Institut Suisse de Spéléologie et Karstologie

Datalogger Guide: Usage, Setup, and Operational Insights

by

Nicolas Schmid

A report made during my civil service at ISSKA, to explain my work and make it easier for other employees or civilists to use and modify the dataloggers.

August 25, 2023

Contents

1 Datalogger General Overview

ISSKA requires a variety of sensors to collect data in caves or in outdoor locations. A Datalogger is needed for this to control the sensors and store their values with a given time step. Most commercial solutions are not suitable for these environments due to factors such as mud, dust, high air humidity, difficult accessibility, lack of internet connection and absence of external power sources. Some commercial solutions exist, but they're expensive and can't connect with just any sensor.

The institute manufactures its own dataloggers, which makes it possible to adapt these to almost any sensor. It has the advantage of beeing cheap and flexible, but it is not a plug-and-play solution. It requires basic knowledge about electronics and Arduino programming, which are explained in this report.

1.1 The TinyPico Micro-controller

The dataloggers utilize a TinyPico micro-controller, equipped with a cost-effective yet powerful ESP32 microprocessor. The ESP32 offers superior performance compared to similar micro-controllers like Arduino, thanks to its enhanced memory and higher clock frequency. This increased power is sufficient for any datalogger application.

The TinyPico can be placed in deep sleep mode, consuming minimal current (around 10 µA). This feature enables dataloggers to operate in caves for extended periods, utilizing smaller and more portable batteries. The micro-controller controls various datalogger components, including the screen, external clock, SD card and sensors. Programming the micro-controller with Arduino language is straightforward; connect its USB port to a computer and upload code using the Arduino IDE. Refer to Section 2.1 for details on programming a TinyPico using the Arduino IDE.

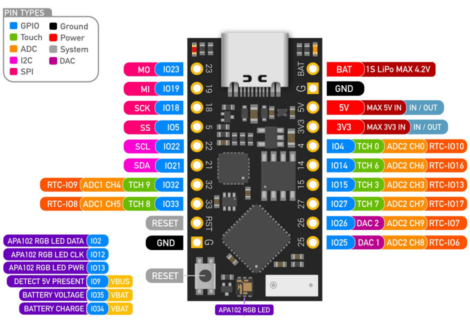


Figure 1: TinyPico micro-controller, featuring labeled ports. The micro-controller uses only 10 µA in deep sleep mode. [1]

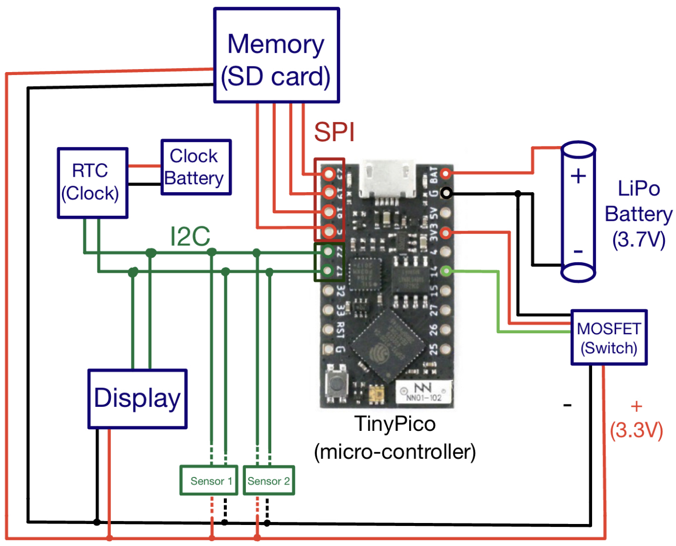


Figure 2: Schematic of a datalogger, illustrating its main components and connections.

1.2 Communication Protocols

Figure 2 illustrates how the TinyPico connects to other datalogger components. Diverse communication protocols are used for data exchange between components. These protocols vary in terms of cable usage (number of wires), maximum data transmission distance, transmission rate, device count, complexity, etc.

3 different communication protocols are used in our dataloggers:

- **I^{2}C :** The datalogger employs I^{2}C to communicate with most sensors, the external clock, and the display. This protocol requires only two wires and supports communication with up to 128 devices on the same bus. This is possible because each device has a unique 7-bit address stored in its memory. Before sending or receiving data from a device, the adress of a device is sent on the I^{2}C bus to inform all the devices connected to the bus if we communicate with them or with another device. Hardware implementation is straightforward; connect the 2 pins from the TinyPico (master device) called SCL and SDA to the SCL and SDA wires from the slave devices.

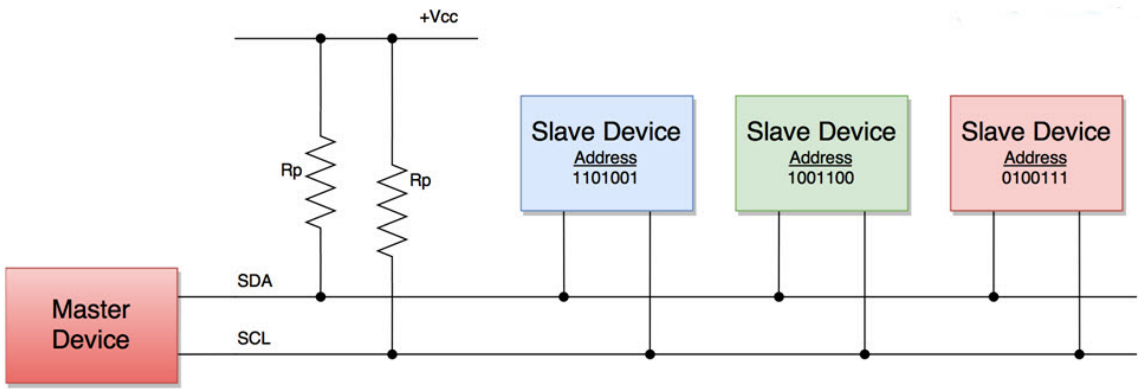


Figure 3: I^{2}C communication protocol.

This protocol was originally designed to communicate over short distances of a few centimeters, but it actually also works over longer distances (up to 50 m in my experience) but the longer the distance is, the lower must the communication frequency be. The communication frequency is usually 100 kHz for distances of a few centimeters, but it must be reduced to about 1000 Hz for a communication cable length of 50 m.

To ensure proper communication, a pull-up resistor R_p should be inserted between the SCL/SDA lines and the 3.3 V power source as depicted in Figure 3. These pull-up resistors allow the voltage to rise faster in the communication cables. The appropriate value of R_p hinges on the total bus capacitance C_{bus} and the desired communication frequency. While a wide range of resistance values is effective, it's worth noting that lower R_p values enhance communication at the expense of increased power consumption. For extended communication cables, consider reducing the overall R_p value by adding pull-up resistors in parallel. Here is a formula for the range of R_p [2]:

$$R_p\text{min} = \frac{V_{cc} - 0.4\text{V}}{3\text{mA}} = 1k\Omega \qquad R_p\text{max} = \frac{1000\text{ns}}{C_{\text{bus}}}$$

But in general, adding a 5 kΩ to 10 kΩ resistance in parallel for each sensor seems like a reasonable compromise for most users. For additional insights into selecting suitable pull-up resistor values for I^2C communication, you can refer to this stack-exchange thread.

- **SPI :** This communication protocol is used to communicate between the TinyPico and the SD card slot. This protocol employs 4 wires and supports multiple devices on the same bus, but unlike I^2C one extra wire is needed for each new sensor (see Figure 4). This makes the connections between devices more complicated since we need 4 wires instead of 2 with I^2C, but this protocol can communicate faster than I^2C and the hardware of a SPI device is simpler, so this is why it is used to communicate with the SD card.

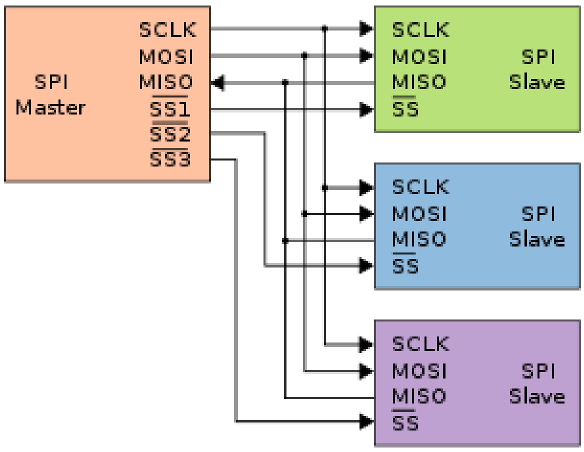


Figure 4: SPI communication protocol.

- **Serial :** Serial protocol is used for code upload and communication with some sensors. It is simple, but rather slow. This protocol also only requires 2 wires, but only 1 device can be connected to these 2 wires. This can be problematic if we want more than one sensor communicate with this protocol. For multiple devices, a serial multiplexer or I^2C-based sensors are preferred.

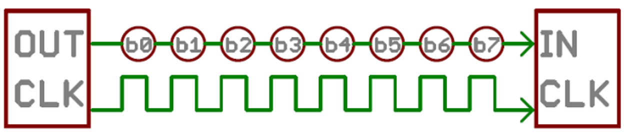


Figure 5: Serial communication protocol.

1.3 The Memory

A microSD card is used to store values measured by the sensors. For most applications, 64MB of storage is sufficient, but because microSD cards have become very affordable over time, we sometimes use ones with up to 8GB of storage.

Sensor values are stored in a file called *data.csv* with the format shown in Figure 6. The file always includes 3 columns: *ID*, *DateTime*, and *Vbatt*. Additional columns depend on the connected sensors. The file is created automatically on the microSD card when the datalogger boots up if the file doesn't already exist.

```
ID;DateTime;VBatt;temperature;pressure;box_humidity;SFMflow;
1;2023/06/15 14:00:00;4.05;28.390;906.16;34.55;0.01666667;
2;2023/06/15 14:05:00;4.06;28.410;906.17;34.12;-0.04166667;
3;2023/06/15 14:10:00;4.05;28.530;906.16;33.75;-0.03333334;
4;2023/06/15 14:15:00;4.06;28.520;906.16;33.79;-0.04166667;
```

Figure 6: Example of the data.csv file content.

Another file called *conf.txt* contains configurations settings for the datalogger. An example configuration file is shown in Figure 7. This file allows users to adjust the time step between measurements and set the correct time on the external clock without having to make changes in the code.

```
300; \\time step in seconds between measurements
1; \\boolean value to set the RTC when booting (1 or 0)
2023/08/25 14:05:00; \\time to set the RTC if the set value is 1
```

Figure 7: Example of the conf.txt file content.

1.4 Real Time Clock

The real time clock (RTC) keeps track of time and wakes up the datalogger at set intervals. Our dataloggers use the DS3231 RTC. This RTC has an accuracy of 2ppm, meaning it only drifts by 2 seconds every 1 million seconds, which is about 1 minute per year [3].

This RTC communicates via I^2C with the TinyPico, hence the SCL and SDA pins of the DS3231 are connected to the TinyPico. The SQW pin is also connected to the TinyPico, because it is used to wake-up the TinyPico from a deep-sleep mode. Before entering deep-sleep mode, the TinyPico sends via I^2C the information to the clock to set an alarm at a given time in the future. When the time comes, the alarm triggers the SQW pin, waking up the TinyPico.

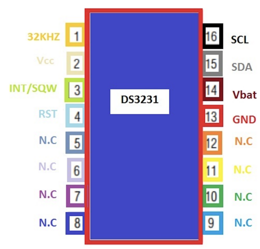


Figure 8: The DS3231 RTC pin layout.

The RTC only keeps track of time when powered, so we have a small battery for it that's always connected. To set the RTC, put a battery in and write a future time on the *conf.txt* file along with "1" for the setRTC boolean value. Put the microSD card in, power on, and press the TinyPico's reset button when the current time matches the one on *conf.txt*. Then, set the boolean value to "0" in *conf.txt* to avoid setting the wrong time next time you power the datalogger.

1.5 Power Management

To minimise the power consumption of the datalogger, the TinyPico is put into deep-sleep mode where in consumes almost no current. However, the sensors need to be powered off too. A MOSFET driver shown in Figure 9 acts as a switch for the sensors' power.

The GND pin is connected to the GND of the TinyPico, and VIN to its 3.3 V pin. INSENS is connected to the pin 14 of the TinyPico, hence the micro-controllers controls it. All sensors, display and SD card reader are connected to both the 3.3 V source and to GSENS.

When INSENS is low (below 1.5 V), GND and GSENS disconnect, but when INSENS is high (above 2.5 V), GND and GSENS connect [4], and the power can flow to the sensors, display and SD card reader. For additional insights about the design of the MOSFET driver and choice of its component you can refer to the MOSFET driver made by Adafruit.

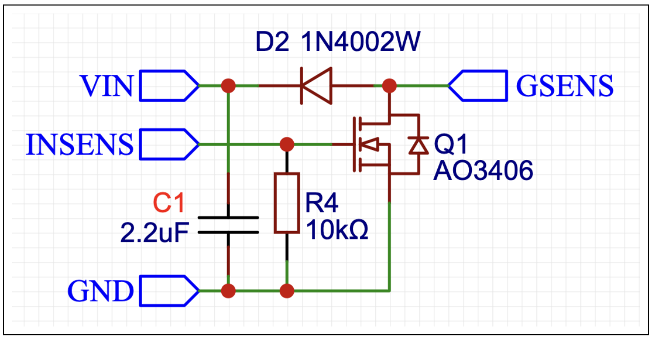


Figure 9: MOSFET driver schematics from EasyEDA.

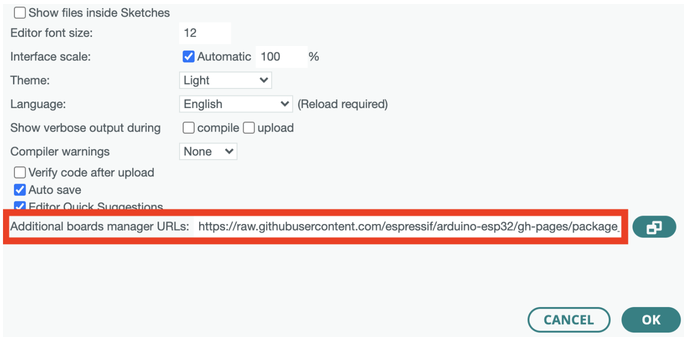
This brings the datalogger's power consumption to under 0.2 mA of current during sleep. For instance, if we use a 3.7 V LiPo battery with 2000mAh capacity, it would last $\frac{2000mAh}{0.2mA} = 10000h$ which is about 400 days. This is the main power consumption, while the rest depends on sensor types. If higher power sensors are needed or longer runtime desired, a battery with greater capacity can be used.

2 Programmation of Dataloggers

2.1 Setup of Arduino IDE

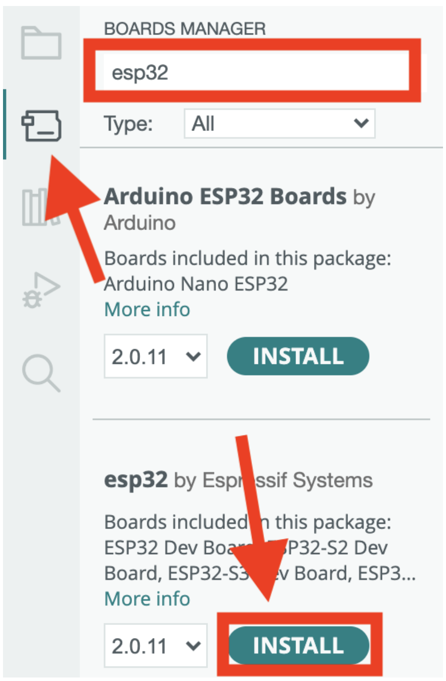
First step is to download the Arduino IDE. I suggest you to download Arduino IDE 2.x since it looks better than version 1 and has a better serial plotter which makes it more comfortable to work with. If you are lost with my explanations, you can try to follow this tutorial.

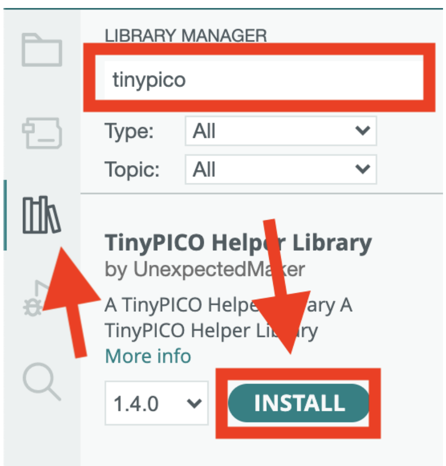
Since we will work with an ESP32, we need to add the TinyPico in the Board manager. For this, first go to **File** → **Preferences** for Windows or to **Arduino IDE** → **Preferences** for macOS. On the bottom of this page in the box for **Additional boards manager URLs** you must copy the following link and then press **OK**:



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

Once this is done you can select the icon **board manager** on the left of your window. Search for **esp32** and select *esp32 by Espressif Systems* by clicking on its **install** button. You will also have to install some libraries. For this, simply search your libraries on the **library manager** (e.g. the TinyPico library) and click on **install**. A list of the needed libraries is shown in Table 1.





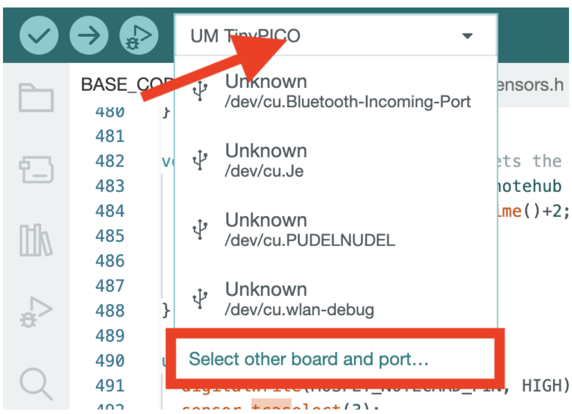
Library name	Developper name	Application of the library
TinyPICO Helper Library	UnexpectedMaker	This library allows us to simply get the voltage of the TinyPico and control its RGB led
RTClib	Adafruit	Library which facilitates communictation with the DS3231 RTC
U8g2	oliver	Library used to communicate with a variety of displays including the U8X8 SSD1306 used in our dataloggers
BMP581 Arduino Library	Sparkfun	The BMP581 is a high precision pressure and temperature sensor which is by default on our dataloggers' PCB
SHT31	Rob Tillaard	This library works with the SHT35 sensor, which is by default on our dataloggers to measure humidity and temperature

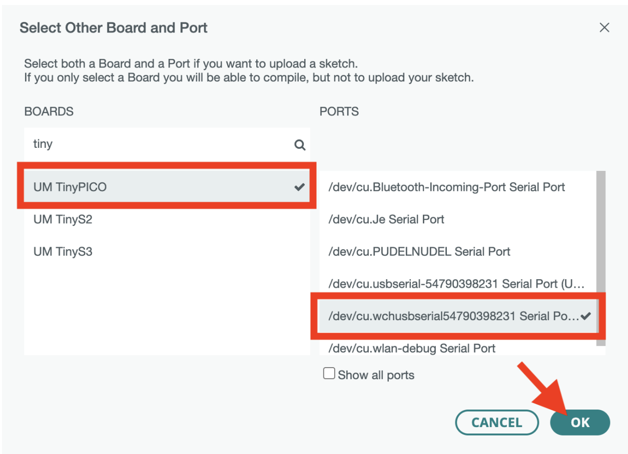
Table 1: Arduino libraries needed to compile the datalogger base code.

With all these library installed you should be able to compile the base code for the dataloggers. The base code is the skeleton of the code of any datalogger, only some minor changes must be made when adding a sensor. Obviously for all new sensors you will have to install new libraries. If the library cannot be found in the library manager refer to this tutorial to install the library manually.

The TinyPico V3 uses a CH9102F to connect the USB to the PICO-D4 ESP microprocessor, so you need to make sure you have the driver installed [5]. You can grab the latest drivers from the WCH website. In my case I installed the WCH34xVCPDriver for macOS.

You can now try to put your first program on a datalogger. You can for example download the base code for the dataloggers on Github (add github link) and open it with the Arduino IDE. Then connect the TinyPico to your computer with an USB cable. On the top of the window you can **select the port and the board**. For the board, search with the search-bar and select the TinyPico board, and for the port select the port to which the TinyPico is connected.





If the driver is installed (you may need a reboot your computer) the TinyPico should appear in the Arduino IDE and as a device on your computer in the following formats:

- macOS (TinyPICO V2): **/dev/tty.SLAB_USBtoUART**
- macOS (TinyPICO V3): **/dev/tty.wchusbserialXXX** where x is the index of the USB device
- Linux (TinyPICO V2): **/dev/ttyUSBx** where x is the index of the USB device
- Linux (TinyPICO V3): **/dev/ttyACM0**
- Windows: **COMn** where n is the port number assigned by Windows

Once the port and board are selected, click on the arrow on the top left of your window to **upload** the code. If the upload of the code fails try maybe another USB cable, some of these are not made for data transfer but only to power devices.

When the code was uploaded successfully you should see something like Figure 10 as output.

```
Hash of data verified.
Compressed 8192 bytes to 47...
Writing at 0x0000e000... (100 %)
Wrote 8192 bytes (47 compressed) at 0x0000e000 in 0.1 seconds (effective 633.8 kbit/s)...
Hash of data verified.
Compressed 420816 bytes to 246311...
Writing at 0x00010000... (6 %)
Writing at 0x0001b250... (12 %)
Writing at 0x00028e83... (18 %)
Writing at 0x0002e2c1... (25 %)
Writing at 0x00033f1b... (31 %)
Writing at 0x000394df... (37 %)
Writing at 0x0003f125... (43 %)
Writing at 0x000447d4... (50 %)
Writing at 0x00049f88... (56 %)
Writing at 0x0004ee93... (62 %)
Writing at 0x0005439a... (68 %)
Writing at 0x0005a9a4... (75 %)
Writing at 0x000639e4... (81 %)
Writing at 0x0006b944... (87 %)
Writing at 0x00070dbe... (93 %)
Writing at 0x000768d0... (100 %)
Wrote 420816 bytes (246311 compressed) at 0x00010000 in 4.2 seconds (effective 801.8 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
```

Figure 10: Upload output from the Arduino IDE.

2.2 Base Code Walk-through

2.2.1 Read and write on SD card

2.2.2 Display Data

2.2.3 Read and write on SD card

2.2.4 Set and Read the RTC

2.2.5 Read Sensor's Value

2.2.6 Deep Sleep

2.3 How to Modify the Code for New Sensors

2.4 ISSKA Github

3 PCB Design

4 Sensors Overview

5 Wireless Transmission

6 User's Guide

The goal of this section is to explain to a user with little knowledge of electronics of to use a datalogger.

References

[1] Seon Rozenblum, *TinyPico, A Tiny Mighty ESP32 Development Board*, August 2023.

[2] Texas Instrument, *I2C Bus Pullup Resistor Calculation*, February 2015

[3] Maxim Integrated Products *Extremely Accurate I2C-Integrated RTC/TCXO/Crystal*, 2015

[4] Alpha and Omega semiconductor, *AO3406 30V N-Channel MOSFET*, 2011

[5] Seon Rozenblum, *Getting satrted, TinyPico FAQ*