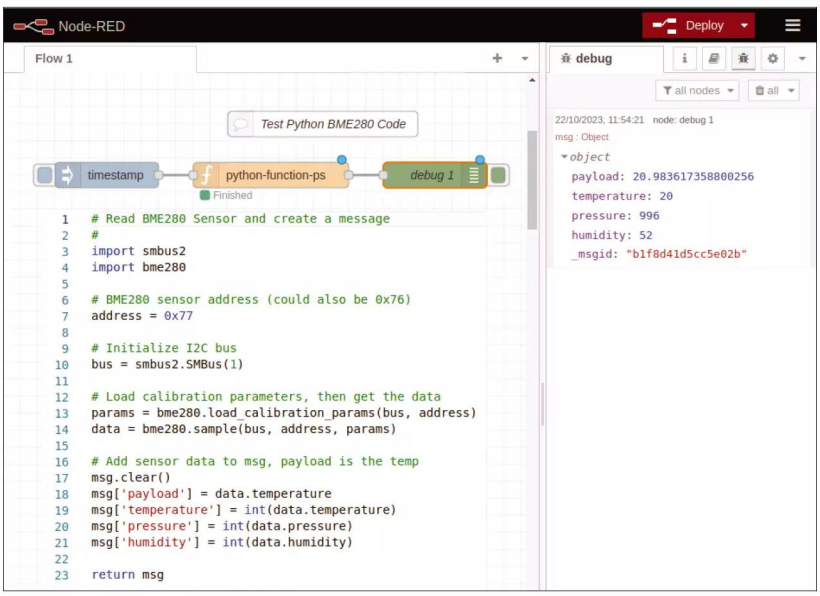


Figure 8: Node-RED BME280 test logic.

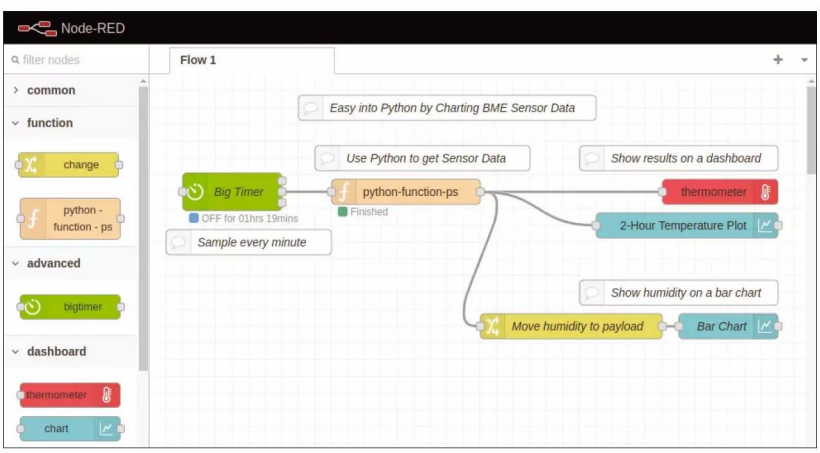


Figure 9: Chart BME sensor data.

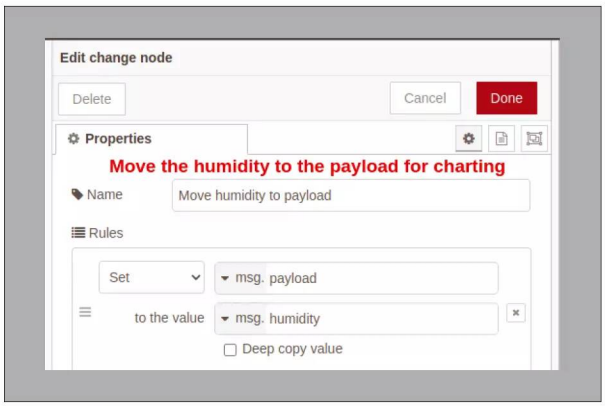


Figure 10: Move the humidity message item to the payload.

to show the complete message, so the debug pane shows all the sensor results.

The next step is to show the results in a web dashboard, which includes the addition of two new widgets. The first new addition is an old-style mercury thermometer widget (*ui-widget-thermometer*), and the second is a scheduler (*big-timer*). Note that it might be useful to include the Node-RED BME280 component for a comparison check.

The final application (Figure 9) uses the same Python code as in the earlier test circuit, but a *big-timer* widget schedules its execution. Although this widget has excellent scheduling functionality, to keep things simple, you can just use the widget's middle output to send a pulse every minute.

The *thermometer* flow shows the temperature value, which is the payload message from the Python code. A *chart* widget reads the same temperature value and presents the results in a two-hour line plot. A *change* flow moves the humidity to the message payload (Figure 10), which allows a second *chart* to show the humidity in a bar chart.

Figure 11 shows the Node-RED BME280 dashboard with the Raspberry Pi and sensor setup. This project could be enhanced to show the results from multiple sensors.

Summary

Python scripting in Node-RED offers new programmers a great way to build some interesting applications without getting too bogged down in graphical interfaces. In a few cases, the *python-function-ps* widget crashed. For me, this occurred with hardware-specific libraries like *pyusb*. A good workaround is to use the built-in *exec* component, which can run an external Python program. The *exec* widget supports appending message payloads to the called program string. ■■■

Info

[1] Node-RED: *https://nodered.org/*

[2] Installing Node-RED: *https://nodered.org/docs/getting-started/*

Author

For more of Pete's projects, see: *https://funprojects.blog*.

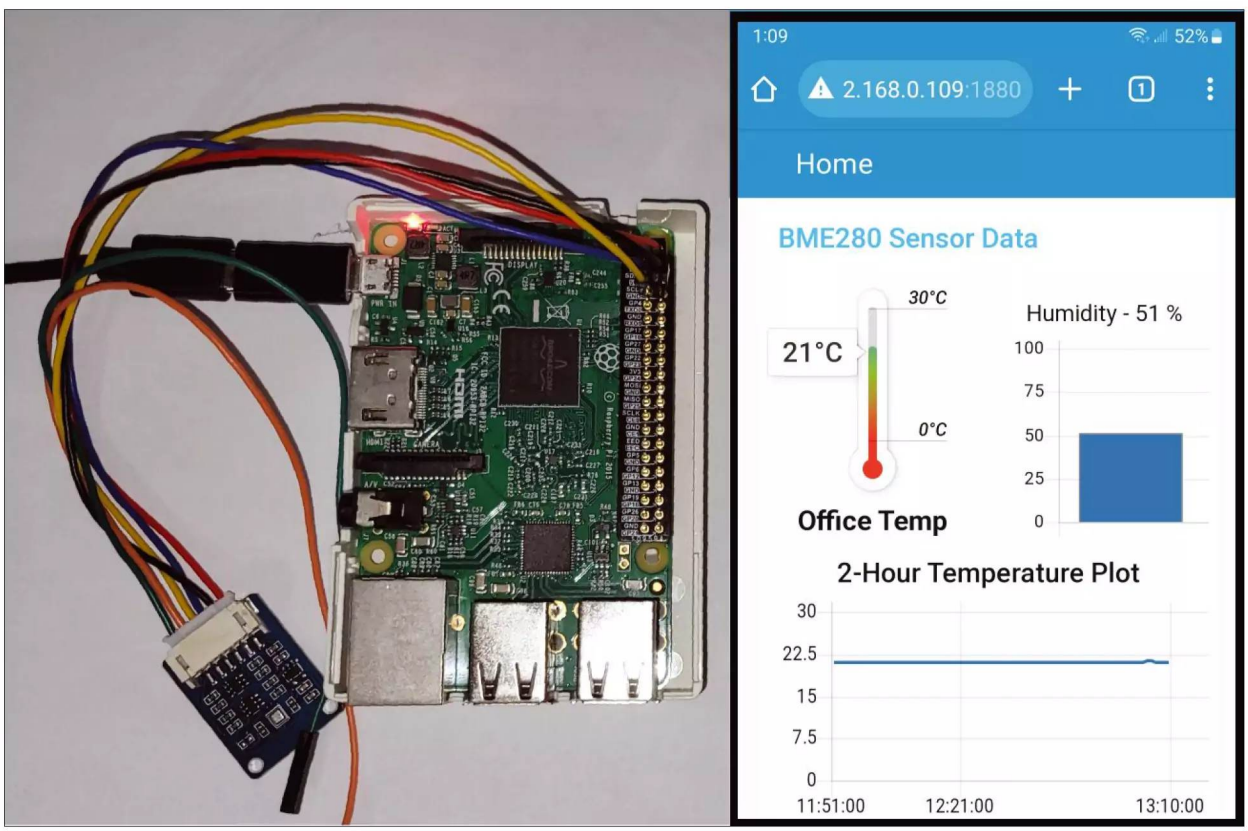


Figure 11: Raspberry Pi with BME280 sensor and the Node-RED dashboard.

Low-code programming for the Arduino with Snap4Arduino

Cold Snap!

Snap4Arduino brings the power of low-code programming to the Arduino hardware environment. *By Pete Metcalfe*

Scratch [1], from MIT, is a graphical coding environment that was originally designed for young programmers. Scratch is bundled with many Raspberry Pi releases, and it lets you create digital stories, games, and animations that communicate with the Pi's General Purpose Input/Output (GPIO) pins. There also some options for using Scratch with Arduino modules, however most of these implementations are somewhat limiting.

Snap! [2], which was created at the University of California, Berkeley, is an extended implementation of Scratch. The major difference between Snap! and Scratch is that Snap! has a rich set of technical libraries. Some of these Snap! libraries include database and SQL interfaces, graphical trends, matrix manipulation, MQTT (standard messaging protocol for IoT), and Neural Net modeling. These additional libraries and other advanced features mean that Snap! is not just a teaching tool but is also ready to serve as a low-code alternative for IoT solutions.

The graphical low-code model is often useful in IoT environments, where less-experienced programmers are sometimes forced to adapt to the ideosyncracies of unfamiliar hardware. The Snap4Arduino [3] version of Snap! offers a unique set of libraries that will upload and configure Arduino modules without the user needing any Arduino knowledge or software.

You can run Snap4Arduino either from a web page or as a standalone Linux, macOS, or Windows app. In this article, I will show you three projects that will help you get started with Snap4Arduino. The first project allows the user to adjust the colors on a Neopixel array. The second project creates a dynamic bar chart that shows temperature and humidity values from a DHT11 sensor. The final project uses the SciSnap! library to show sensor data in a realtime line plot and then save the results to a CSV file.

Getting Started

The USB/serial communications between Snap4Arduino and the microcontroller module is via the Firmata protocol [4]. Firmata is a protocol designed to support communication between a microcontroller and a computer system. At project design time, the user can select which type of Firmata configuration they would like to install on their Arduino module.

The Snap4Arduino project offers an online web interface [5]. To get full functionality, you'll need to install a Chrome/Chromium plugin.

Desktop versions for 32- and 64-bit GNU/Linux systems are also available. It's important to note that there may be some compatibility issues when

Lead Image © Benoit Chartron, 123RF.com

switching between the desktop and the web versions, so it's best to stick to one version.

After you launch the Snap4Arduino web page or desktop application, you can reach the reference manual by clicking on the top-left *A* icon (Figure 1). The Snap4Arduino interface has three main areas. The left pane contains a palette area of blocks. These blocks are grouped and color-coded together based on their function. The center pane is the scripting area, and that is where the logic is created by dragging and dropping blocks together in a downward flowing pattern. The top-right pane is the stage area, which shows the visual output when a script is run.

The first step in any project is to load the required Firmata configuration onto an Arduino module (Figure 2). The Snap4Arduino interface supports Arduino UNO and Mega-compatible modules for uploads. While you're loading the Firmata configuration, you'll also need to enable *JavaScript extensions* and *Extension blocks* in the settings drop-down menu.(Note: other modules such as: Nano, Leonardo, Micro, Due, NodeMCU, etc., are supported, but you'll need to install Firmata manually through the Arduino IDE.)

To prove that the communications are working, you can create a test script to blink the module's onboard LED (pin 13). Figure 3 shows a blink example. The script uses four `Control` blocks (in orange), and three `Arduino` blocks (in light blue). Communications are enabled by clicking on the `Connect Arduino` button and then by selecting a port. (For most Linux systems the port will be `/dev/ttyACM0`.)

This test script uses a `for` block to cycle four times with a `set digital pin` to toggle pin 13 on and off. Time delays are enabled with `wait` blocks.

The Neopixel Project

Neopixels are individually addressable RGB color pixel arrays that come in strips, rings, and rectangular grids. From the previous test script, the Arduino module has already been loaded with the Neopixel firmata firmware. The next step is to import the Neopixel library into the Snap4Arduino project (Figure 4).

For this project, I used a 24-LED Neopixel ring (Figure 5), however, you could use any Neopixel array type. The alligator clips were used to connect the 5V, GND, and data pins (pin 6). My goal for this project was to allow the user to adjust the background color of the LEDs while a red LED circles around the ring. The script logic (Figure 6) starts by setting the Arduino connect port and defining 24 Neopixel LEDs on data pin 6. Four variables are defined: *pin, Red, Green,* and *Blue*. On the right stage pane, the color variables are defined with user-adjustable sliders.

Two `for` loops are embedded within a `forever` loop. The inner `for` sets all the LEDs to the values from the color variable (Red, Green, and Blue) sliders. The outer `for` loop moves the red LED around the circle every 0.5 seconds. When the script is running, the `pin` variable will show the position of the red circling LED.

On the top right of the Snap4Arduino screen are three icons to start (green flag), pause, and stop the script from

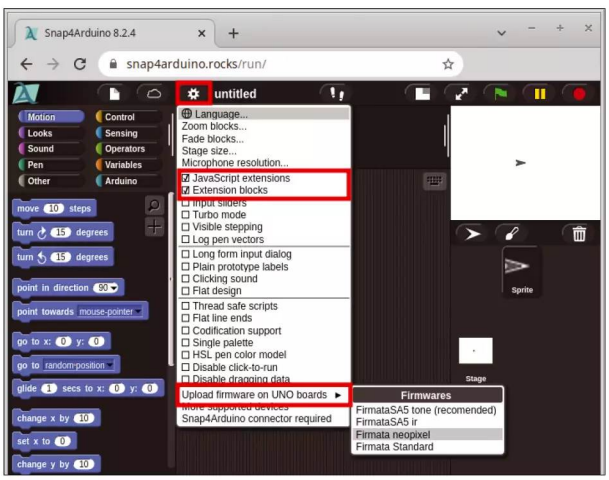


Figure 2: Upload Firmata firmware to an Arduino UNO module.

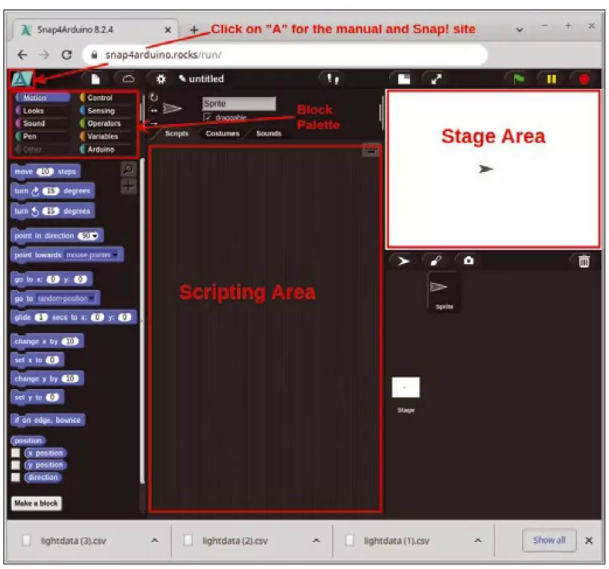


Figure 1: Snap! help is available in the Snap4Arduino interface.

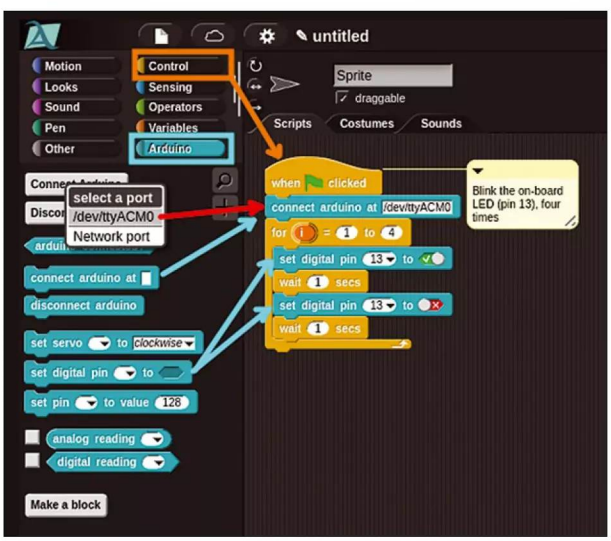


Figure 3: The blink script for an UNO onboard LED.

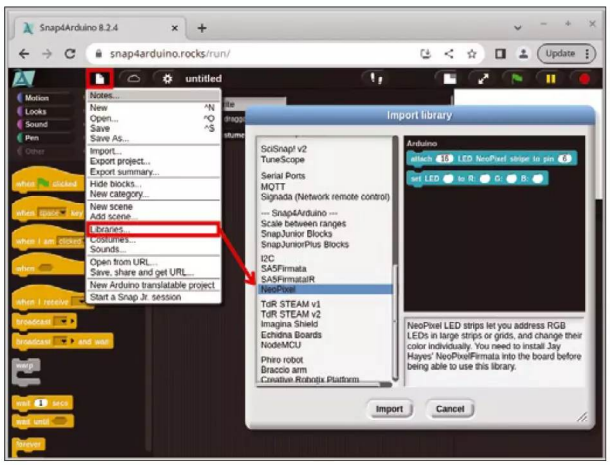


Figure 4: Import the Neopixel library into the Snap4Arduino project.

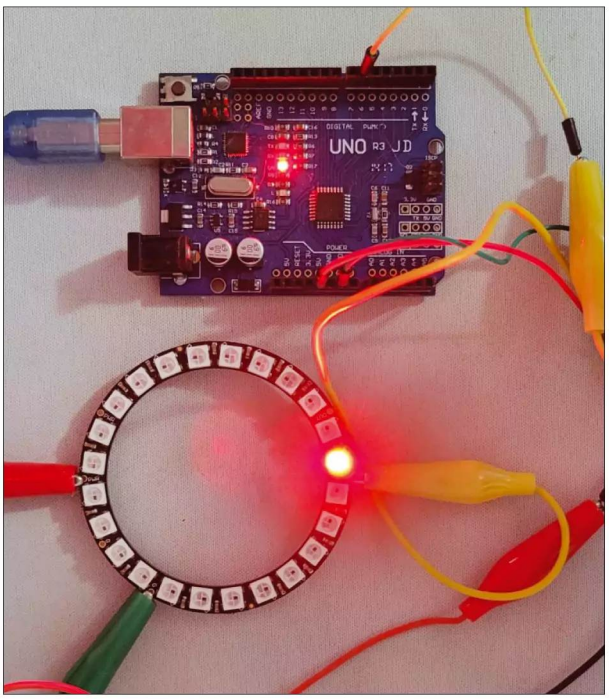


Figure 5: Setup for the Neopixel project.

running. You can also start scripts by clicking on the top of the logic `when green flag clicked` block.

DHT11 Temperature and Humidity Sensor Project

A DHT11 Sensor can be connected via expansion boards or by manually wiring the sensor to Arduino pins. Note, the Firmata firmware sets the DHT11 data pin on pin 4.

For my testing I used a TdR-STEAM-compatible board (Figure 7). These expansion boards are relatively inexpensive (~$12) and they offer two color LEDs, a Neopixel LED, a buzzer, a light sensor, a DHT11 sensor, and two input buttons.

This project uses a different Firmata configuration, so the Arduino firmware needs to be re-uploaded with the (recommended) FirmataSA5 tone firmware.

Next, Snap4Arduino requires you to import two libraries (Figure 8), the *Bar charts* and *TdR STEAM v2* libraries. The *Bar charts* library expects the data to be in a table or a CSV file. Tables can be created in Snap! by making a variable a list of lists.

A `Readings` variable can be made into a 2x2 table by using three `list` blocks and a `set` block (Figure 9). The `plot bar chart` block (which is found in the *Pen* section) is passed the `Readings` variable, along with x-y coordinates, width, and height values.

When these three blocks are run, the stage will show the `Readings` variable updated with the table information and a gray, static bar chart will appear. The final step is to make the bar chart dynamic with DHT11 sensor data. For this I create two additional variables, `humidity` and `tempC`, and I position the variables around the bar plot stage (Figure 10).

The two new variables are updated with sensor data by using `set` blocks to read `TdR temperature` and `TdR humidity` blocks. The chart color is defined by the `set pen color` block. Before a new bar chart can be drawn, a `clear` block is required.

When the final script is run, the bar chart and variables are refreshed with new sensor data every five seconds. Some good future features could include adding a time value to the chart or a toggle between degrees C and F.

In the next project, I'll look at how to save sensor data to a CSV file.

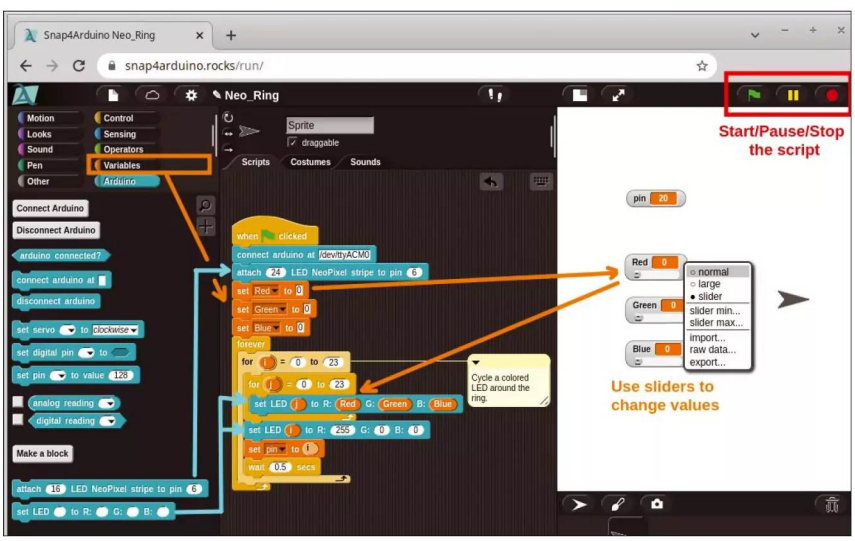


Figure 6: Script for the Neopixel project.

The SciSnap! Library

Before getting started on the final project, it is worthwhile to get some background on the SciSnap! library. This package is imported like the other libraries, and it offers a huge variety of extra blocks that

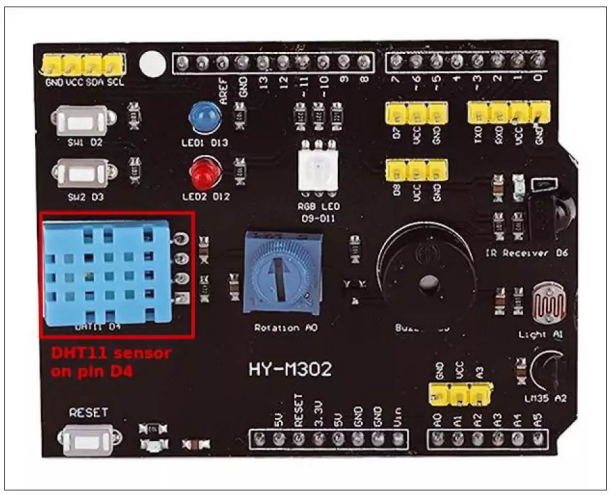


Figure 7: TdR-STEAM-compatible expansion board.

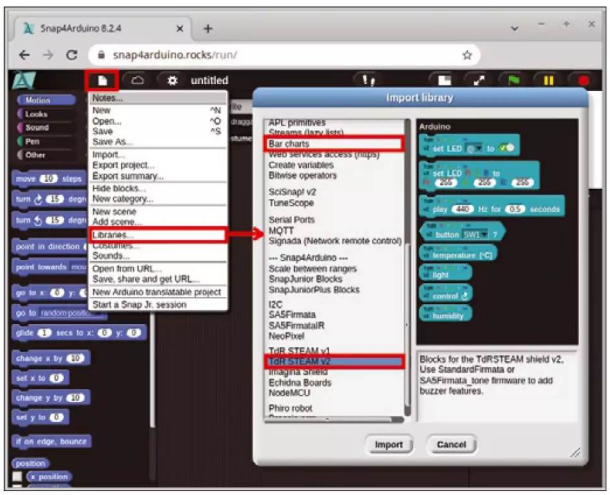


Figure 8: Import bar charts and TdR STEAM libraries for the DHT11 sensor project.

are grouped together in nine additional categories.

Figure 11 shows the power of the SciSnap library. This example lets the user select a CSV file from their local system. The data is then displayed in a table, with the maximum value in column 2 shown as a separate variable. The entire script to do this only takes four blocks!

When using this library in a project, you'll need to call a `start SciSnap` block before you can use any of its library blocks. This example uses the `Data tools` grouping. An `import` block is configured to use the *filepicker* option to select a CSV file and insert the data into the predefined `SciSnap!Data` variable.

To show the maximum data value, a `sort` is embedded within a `select row` block, and the result is set in the `Max_Value` variable. (Note: Row 1 has the headings, so row 2 has the maximum value.)

The SciSnap! library supports a large number of advanced features such as image manipulation, SQL tools, table and vector management, and neural networks. In the final project I'll use SciSnap! to show dynamic updates on a line plot.

Real-Time Plots Project

You can use any sensor input for the final project. I used the *TdR light* sensor, with a flash light to generate some peak values. To get started, the first step is to import the *SciSnap!* and *TdR STEAM* libraries into Snap4Arduino.

Figure 12 shows the full script and results for a line plot with 60 light sensor data points. The script logic starts with

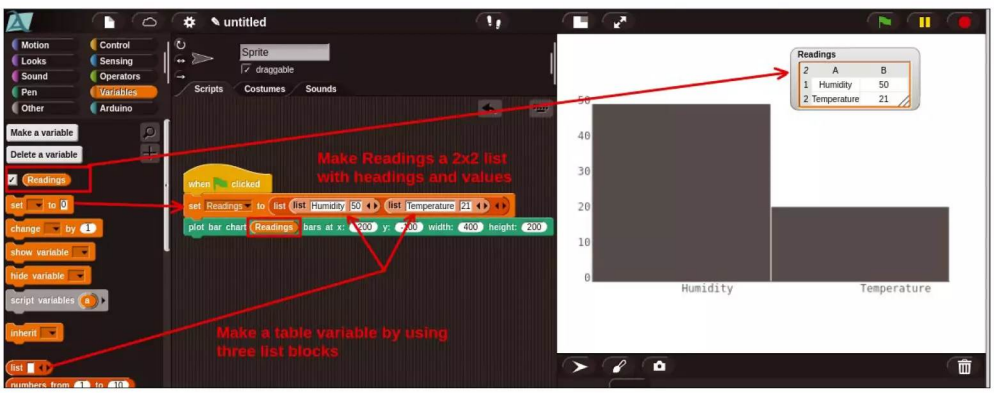


Figure 9: Create a static bar chart in Snap!

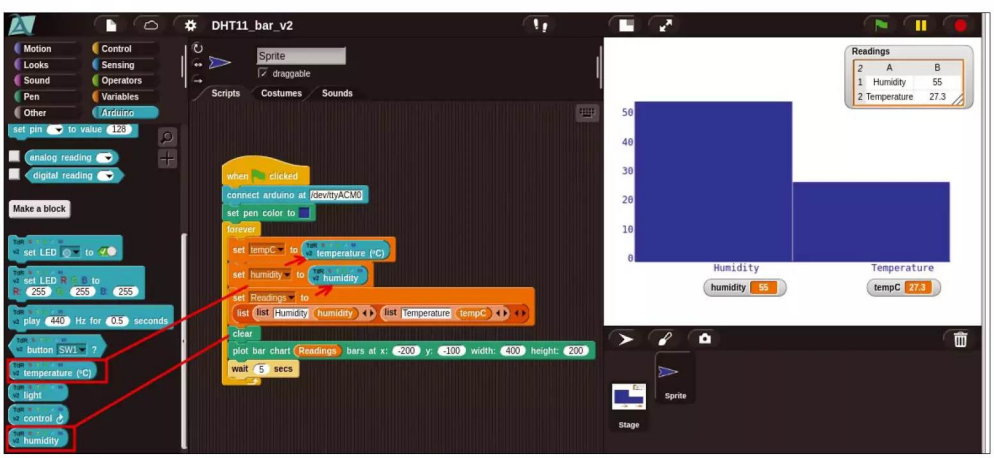


Figure 10: Displaying DHT11 Sensor data.

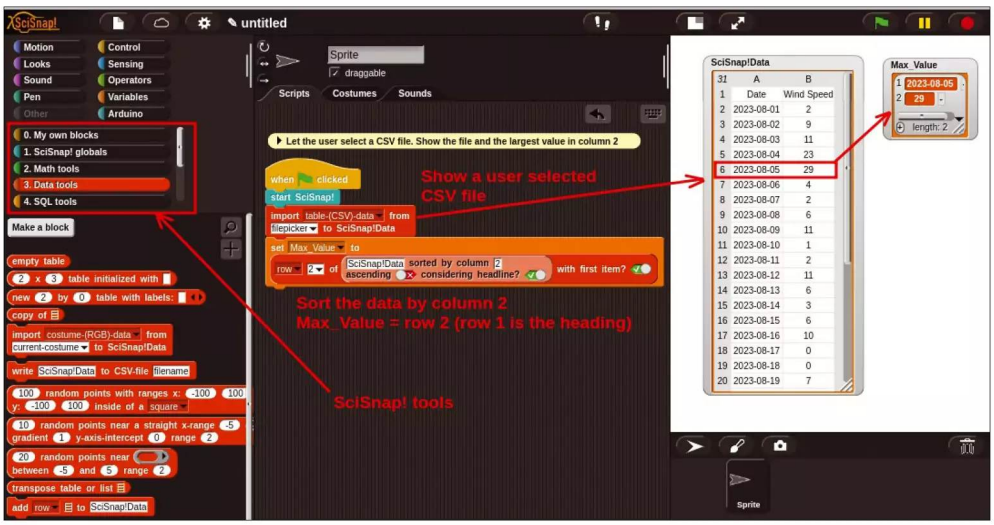


Figure 11: Open a user-selected CSV file and show a max value.

Figure 12: Script and real-time plot of sensor data.

a `connect arduino` and a `start SciSnap` block. The `add row` block builds a two-column table of the sample number and the sensor readings in the `SciSnap!Data` variable.

The data plot is created using the `PlotPad for data plots` grouping. The `configure as a PlotPad` block defines the chart size and positioning. The `set PlotPad ranges` sets the x-y scaling of the plot. (Note, the plot grows dynamically as new real-time data is added). The `set PlotPad labels` block defines the title and x-y labels. The `add dataplot of mixed data` block creates the line plot of evenly sampled data. Finally the `add axes and scales to PlotPad` shows the full labels and axes descriptions.

After the `for` loop completes, the sampled data is written to a CSV file (`lightdata.csv`), and a message box alerts the user that the script is complete. To save a copy of the data plot, right-click on the stage and select *pic*.

This project used a single data value, but it is also possible to use SciSnap! to handle multiple sensor values and do an x-y plot with linear regression.

Summary

Low-code programming techniques make it easy for inexperienced users to get started with programming, and Snap4Arduino takes care of the Arduino details, offering an ideal entry point for new users to quickly hook up and play with Arduino modules and simple sensors.

I found that, when I was moving between different Arduino modules and projects, I sometimes had to restart the Snap4Arduino application or web page. For users looking for more complex features, the SciSnap! library offers a wealth of functionality. ■■■

Author

For more of Pete's projects, see: *https://funprojects.blog*.

Info

[1] Scratch homepage: *https://scratch.mit.edu/*

[2] Snap! homepage: *https://snap.berkeley.edu/*

[3] Snap4Arduino homepage: *https://snap4arduino.rocks/*

[4] Firmata documentation: *https://github.com/firmata/protocol*

[5] Snap4Arduino web interface: *https://snap4arduino.rocks/run/*

Looking for your place in open source?

Set up job alerts and get started today!

OpenSource
JOB HUB

opensourcejobhub.com/jobs

Arduino development on the command line

At Your Command

When programming an Arduino microcontroller board for the first time, most people use the Arduino IDE, a graphical development environment. However, if you prefer the command line, you have a powerful alternative: Arduino CLI. *By Koen Vervloesem*

The Arduino [1] project was initially created in 2005 at the Interaction Design Institute Ivrea in Italy as an educational tool to teach students how to create and program interactive devices with sensors and actuators. Over time, the project expanded beyond its academic origins and became the go-to platform for hobbyists interested in programming microcontroller boards.

The Arduino graphical integrated development environment (IDE) [2] (Figure 1) has played a significant role in Arduino's success. It's easy to learn, without too many bells and whistles, but with all the basics you need. If you're satisfied with the Arduino IDE or if you use another IDE for Arduino, such as Visual Studio Code [3] with PlatformIO [4], feel free to continue using them. However, many developers have a command-line-centric workflow because it allows them to work more efficiently, while making it possible to check automatically whether an Arduino sketch still compiles correctly after a code or dependency update.

Fortunately, if you prefer a command-line environment for Arduino development, you have an official solution: Arduino CLI [5]. Although its API is still considered unstable until a 1.0 release, it's already an integral part of the Arduino IDE. The command-line interface (CLI) is used by the IDE as a back end for tasks such as detecting boards, compiling sketches, uploading firmware to boards, installing cores and libraries, and more.

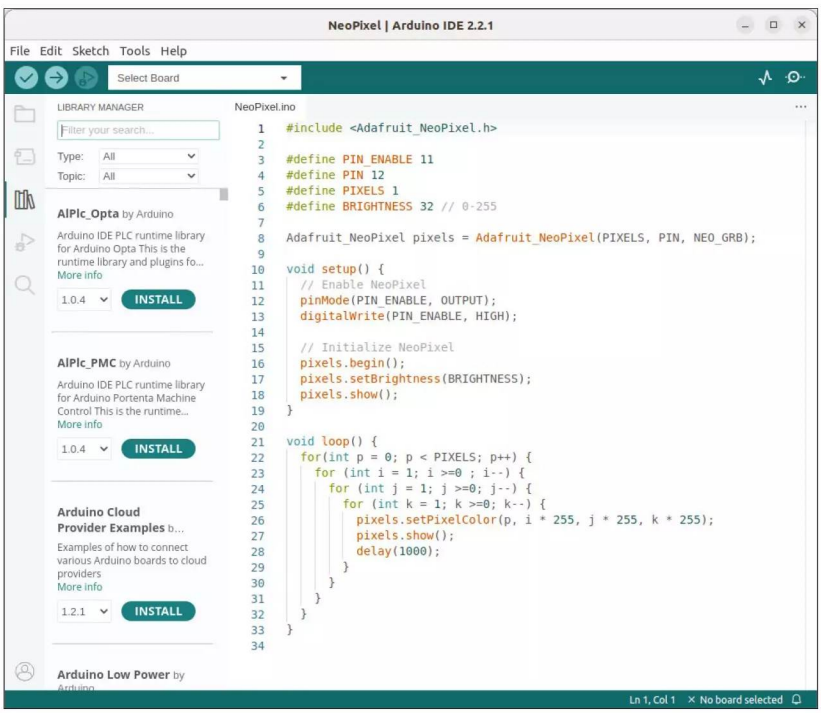


Figure 1: The Arduino IDE has played a significant role in Arduino's success.

Lead Image © innovari, Fotolia.com

In this article, I explore how to use Arduino CLI to develop, compile, and upload Arduino sketches entirely from the command line.

Installing Arduino CLI

GitHub has pre-built Linux binaries for all Arduino CLI releases [6], but the easiest way to install the latest version is with an install script:

```
$ curl -fsSL
  https://raw.githubusercontent.com/
  arduino/arduino-cli/master/
  install.sh | BINDIR=~/.local/bin sh
```

This script installs the `arduino-cli` command into the `~/.local/bin` directory. Change the `BINDIR` to any directory you want that's in your `PATH` environment variable. At the time of writing this article, this script installed Arduino CLI 0.34.2.

One additional task you need to perform to access your Arduino hardware over USB is to add your user to the `dialout` group if it is not already a member:

```
$ sudo usermod -a -G dialout $USER
```

You can check with the `id` or `groups` command. To activate this change, you'll need to log in again.

If you now run `arduino-cli` without any parameters, you'll see a list of supported subcommands (Figure 2). For each subcommand, you can request additional help by running either of the following commands:

```
arduino-cli subcommand --help
arduino-cli help subcommand
```

If you are already somewhat familiar with how Arduino works, you'll be able to figure out a lot on your own with these commands, because the approach is similar to that of the Arduino IDE, but on the command line.

Configuration

Arduino CLI is configured by command-line flags, environment variables, and a configuration file, in that order of priority. If a configuration option is not set, Arduino CLI uses a default value.

Thanks to these default values and the ability to configure Arduino CLI with command-line flags and environment variables, the command doesn't strictly require a configuration file to function. However, it's easier to store some commonly used options in a configuration file, so to begin, I'll create one with the default settings:

```
$ arduino-cli config init
Config file written to:
  /home/koan/.arduino15/
  arduino-cli.yaml
```

After creating the configuration file, this command also helpfully shows the file's location. If you want to view the current configuration, use the command:

Listing 1: Default Configuration

```
board_manager:
  additional_urls: []
build_cache:
  compilations_before_purge: 10
  ttl: 720h0m0s
daemon:
  port: "50051"
directories:
  data: /home/koan/.arduino15
  downloads: /home/koan/.arduino15/
staging
  user: /home/koan/Arduino
library:
  enable_unsafe_install: false
logging:
  file: ""
  format: text
  level: info
metrics:
  addr: :9090
  enabled: true
output:
  no_color: false
sketch:
  always_export_binaries: false
updater:
  enable_notification: true
```

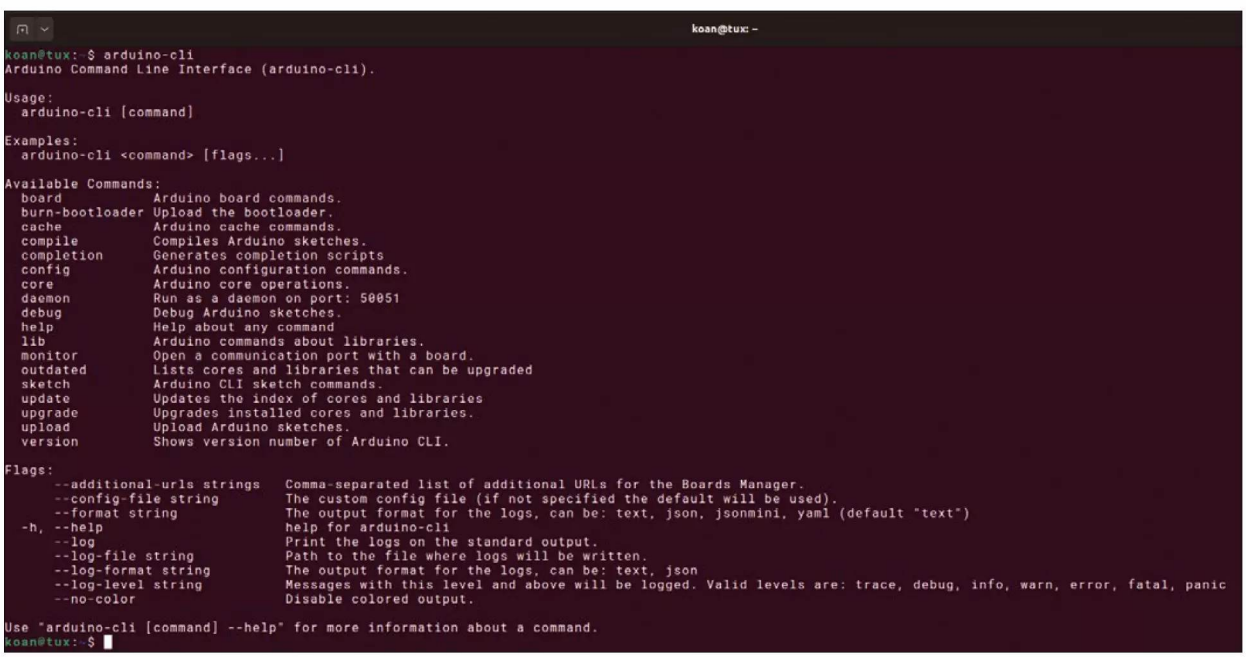


Figure 2: The `arduino-cli` command supports several subcommands for Arduino software development.

Listing 2: Query Board Core

```
Board Name          FQBN
Arduino NANO 33 IoT arduino:samd:nano_33_iot
Arduino Nano        arduino:avr:nano
```

Listing 3: Recognized Boards

```
$ arduino-cli board listall nano
Board Name          FQBN
Arduino NANO 33 IoT arduino:samd:nano_33_iot
Arduino Nano        arduino:avr:nano
Arduino Nano ESP32  arduino:esp32:nano_nora
```

```
$ arduino-cli config dump
```

Listing 1 shows the output of this command for the default configuration. If you need to know the location of the configuration file at a later time, add the `--verbose` option to the previous command.

Note that the content of the configuration file might differ from the output of the `arduino-cli config dump` command. If the configuration file doesn't have a specific configuration option, the command shows its default value instead.

Managing Arduino Cores

After installing Arduino CLI, the next thing you should do is update the local index of available Arduino cores:

```
$ arduino-cli core update-index
```

An Arduino core provides support for a specific board family. You can then query the list of installed cores:

```
$ arduino-cli core list
```

If you've already added cores from the Arduino IDE, you'll see them listed here as well because the command-line program uses the same installation files located in `~/.arduino15` and `~/Arduino`. The command displays the installed and the latest available version of each core, together with its ID and name. For example, the core with ID `arduino:avr` supports all Arduino boards with an AVR microcontroller, including the Arduino Uno, Mega, Nano, and Duemilanove.

If the command shows that a newer version of a specific core is available, you can upgrade with the command:

```
$ arduino-cli core upgrade arduino:avr
```

You can also upgrade all installed cores at once:

```
$ arduino-cli core upgrade
```

If you want to find out which core you need for a specific board (e.g., the Arduino Nano 33 IoT), submit the command:

```
$ arduino-cli board listall nano
```

Listing 2 shows that the Arduino Nano 33 IoT requires the `arduino:samd:` core. The fully-qualified board name (FQBN) is the unique identifier for each board, including its associated core.

If the previous command doesn't show the board in which you are interested (e.g., the Arduino Nano ESP32), it means the board is not supported by any of the installed cores. In that case, you need to search for the board in the Board Manager, which will show the core you need:

```
$ arduino-cli board search nano esp32
Board Name          FQBN Platform ID
Arduino Nano ESP32  arduino:esp32
```

The output reveals that the Nano ESP32 requires the `arduino:esp32` core. You can then install this core with the command:

```
$ arduino-cli core install arduino:esp32
```

After installation, check again whether your board is now recognized (Listing 3).

Listing 4: Blink the Built-In LED

```
01 void setup() {
02   pinMode(LED_BUILTIN, OUTPUT);
03 }
04
05 void loop() {
06   digitalWrite(LED_BUILTIN, HIGH);
07   delay(1000);
08   digitalWrite(LED_BUILTIN, LOW);
09   delay(1000);
10 }
```

Creating a New Sketch

Now that your Arduino cores are set up, it's time to do something with your microcontroller board. Just run the command

```
$ arduino-cli sketch new Blink
```

to create a new Arduino sketch named Blink. This command creates a directory named `Blink` in your current directory, containing a file named `Blink.ino`, which contains a template for an Arduino sketch with two empty functions, `setup()` and `loop()`:

```
void setup() {
}

void loop() {
}
```

Instead of having to edit this sketch in the Arduino IDE, you can now use your favorite editor (e.g., Vim or Emacs) to make changes. For example, to create a simple blinking light with the built-in LED that most Arduino boards have, enter the Arduino sketch in Listing 4, and save the file when you're done.

Connecting Your Arduino Board

Now connect your Arduino board to your PC with a USB cable and check whether Arduino CLI recognizes it (Listing 5). You should see the board name listed. If, however, the board name shows up as *Unknown*, it means Arduino CLI was unable to detect the FQBN and core automatically. In that case, you'll need to search manually for the FQBN, as explained earlier.

Compiling and Uploading a Sketch

Copy the string in the FQBN column of Listing 5 or obtain it from the board search results. To compile your sketch, enter:

Listing 5: Recognizing an Arduino Board

```
$ arduino-cli board list
Port         Protocol Type              Board Name  FQBN            Core
/dev/ttyUSB0 serial Serial Port (USB)   Arduino Uno arduino:avr:uno arduino:avr
```

```
$ arduino-cli compile ➥
  -b arduino:avr:uno Blink
```

Replace `arduino:avr:uno` with the FQBN of your board, and `Blink` with the directory in which your sketch is located.

If your code contains a syntax error, you'll see an error message. If your code is syntactically correct, the compiled sketch will appear in a temporary directory, and the command will exit without errors.

Next, copy the port from the Port column of Listing 5 (e.g., `/dev/ttyUSB0`). Use this port when uploading the compiled sketch to your board:

```
$ arduino-cli upload ➥
  -p /dev/ttyUSB0 ➥
  -b arduino:avr:uno Blink
```

If everything goes well, you should see a short output message saying *New upload port: /dev/ttyUSB0 (serial)*. If you want to verify that the sketch has been uploaded successfully, you can add the `--verbose` option to the upload commands, which provides you with the complete output of the uploading tool.

Once the code has been uploaded, the board's processor will reset, your code will start executing, and the built-in LED will start blinking. You can now disconnect your Arduino board from your PC and power it from another source. Your sketch is stored in the board's built-in flash memory and will start running every time the Arduino board boots up.

Adding External Cores

Support for additional boards can be added by adding a URL to Arduino's Board Manager. For example, Earle Philhower maintains an Arduino core for RP2040 boards, including the Raspberry Pi Pico Arduino core, Arduino-Pico [7]. You can add this URL to Arduino CLI's configuration with the command:

```
$ arduino-cli config ➥
  add board_manager.additional_urls ➥
  https://github.com/earlephilhower/➥
   arduino-pico/releases/download/➥
   global/package_rp2040_index.json
```

After adding the URL, you need to update the local index of available Arduino cores:

```
$ arduino-cli core update-index
```

If you now search for available RP2040 boards,

```
$ arduino-cli board search rp2040
```

you'll see a much longer list than before, with `rp2040:rp2040` as the necessary core, which you can install now:

```
$ arduino-cli core install rp2040:rp2040
```

Afterward, you can find the possible FQBNs for RP2040 boards with:

```
$ arduino-cli board listall rp2040
```

For example, to compile the Blink sketch on the Seeed Studio XIAO RP2040 (Figure 3), you need to use its FQBN with the `-b` flag:

```
$ arduino-cli compile ➥
  -b rp2040:rp2040:seeed_xiao_rp2040 ➥
  Blink
```

After connecting the XIAO RP2040 board to one of the USB ports on your PC and running `arduino-cli board list`, the command shows `/dev/ttyACM0` as its port, so now upload the compiled sketch:

```
$ arduino-cli upload ➥
  -p /dev/ttyACM0 ➥
  -b rp2040:rp2040:seeed_xiao_rp2040 ➥
  Blink
```

After the board resets, you should see a red LED blinking next to the USB C connector. However, the green and blue LEDs next to it are still on. To turn them off, you can modify the `Blink.ino` sketch. I found the pinout on a Seeed Studio wiki page [8], and Listing 6 shows the modified Arduino sketch to blink the green LED.

In this modified sketch, you first turn off the three user LEDs of the XIAO RP2040 (connected active low) with the `HIGH` state in the `setup()` function. Then, in the `loop()` function, you let one of the LEDs blink – in this case, the green one.

Listing 6: Blink XIAO RP2040 LED

```
01 #define LED_RED 17
02 #define LED_GREEN 16
03 #define LED_BLUE 25
04
05 #define LED LED_GREEN
06
07 void setup() {
08   pinMode(LED_RED, OUTPUT);
09   pinMode(LED_GREEN, OUTPUT);
10   pinMode(LED_BLUE, OUTPUT);
11
12   // The LEDs are connected active low.
13   // Set them to HIGH to turn them off.
14   digitalWrite(LED_RED, HIGH);
15   digitalWrite(LED_GREEN, HIGH);
16   digitalWrite(LED_BLUE, HIGH);
17 }
18
19 void loop() {
20   digitalWrite(LED, LOW);
21   delay(1000);
22   digitalWrite(LED, HIGH);
23   delay(1000);
24 }
```

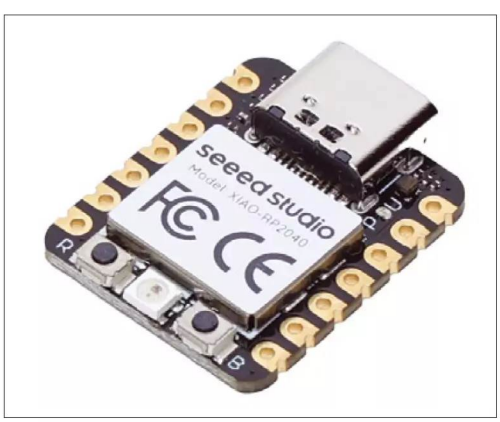


Figure 3: The Seeed Studio XIAO RP2040 is a powerful microcontroller board in a tiny package.

Managing Libraries

In reality, an Arduino sketch is rarely as simple and self-contained as the examples shown here. Most of the time, you'll need to use one or more libraries to communicate with sensors or other devices you connect to or for specific functionality such as JSON or MQTT.

For example, the XIAO RP2040 board also features a WS2812 RGB LED, commonly known as a NeoPixel, for which you will need a support library to make use of this functionality. To begin, update the index of Arduino libraries:

```
$ arduino-cli lib update-index
```

Next, query the list of installed libraries:

```
$ arduino-cli lib list
```

If you've already installed libraries from the Arduino IDE, you'll see them listed here. The command will display the installed and the latest available version for each library. If you notice that a newer version of a particular library (e.g., ArduinoJson) is available, you can update it:

```
$ arduino-cli lib upgrade ArduinoJson
```

If you are running the latest version, the Available column doesn't show a version number.

The command

```
$ arduino-cli lib upgrade
```

updates all installed libraries at once.

Adding a NeoPixel Library

Now that you have an updated list of libraries, search for a library in the list of available libraries that allows you to control WS2812 RGB LEDs. Use `neopixel` as the search word:

```
$ arduino-cli lib search neopixel
```

This command displays a lot of information for each library, such as the author, maintainer, project, supported architectures, and available versions (Figure 4). Choose the library that best suits your needs. For example, choose the Adafruit NeoPixel [9] library and install it (the library name can be found on one of the lines that starts with *Name:*):

```
$ arduino-cli lib ➥
  install "Adafruit NeoPixel"
```

Note that many libraries come with example sketches. You can find their location with the command

```
$ arduino-cli lib examples ➥
  "Adafruit NeoPixel"
```

Now that the library is installed, you can use it in your Arduino sketches.

Controlling the NeoPixel

To begin, create a new Arduino sketch:

```
$ arduino-cli sketch new NeoPixel
```

In the `NeoPixel/NeoPixel.ino` file, create the code shown in Listing 7 to test the NeoPixel on the Seeed Studio XIAO RP2040.

On the first line, you include the library installed earlier, before defining the number of pixels on the NeoPixel. Because the Seeed Studio XIAO RP2040 only has one built-in WS2812 RGB LED, you set `PIXELS` to 1. The brightness is set quite low because the LEDs are blindingly bright. Next, define the GPIO pin to which the NeoPixel is connected, as well as an enable pin, which is needed on the XIAO RP2040 to turn on the NeoPixel. After defining these macros, you create a `pixels` object with the number of pixels, the pin, and the LED type as arguments.

In the `setup()` function, you enable the NeoPixel and initialize it, setting its brightness. The `loop` function has a couple of nested loops to iterate through all the pixels (so this code also works if you connect a NeoPixel LED strip or ring) and RGB (red, green, blue) components. Basically, each pixel gets eight colors in a row with a delay of a second in between, after which the pixel is off (all RGB components are `0` at the end of the loop) and the next pixel takes its turn. It's a simple way to test whether all your pixels are working.

Finally, compile and upload the sketch to your board with the same

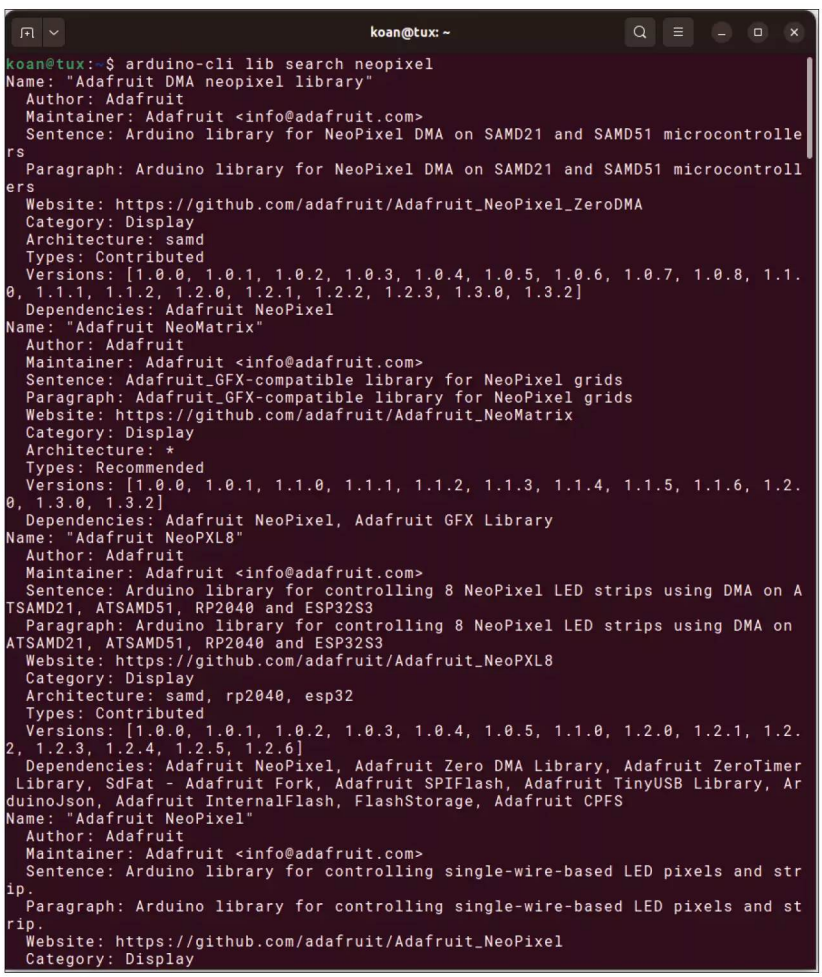


Figure 4: Search through all available Arduino libraries with `arduino-cli`.

commands you learned earlier. If everything goes well, you should now see the NeoPixel on your board cycling through different colors.

Arduino CLI in GitHub Actions

If you have a GitHub repository with Arduino sketches, you can use the `arduino/compile-sketches` GitHub action [10] to check whether the sketches compile correctly. Under the hood, this GitHub action runs Arduino CLI, installing the necessary cores and libraries that you specify, which allows you to check, for every commit, whether the sketches still compile against the latest cores and libraries for all the boards you specify. Another useful tool is Arduino Lint [11], which checks for common problems in Arduino projects. You can also run this tool automatically as part of a GitHub workflow with the `arduino/arduino-lint-action` [12].

Listing 7: Test NeoPixel on XIAO RP2040

```
01 #include <Adafruit_NeoPixel.h>
02
03 #define PIXELS 1
04 #define BRIGHTNESS 32 // 0-255
05 #define PIN 12
06 #define PIN_ENABLE 11
07
08 Adafruit_NeoPixel pixels = Adafruit_NeoPixel(PIXELS, PIN,
   NEO_GRB);
09
10 void setup() {
11   // Enable NeoPixel
12   pinMode(PIN_ENABLE, OUTPUT);
13   digitalWrite(PIN_ENABLE, HIGH);
14
15   // Initialize NeoPixel
16   pixels.begin();
17   pixels.setBrightness(BRIGHTNESS);
18   pixels.show();
19 }
20
21 void loop() {
22   for(int p = 0; p < PIXELS; p++) {
23     for (int i = 1; i >=0 ; i--) {
24       for (int j = 1; j >=0; j--) {
25         for (int k = 1; k >=0; k--) {
26           pixels.setPixelColor(p, i * 255, j * 255, k * 255);
27           pixels.show();
28           delay(1000);
29         }
30       }
31     }
32   }
33 }
```

Conclusion

In this article, I've explored the main features and tasks you can perform with Arduino CLI. The command provides many other options to customize its behavior. Be sure to check the output of

```
arduino-cli subcommand --help
```

for more information about available options. The `-v` (verbose) option is especially useful for troubleshooting, because it provides more detailed information. Once you get the hang of using Arduino CLI, you might find it difficult to go back to the Arduino IDE!

Author

Koen Vervloesem has been writing about Linux and open source, computer security, privacy, programming, artificial intelligence, and the Internet of Things for more than 20 years. You can find more on his website at *koen.vervloesem.eu*.

Info

[1] Arduino: *https://www.arduino.cc*

[2] Arduino IDE: *https://www.arduino.cc/en/software*

[3] Visual Studio Code: *https://code.visualstudio.com*

[4] PlatformIO: *https://platformio.org*

[5] Arduino CLI: *https://arduino.github.io/arduino-cli/*

[6] Arduino CLI releases: *https://github.com/arduino/arduino-cli/releases*

[7] Arduino-Pico: *https://arduino-pico.readthedocs.io*

[8] Getting Started with Seeed Studio XIAO RP2040: *https://wiki.seeedstudio.com/XIAO-RP2040/*

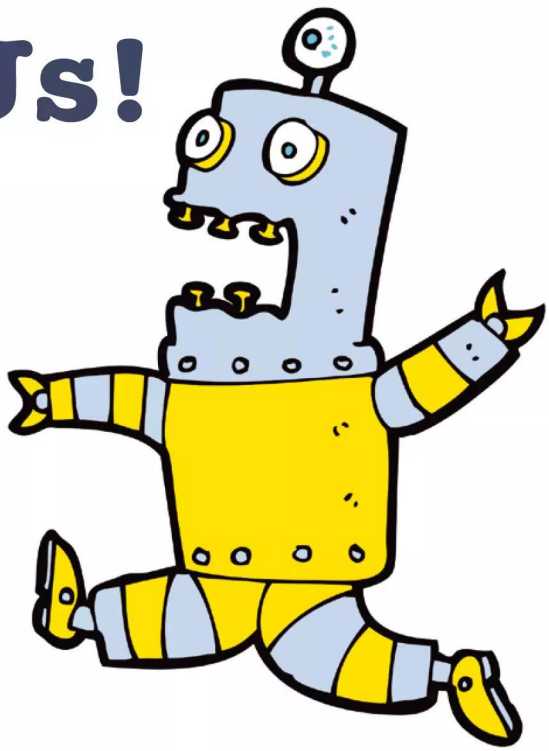
[9] Adafruit NeoPixel: *https://github.com/adafruit/Adafruit_NeoPixel*

[10] arduino/compile-sketches action: *https://github.com/arduino/compile-sketches*

[11] Arduino Lint: *https://arduino.github.io/arduino-lint/*

[12] arduino/arduino-lint-action: *https://github.com/arduino/arduino-lint-action*

Write for Us!



MakerSpace is looking for fresh, original articles on Raspberry Pi and other maker hardware platforms. If you work with Raspberry Pi, Arduino, BeagleBone, MinnowBoard, Parallella, or another similar technology, and you have an interesting story about a recent project or configuration, drop us a line at *edit@makerspace-magazine.com*. We're also seeking articles on software tools for maker hardware environments – including applications in the repositories of the leading Raspberry Pi operating systems, as well as homegrown scripts for custom configurations. We're especially interested in electronics projects that use Raspberry Pi's GPIO to control real-world hardware devices for a practical (or whimsical) purpose. Write for *MakerSpace* and share your story.

Authors

Contact Info

Editor-in-Chief
Hans-Georg Eßer

Senior Editor
Joe Casad

Managing Editor
Lori White

Localization & Translation
Ian Travis

Copy Editors
Amy Pettle, Aubrey Vaughn

Layout
Dena Friesen, Lori White

Cover Design
Dena Friesen, Illustration based on graphics by studiostoks,123RF.com

Advertising
Brian Osborn, bosborn@linuxnewmedia.com

Marketing Communications
Gwen Clark, gclark@linuxnewmedia.com

Publisher
Brian Osborn

Customer Service / Subscription
For USA and Canada:
Email: cs@linuxnewmedia.com
Phone: 1-866-247-2802
(toll-free from the US and Canada)
For all other countries:
Email: subs@linuxnewmedia.com
Linux New Media USA, LLC
4840 Bob Billings Parkway, Ste 104,
Lawrence, KS 66049, USA.
www.linuxnewmedia.com

While every care has been taken in the content of the magazine, the publishers cannot be held responsible for the accuracy of the information contained within it or any consequences arising from the use of it. The use of the DVD provided with the magazine or any material provided on it is at your own risk.

Copyright and Trademarks © 2024 Linux New Media USA, LLC

No material may be reproduced in any form whatsoever in whole or in part without the written permission of the publishers. It is assumed that all correspondence sent, for example, letters, email, faxes, photographs, articles, drawings, are supplied for publication or license to third parties on a non-exclusive worldwide basis by Linux New Media unless otherwise stated in writing.

MakerSpace is published annually as MakerSpace (ISSN 2831-7165) by: Linux New Media USA, LLC, 4840 Bob Billings Pkwy. Ste 104, Lawrence, KS 66049, USA.

All brand or product names are trademarks of their respective owners. Contact us if we haven't credited your copyright; we will always correct any oversight.

Printed in Nuremberg, Germany by Kolibri Druck.

Distributed by Seymour Distribution Ltd, United Kingdom

Represented in Europe and other territories by: Sparkhaus Media GmbH, Bialasstr. 1a, 85625 Glonn, Germany.

Online: ISSN 2831-7173, Print: ISSN 2831-7165

Image © lineartestpilo, 123RF.com

Every week that you don't subscribe to Elektor's Newsletter (e-zine) is a week with great electronics-related articles and projects that you miss!

Stay Informed, Get Creative, and Win Prizes – Subscribe Now to be Part of it!

www.elektormagazine.com/ms-news

What can you expect?

Editorial
Every Friday, you'll receive the best articles and projects of the week. We cover MCU-based projects, IoT, programming, AI, and more!

Store
Don't miss our Elektor Store promotions. Every Tuesday and Sunday (and occasionally Thursdays), we'll have a special deal for you.

Partner mailing
You want to stay informed about the ongoing activities within the industry? Then this e-mail will give you the best insights. Non-regular but always Wednesdays.

elektor
design > share > earn

AMD power @ TUXEDO

TUXEDO Polaris 17 - Gen5

17.3-inch mid-range Linux gamer with highly efficient AMD Ryzen processor, fast NVIDIA RTX graphics and up to 96(!) GB DDR5 5600 MHz RAM.

TUXEDO Sirius 16 - Gen2

All-AMD Linux gaming laptop with highly efficient Ryzen 7 8845HS and fast Radeon RX 7600M XT graphics.

TUXEDO Pulse 14 - Gen3

Ultra portable CPU work-station with AMD Ryzen 7 7840HS, high-res 3K display and 32 GB LPDDR5-6400 high-efficiency RAM.

tuxe.do/masp2024

Linux compatible

Up to 5 Years Guarantee

Immediately ready for use

TUXEDO

Made in Germany

German Data Privacy

German Tech Support