SG Wireless Documentation

Release 1.4.0

SG Wireless

May 22, 2026

CONTENTS

We designed the F1 Starter Kit to get you connected straight out of the box.

Begin your IoT journey in three simple steps:

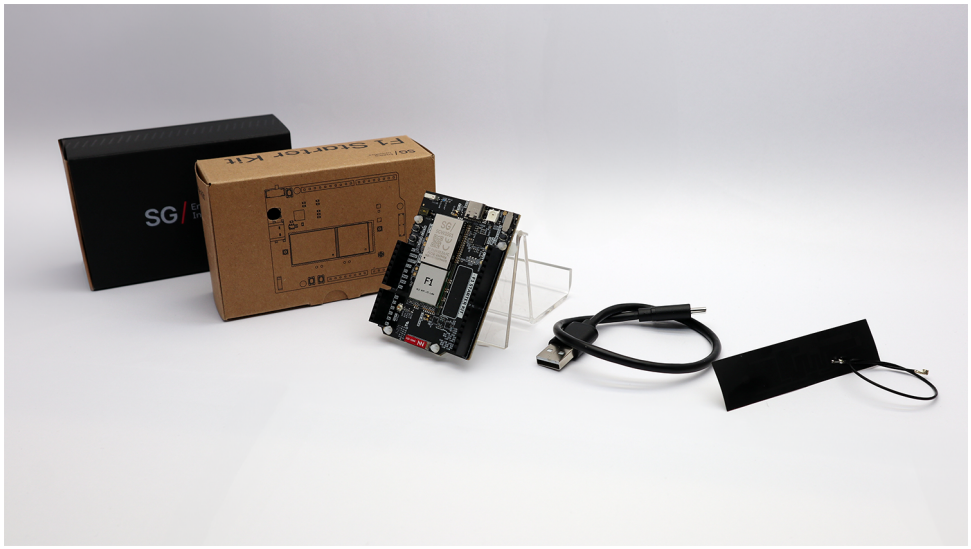
1. *Provision your F1 Starter Kit* – connect it to Ctrl through your preferred network connection.
2. *Set up your CAP/T sensor* – send its data to Ctrl through F1.
3. *Execute your first code* – get your first "Hello F1" printed out.

CHAPTER

ONE

UNBOXING YOUR STARTER KIT

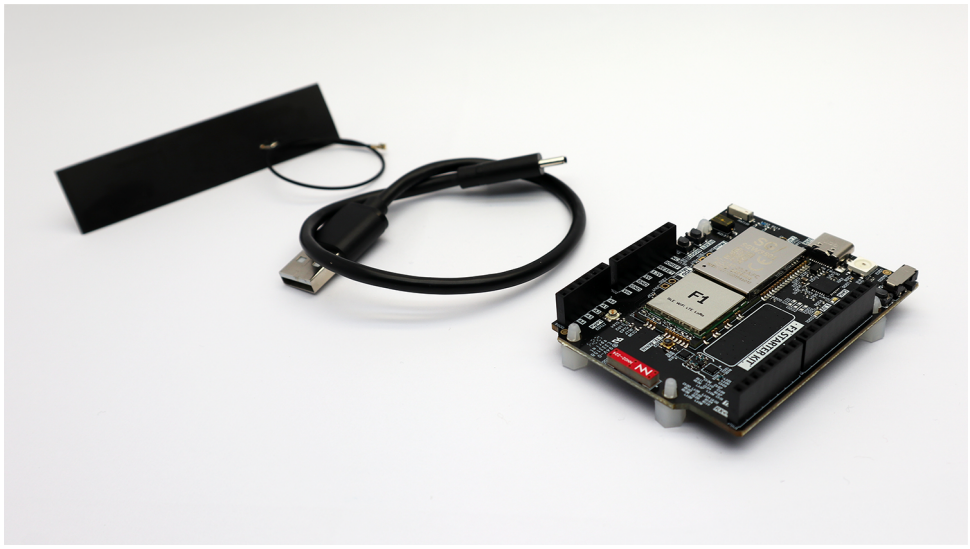
While our packaging is recyclable, don't throw it away just yet! It is in fact a ready-to-go housing that you can already use for initial prototyping. Here you'll see why.



Your F1 Starter Kit has 3 components:

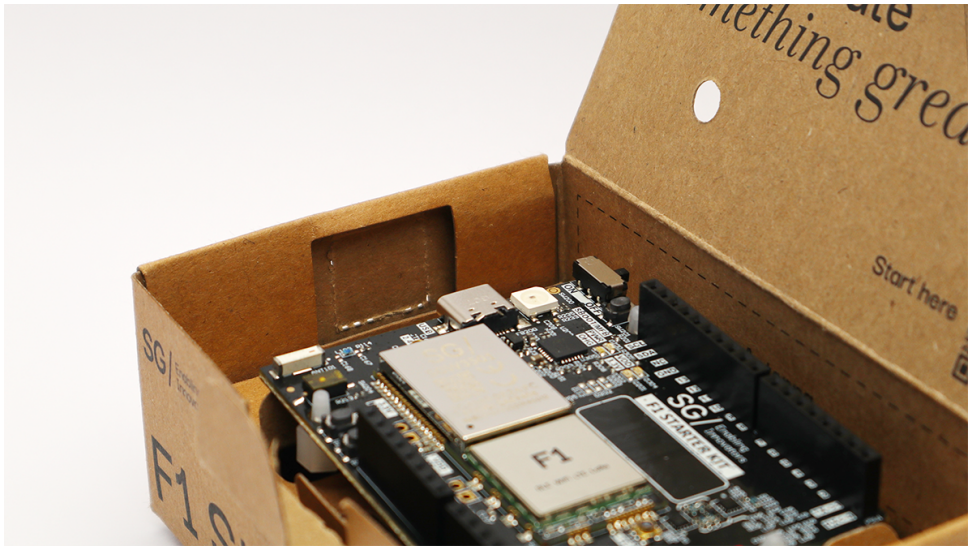
- F1 Evaluation Board with global SIM card (pre-inserted)
- USB cable (type A to type C)
- LoRa FPC antenna

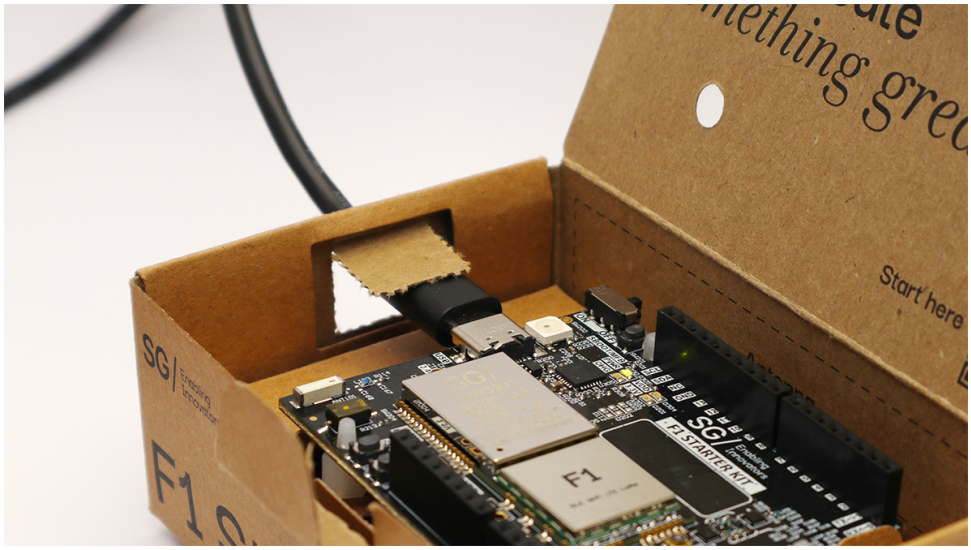
The F1 EVB is enclosed in an anti-static bag (not pictured).



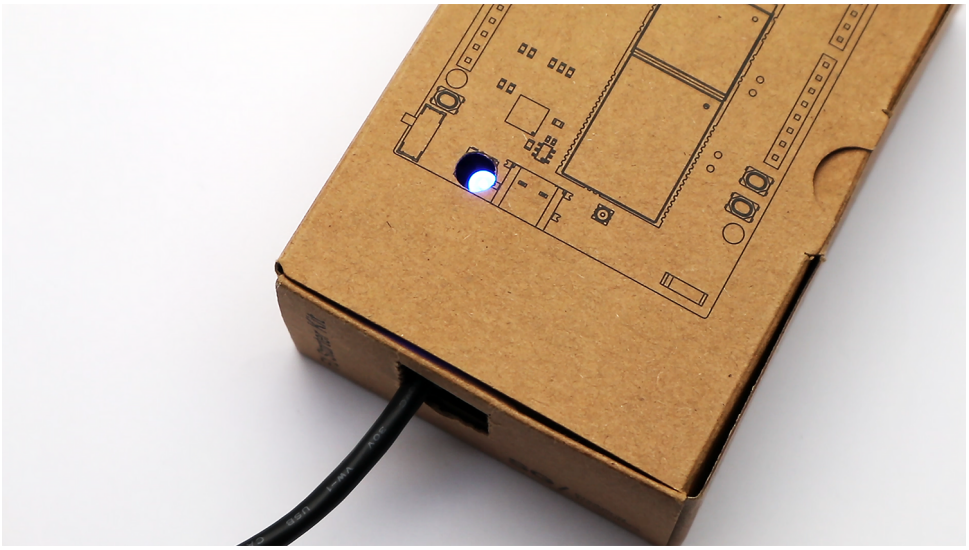
The packaging will secure your F1 Evaluation Board and keep it safe from the elements as you conduct your evaluation.

The rectangular cutout on the side of the box secures the cable when you power up the Board.





The blue LED indicator for ZTP is visible even when the box is closed.



Handily included are options for securing the LoRa FPC antenna, and you can even connect and secure an external antenna (not included) to the Board with the circular cutout at the side of the box.

CHAPTER

TWO

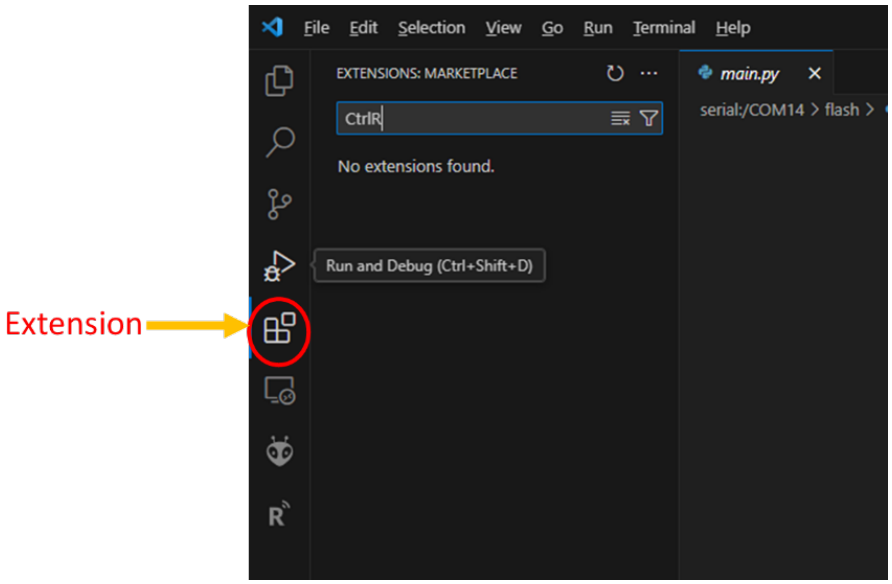
INTRODUCING VS CTRLR

For custom IoT application development, the F1 Starter Kit can be configured on the Microsoft Visual Studio Code IDE platform with the CtrlR Plugin.

You will need the latest version of Visual Studio Code to proceed, which can be downloaded and installed here.

2.1 Installing VS CtrlR Plugin

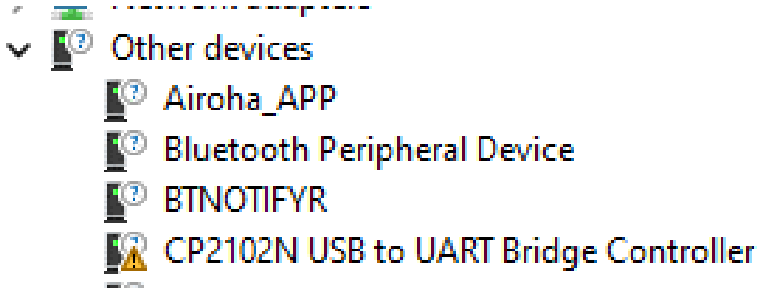
1. Launch Visual Studio Code and navigate to Extensions. Search for "CtrlR" and click Install.



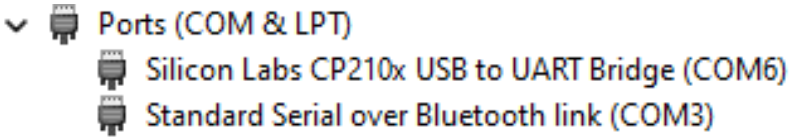
2. Press the Reload button to complete the CtrlR Plugin installation.

2.2 Configuring F1 Starter Kit for CtrlR Plugin on your PC

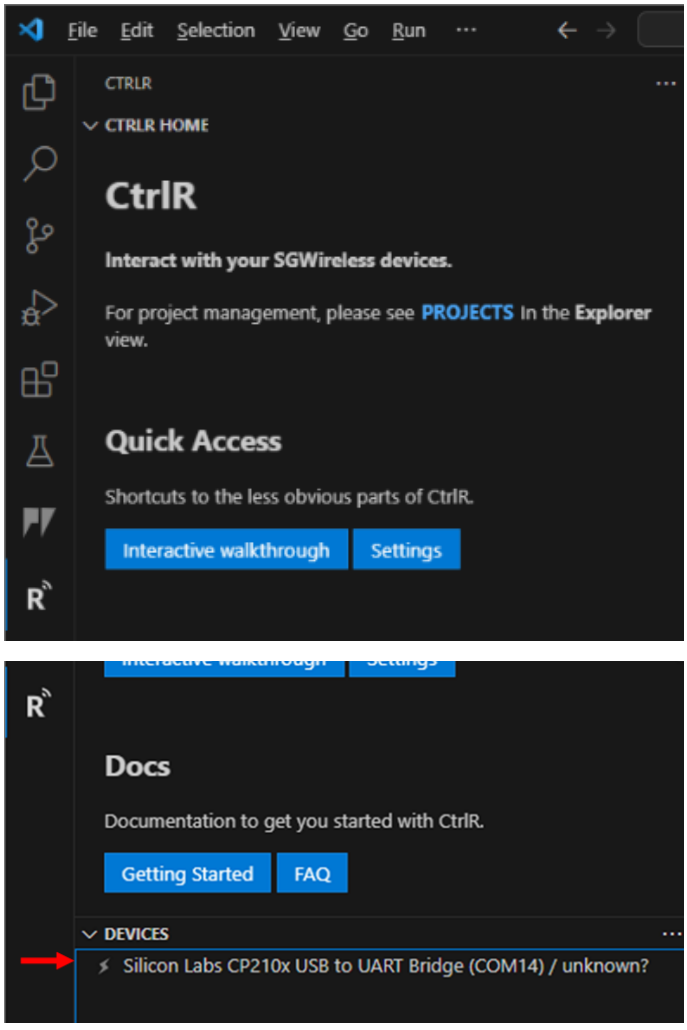
1. Connect your F1 Starter Kit to your PC and turn it on (SW200 switch from OFF to ON).
2. If your PC doesn't have the Silabs CP2102N Virtual COM port driver installed previously, an unknown device will be shown in your PC's Device Manager.



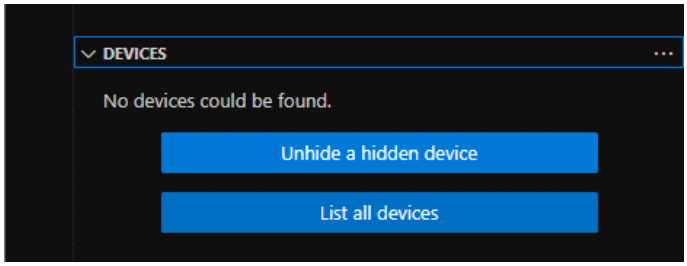
3. Download and install the Virtual COM Port driver.
4. Upon installation of the driver, a new COM port should show up in your PC's Device Manager.



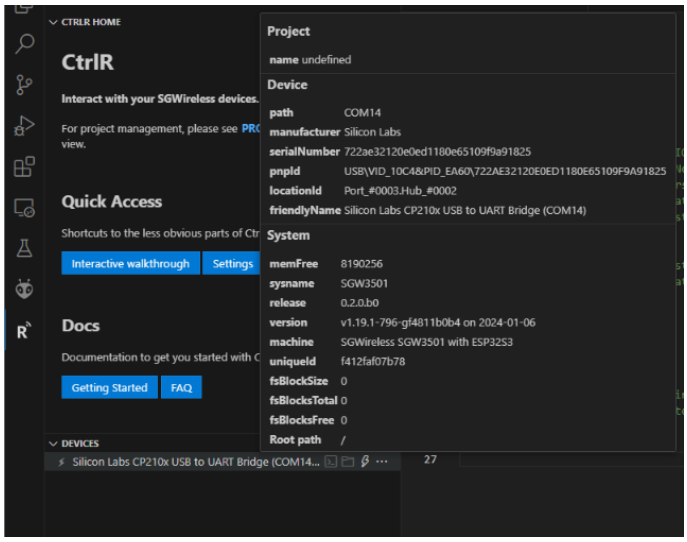
5. Go back to Visual Studio Code and ensure that the CtrlR Plugin has been correctly installed.



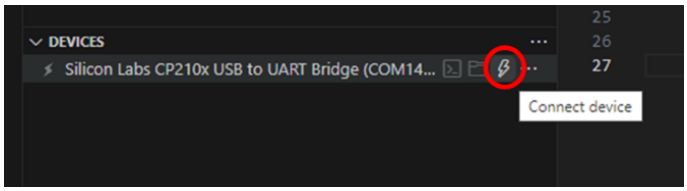
6. Normally, your device will be auto-detected as shown above. If this does not work, click **[List all devices]** in the DEVICES window.



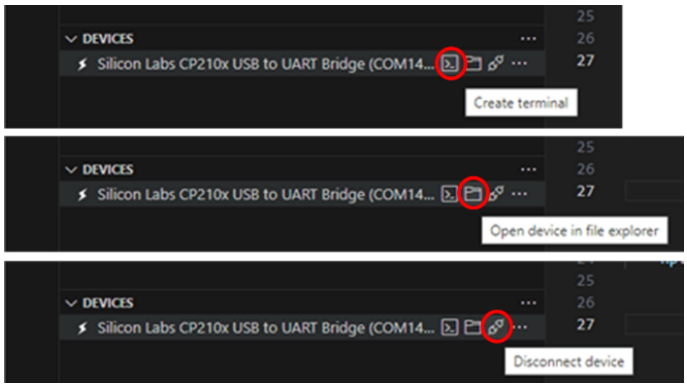
7. If everything is correct, device details will pop up when your mouse pointer is placed on the device:



8. Use the “Connect” button to connect VS Code with the device.



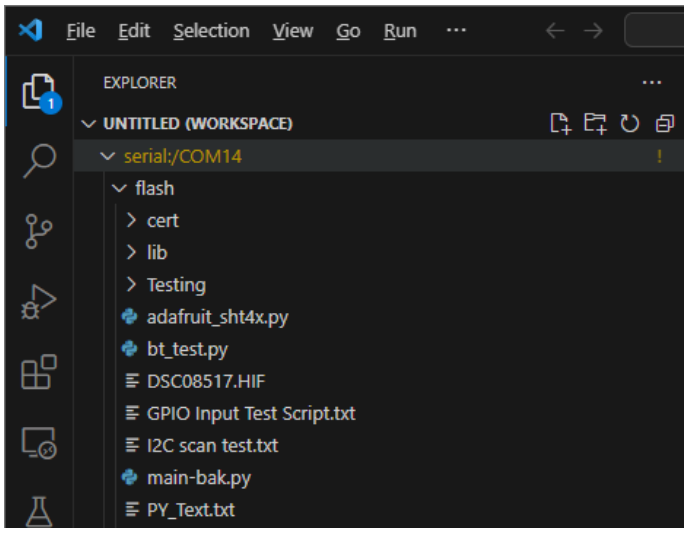
9. Next, you will be able to invoke a terminal, launch file explorer, or disconnect from the device with the 3 buttons.



10. The terminal will let you interact with the device’s MicroPython REPL. Invoking a simple command `os.uname()` will print the firmware version of your device.

```
MicroPython v1.19.1-796-gf4811b0b4 on 2024-01-06; SGWireless SGW3501 with ESP32S3
Type "help()" for more information.
>>>
>>>
>>> os.uname()
(sysname='SGW3501', nodename='SGW3501', release='0.2.0.b0', version='v1.19.1-796-gf4811b0b4 on 2024-01-06', ma
chine='SGWireless SGW3501 with ESP32S3')
>>>
```

11. The file explorer provides a drag-and-drop interface to view, add, and edit the files within your device.



CHAPTER

THREE

ZERO-TOUCH PROVISIONING (ZTP)

Zero-Touch Provisioning (ZTP) automatically performs the following three tasks to set up your Starter Kit seamlessly:

- Get your F1 Starter Kit online using the default LTE or LoRa network, depending on your selected configuration.
- Add your F1 Starter Kit and CAP/T sensor to your Ctrl account for remote monitoring and management.
- Link your F1 Starter Kit and the CAP/T sensor to start viewing its data through your Ctrl page. This feature is not available on ZTP over LoRa.

3.1 Regional Coverage

Start by checking if you're in a region that supports ZTP as listed below:

	F1 Starter Kit Configuration	Regions with Coverage
1	NA Cat M1	Canada, USA, Puerto Rico
2	EU Cat M1	Austria, Belgium, Denmark, Estonia, Latvia, Ireland, Finland, Germany, Hungary, The Netherlands, Norway, Poland, Spain, Sweden, Switzerland, UK, France, Luxembourg, Romania
3	Global Cat M1	Argentina, Brazil, Taiwan, South Korea, Japan, Australia, New Zealand
4	Global NB-IoT	Bulgaria, Croatia, Czech Republic, Greece, Iceland, Italy, Portugal, Slovakia, China, Russia

Note

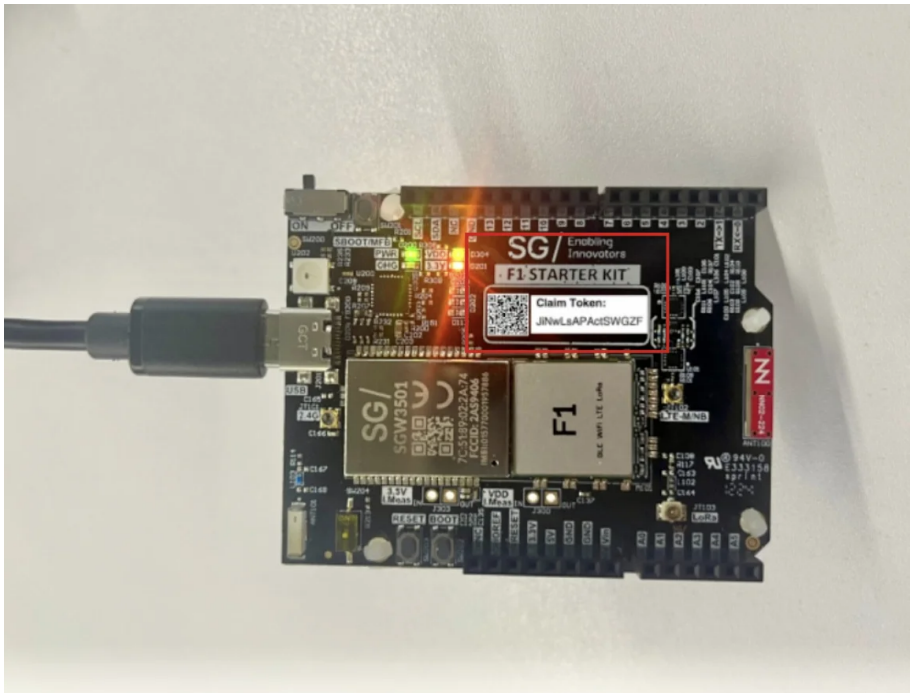
Regions supporting Cat M1 and/or NB-IoT may change without notice. Check here for the latest update.

You can continue with *Manual Provisioning* if ZTP is not supported in your region.

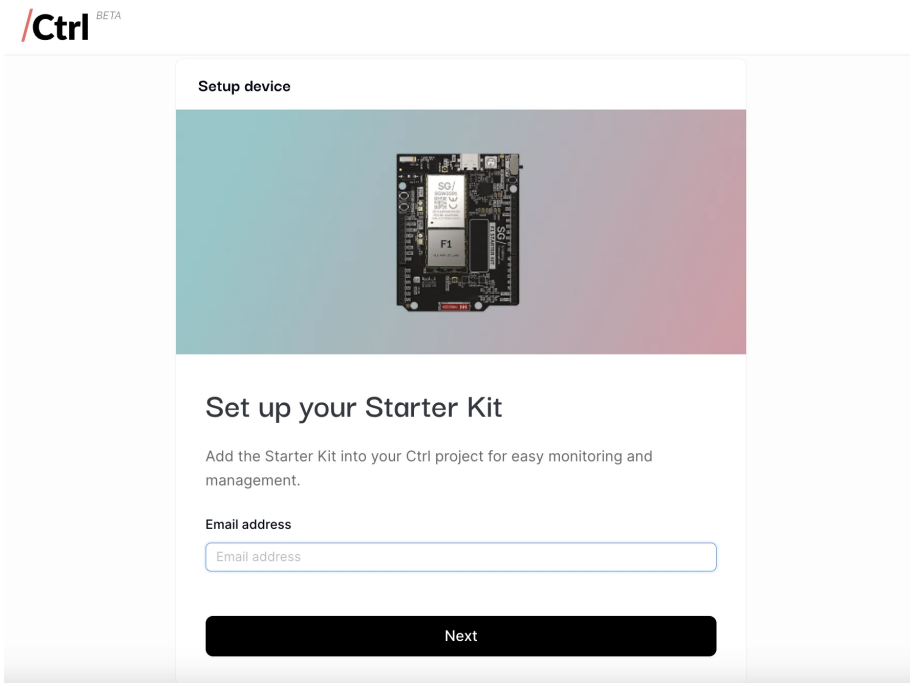
3.2 Provisioning Steps

1. IF you are performing ZTP over LoRa, ensure you have a LoRaWAN gateway within accessible range of your device. The LoRa region setting should be the same as the one you requested for your device pre-configuration during purchase. Make sure that it's also connected to the internet.

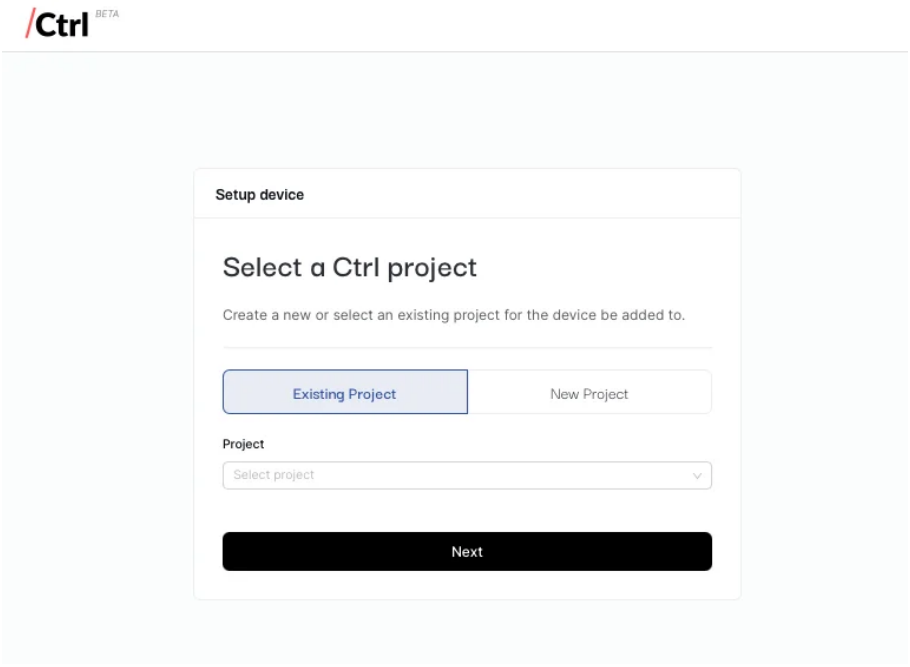
2. Unpack the F1 Starter Kit and scan the QR code on the F1 evaluation board. You'll be directed to the Ctrl IoT Platform.



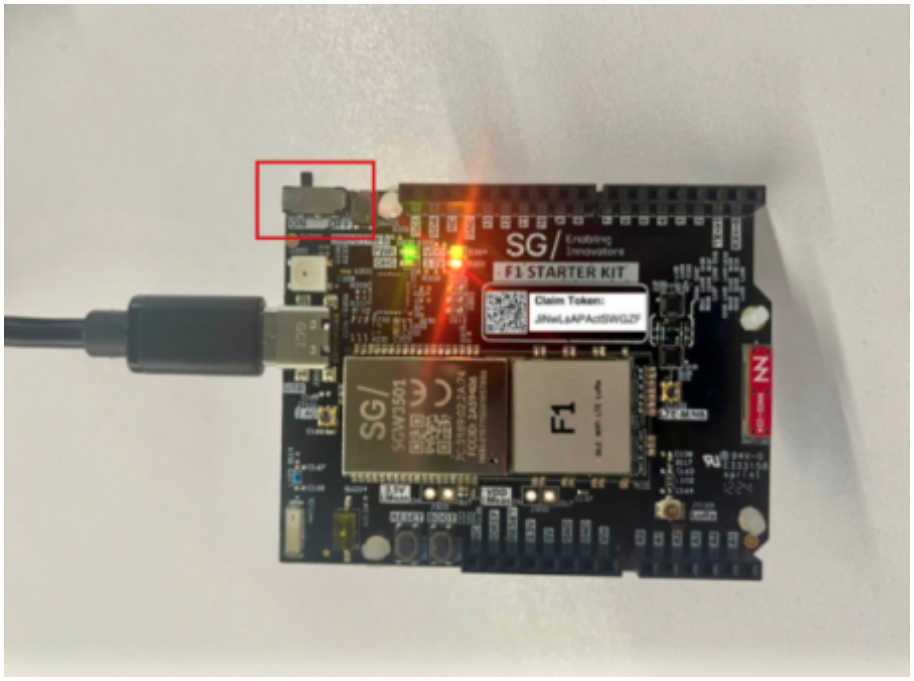
3. Input the email address to be associated with your Ctrl account. Your Starter Kit will be tied to this account. (No other Ctrl accounts can use the same Starter Kit until it is deleted from the account it is tied to.)



4. Select a Ctrl project & Device Template – as this is your first device, you'll land on both "New Project" and "New device template" pages by default and you'll be given system-generated identifiers accordingly. Click "Next". (Device Templates will be important as you scale – more on this later.)



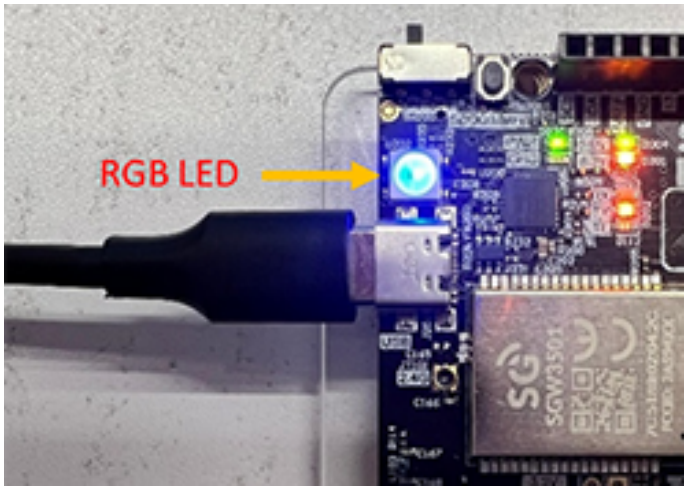
5. Connect the F1 evaluation board to a power source and toggle SW200 from OFF to ON (LEDs will light up).



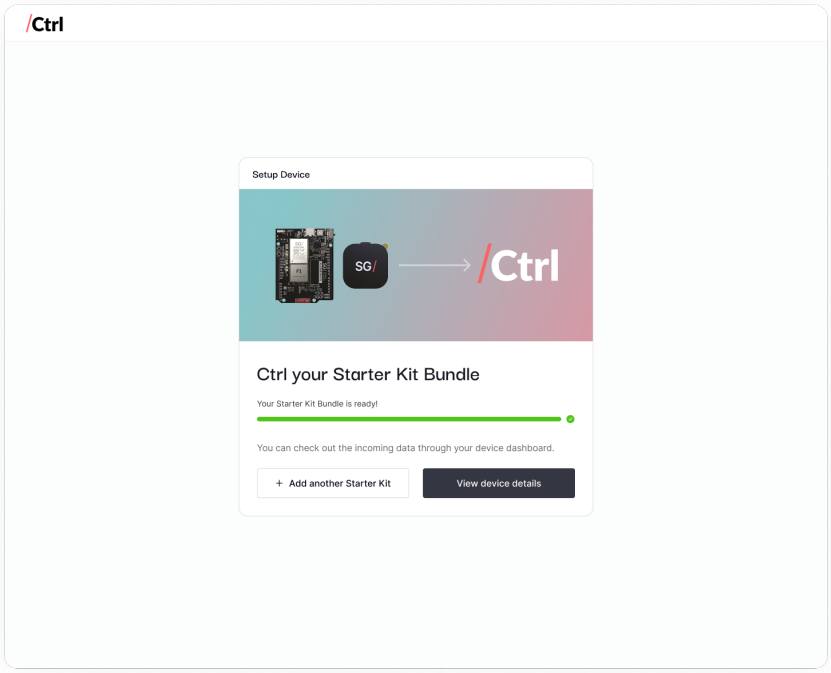
6. (Not applicable for ZTP over LoRa) If you also purchased a CAP/T sensor, you can directly connect your sensor to your F1 during the provisioning flow. Remove the transparent battery cover from the sensor and press the button on the top corner. Notice that the sensor LED will be blinking green.

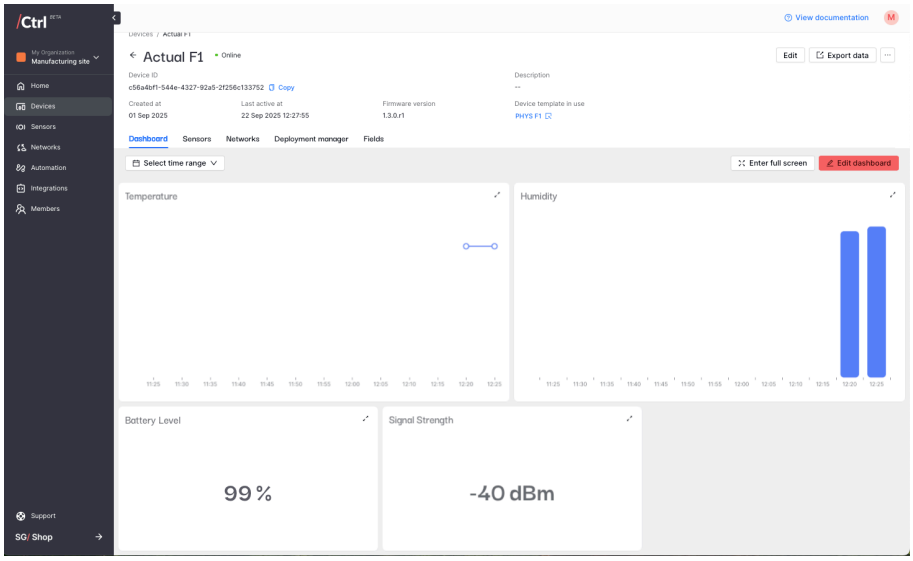
7. Monitor the provisioning status on your screen. You can also check the RGB LED on the F1 evaluation board:

   - **Blinking blue** = registering
   - **Blinking green** = cellular connected
   - **Solid green** = provisioning complete



8. When provisioning is complete, click “Go to device details” on your Ctrl screen to see sensor data coming through.





3.3 Next Steps

From here on, Ctrl will be best used on a PC environment. You can continue your IoT journey by either *programming your F1 Starter Kit* or configuring your dashboards.

CHAPTER

FOUR

MANUAL PROVISIONING FOR F1

Slightly trickier, greater flexibility! These steps allow you to provision your F1 Starter Kit according to your preferences.

- *Configure your network profile*
- *Add your F1 Starter Kit*
- *Configure your F1 Starter Kit Network*
- *Deploy to your F1 Starter Kit*

4.1 Prerequisites

Make sure that you have set up the following software beforehand:

- **CP210x USB to UART Bridge Virtual COM Port driver –** link (only required for Windows and Mac users)
- **Microsoft Visual Studio Code with CtrlR plugin –** link

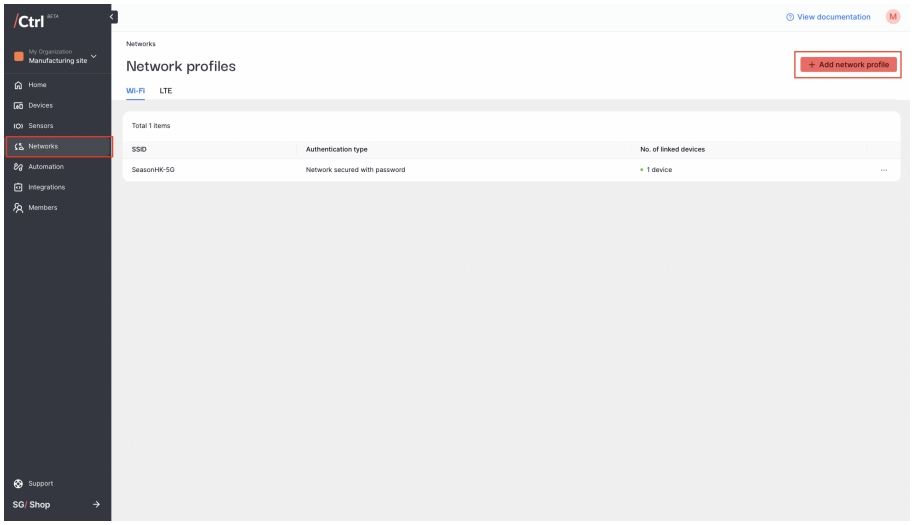
Last but not least, sign in to Ctrl here – create an account or log in, if you already have one.

4.2 Provisioning Steps

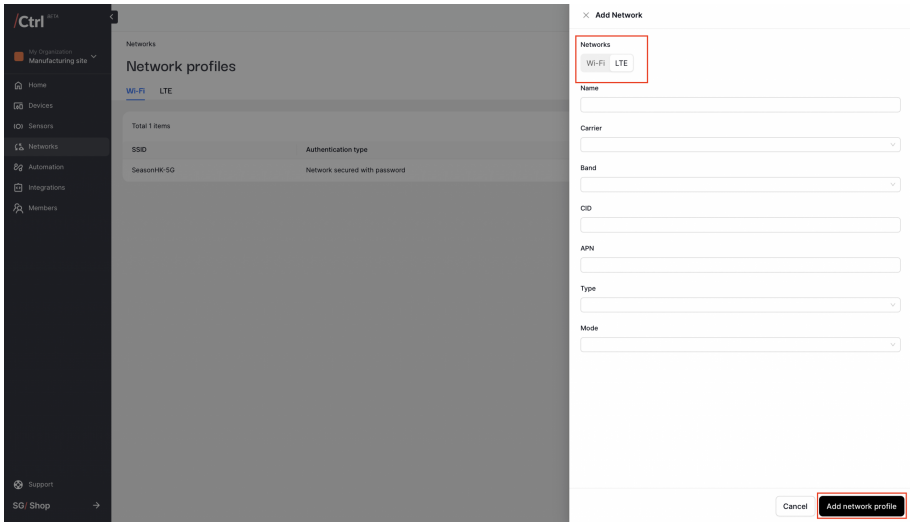
4.2.1 Part 1 – Configure your network profile

Once configured, you can apply these profiles to multiple devices in your project.

1. In the side menu, click “Networks”, then “Add network profile”.

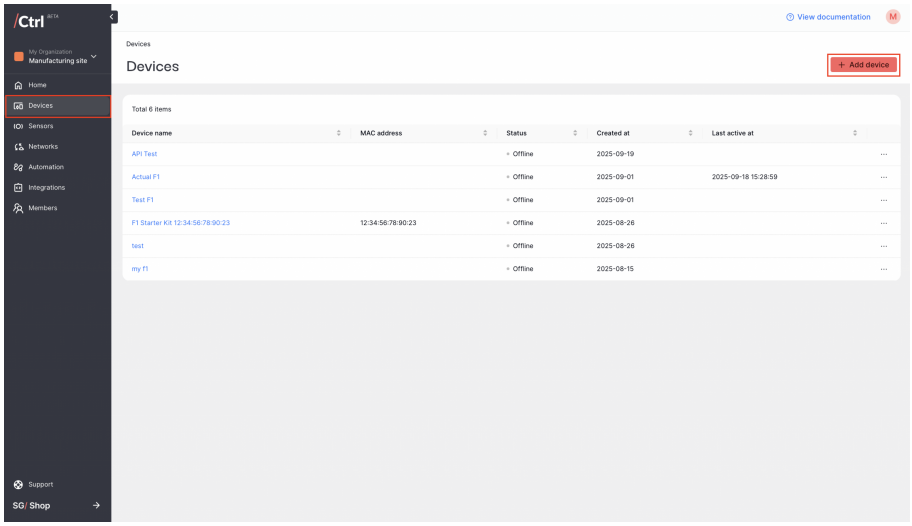


2. Choose “Wi-Fi” or “LTE”, and enter the required network credentials.
3. Click “Add network profile” – you’ll see the added network in “Networks”.

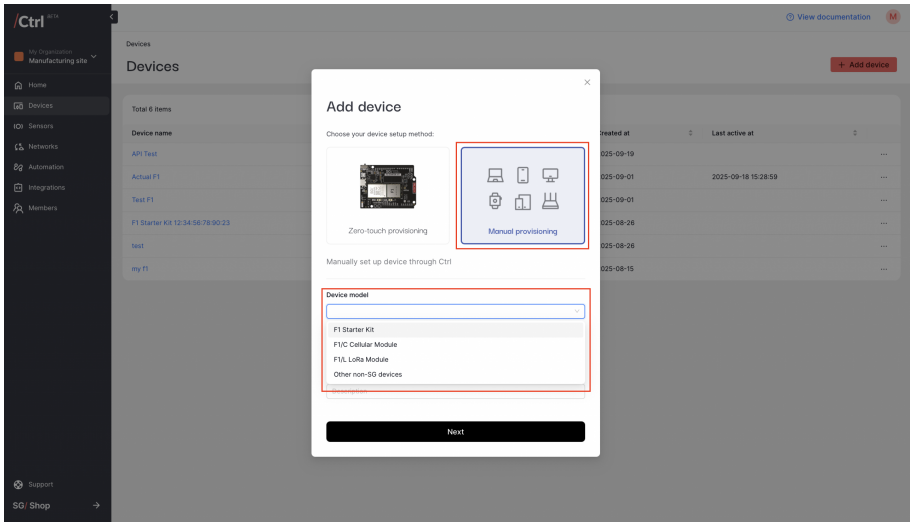


4.2.2 Part 2 – Add your F1 Starter Kit

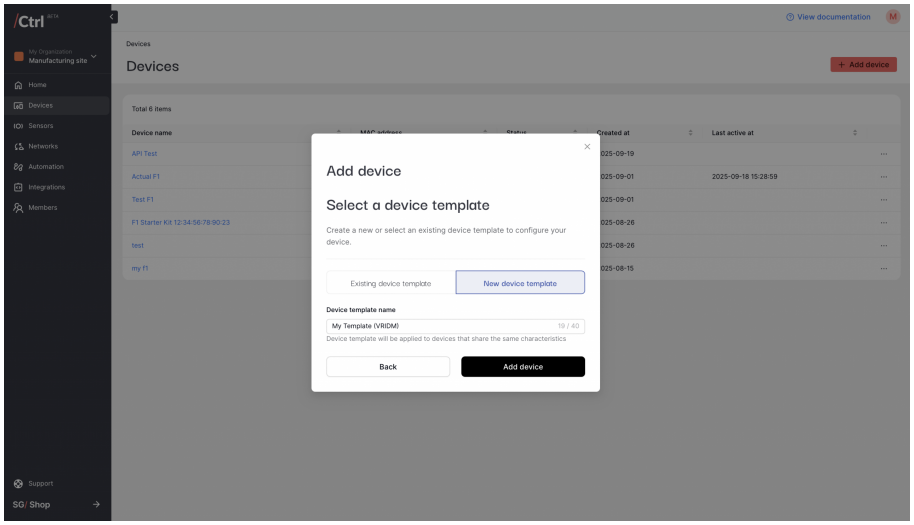
1. In the side menu, click “Devices”, then “Add device”.



2. Choose “Manual provisioning”.
3. Choose “SG F1 Starter Kit” as your device model. Enter a name and description.

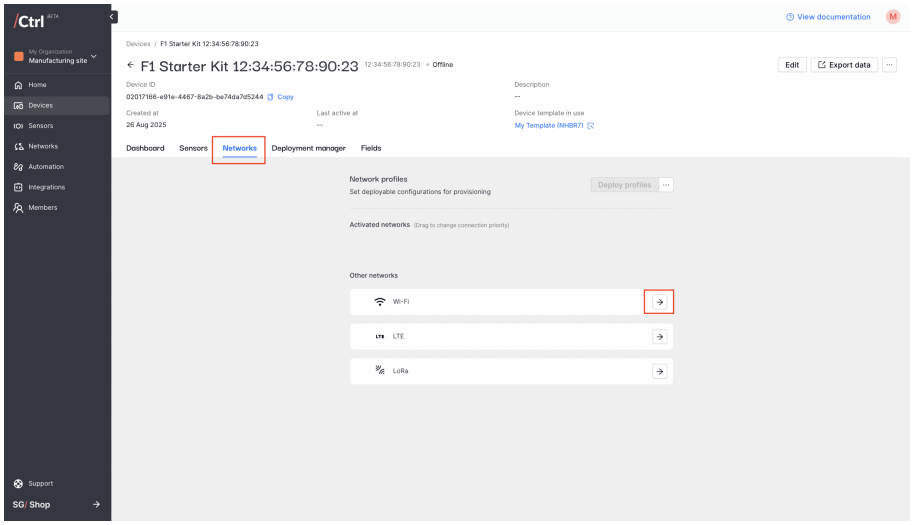


4. Select a Device Template – as this is your first device, you’ll land on “New device template” by default and you’ll be given a system-generated template identifier. (Device Templates will be important as you scale – more on this later!)
5. Click “Add device” to complete the device creation flow.

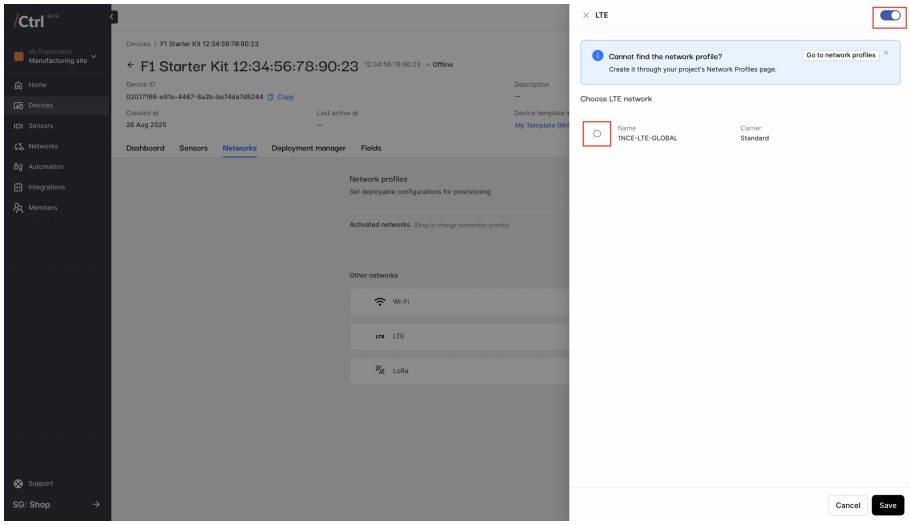


4.2.3 Part 3 – Configure your F1 Starter Kit Network

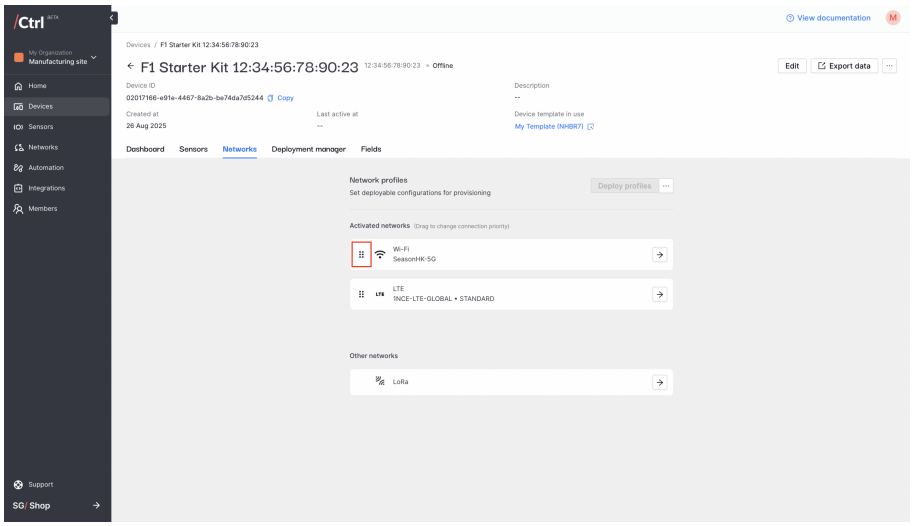
1. Click “View device details” to view your newly-created device page.
2. Click on the “Networks” tab and choose the network type that you want to activate.



3. Toggle the target connection to activate it, then select the target network profile.

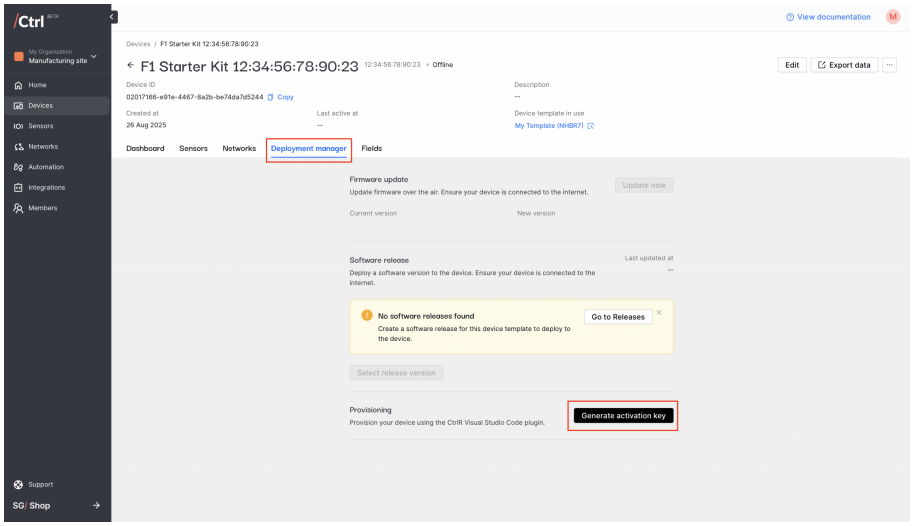


4. If you're enabling more than one network type, hold and drag the network type to adjust the priority.

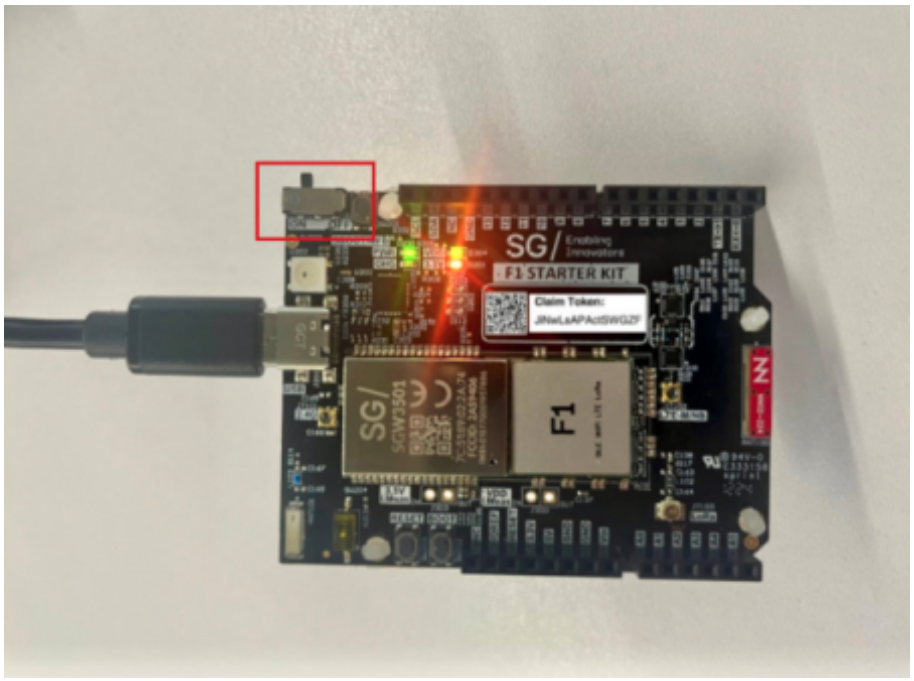


4.2.4 Part 4 – Deploy to your F1 Starter Kit

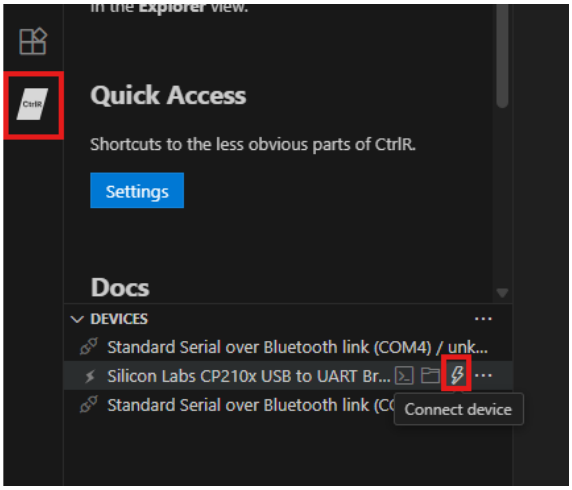
1. In the device's "Deployment Management" tab, click "Generate activation key". You'll need this key in a bit.



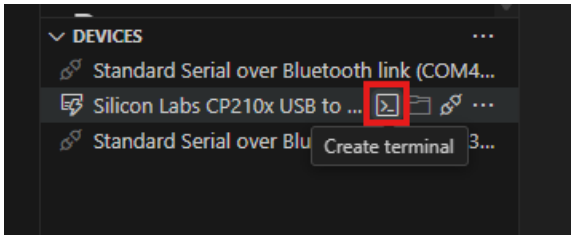
2. Connect the F1 evaluation board to a power source and toggle SW200 from OFF to ON (LEDs will light up).



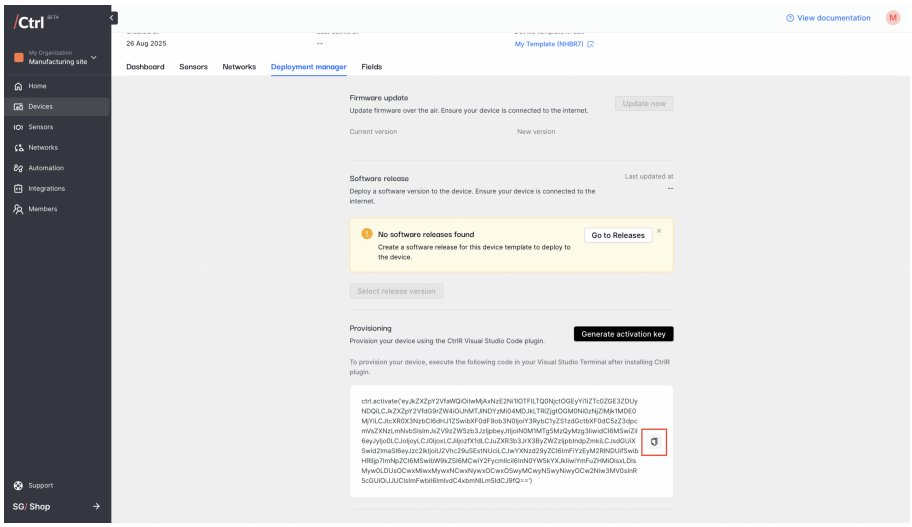
3. Open the CtrlR plugin in Visual Studio Code and click "Connect device" on the detected device.



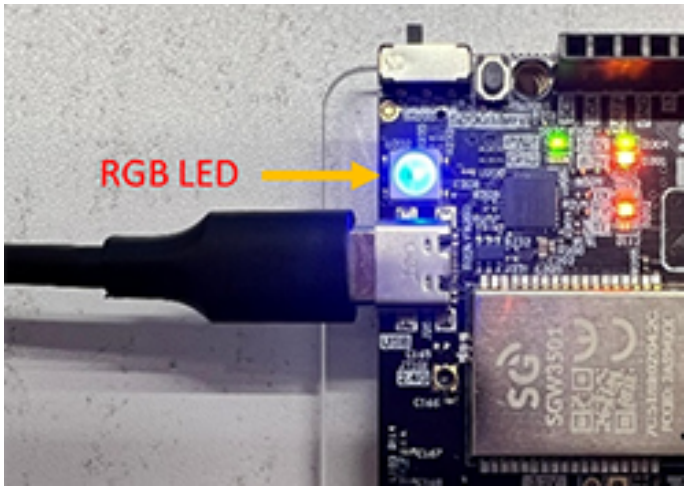
4. Invoke a terminal to access the REPL interface by clicking the “Create terminal” icon.

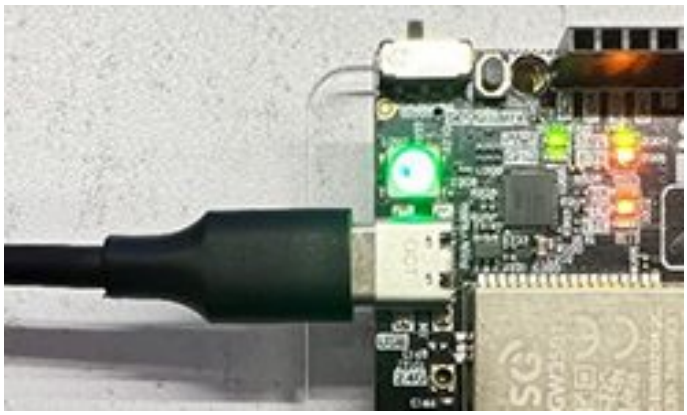


5. Remember the key you generated earlier? Copy it from your Ctrl “Deployment Manager”.
6. Paste it into the terminal, and press Enter.



7. Monitor the RGB LED on the F1 evaluation board:
    - **Blinking blue** = registering
    - **Blinking green** = cellular connected
    - **Solid green** = provisioning complete





8. When provisioning is complete, you're now ready to *link your first CAP/T Sensor* to start collecting data.

CHAPTER

FIVE

YOUR FIRST SENSOR DATA

The final step to your set-up is also the most important – getting data.

Let's start with the CAP/T Sensor which comes with temperature and humidity sensors.

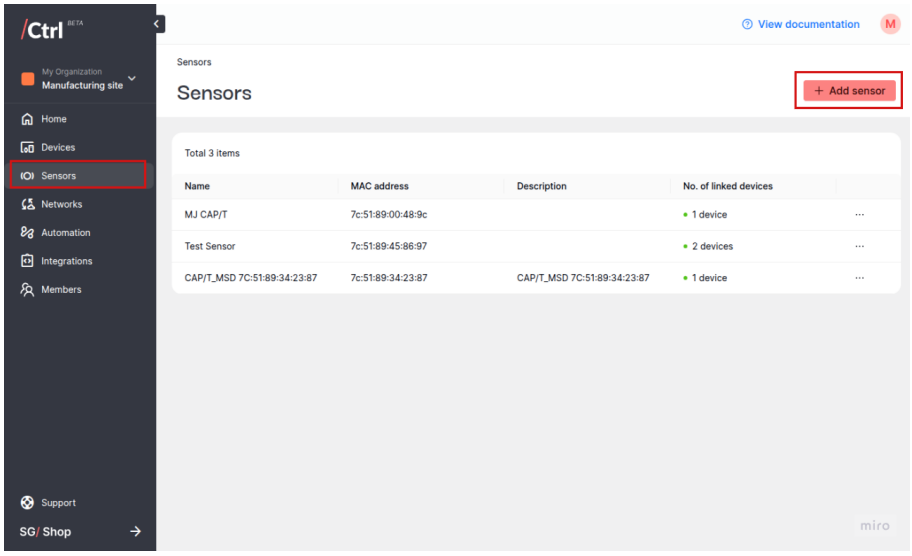
- *Add your Sensor*
- *Link your Sensor*
- *Map Sensor data*

Note

- You can skip this step if you provisioned your device through ZTP.
- This sensor configuration method is only applicable for CAP/T sensors at the moment. Other sensors or peripherals need to be configured directly on the device.

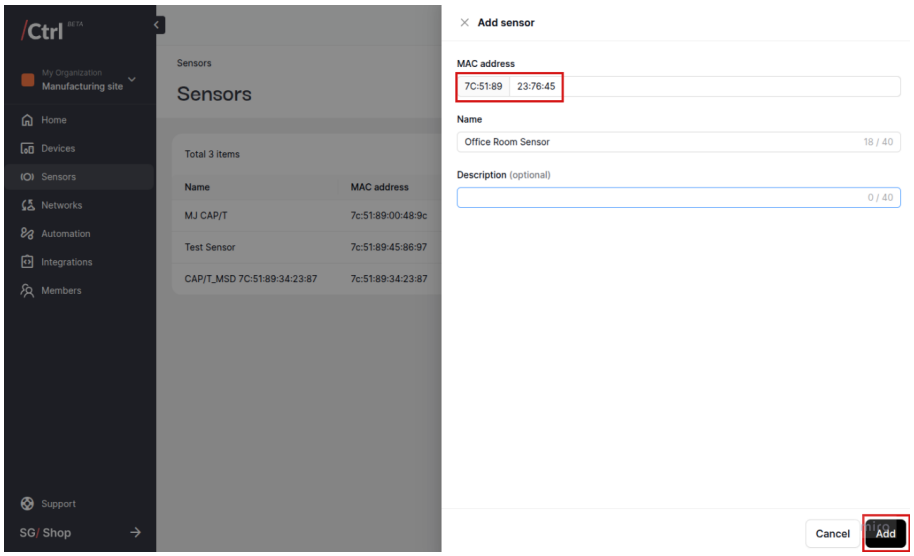
5.1 Add your CAP/T Sensor

1. In the side menu, click "Sensors", then "Add sensor".



2. Enter the last 6 digits of the MAC address of your CAP/T Sensor, which can be found on the Sensor's battery lid. The first 6 digits have been automatically filled for your convenience.

3. Name the Sensor and add a description as needed.
4. Click “Add sensor”.



5.2 Link your CAP/T Sensor to your F1 Starter Kit

One CAP/T Sensor can send data to more than one F1 Starter Kit, giving you the flexibility to use the same set of data in multiple ways for different projects.

This is done by linking the Sensor to the required Kit(s), as follows:

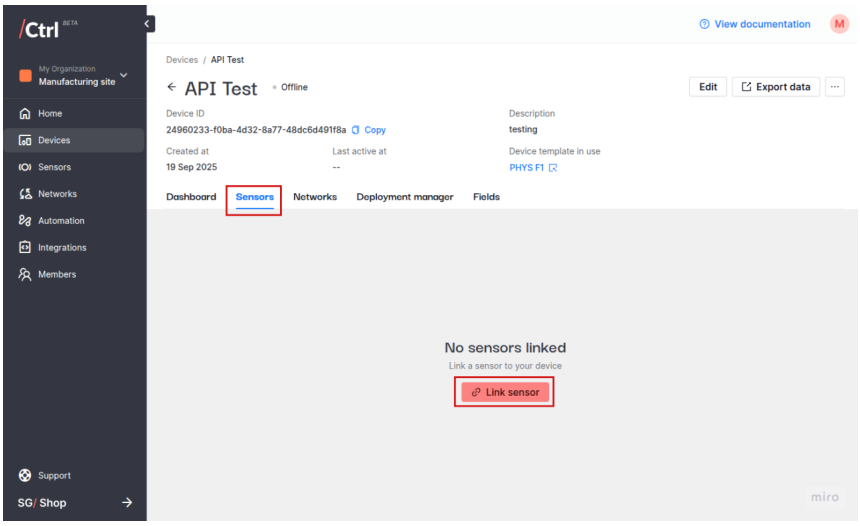
1. Since we are attempting to update the sensor configuration remotely to the Starter Kit, ensure that your Starter Kit has a stable Internet connection. Otherwise, you can regenerate the configuration key and run it through your CtrlR Visual Studio Code plugin after completing the “Map your CAP/T Sensor data” steps.
2. Remove the transparent film from the CAP/T Sensor. Ensure that the sensor LED is blinking green.

3. Initiate the device linkage through one of the two methods:

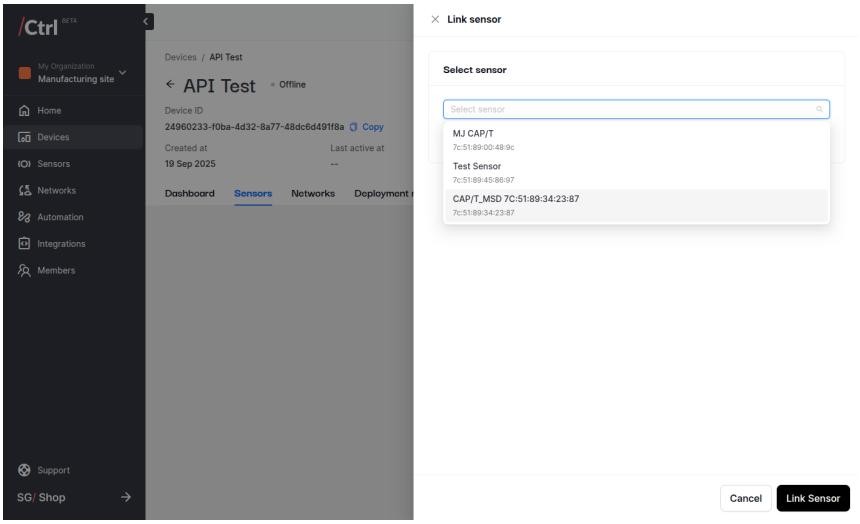
   Through "Devices":

   a. Find the target device on your device list, and go to the "Sensors" tab.

   b. Click "Link sensor" and select the target Sensor.



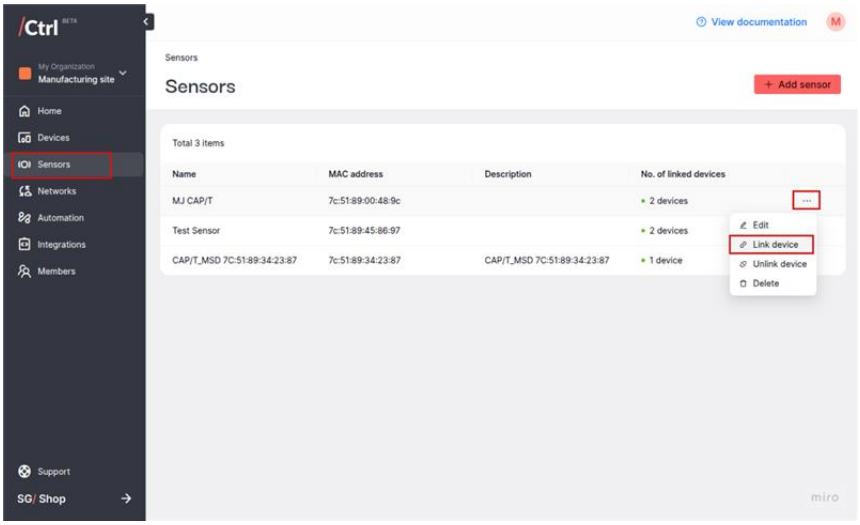
   c. Select a sensor to be linked.



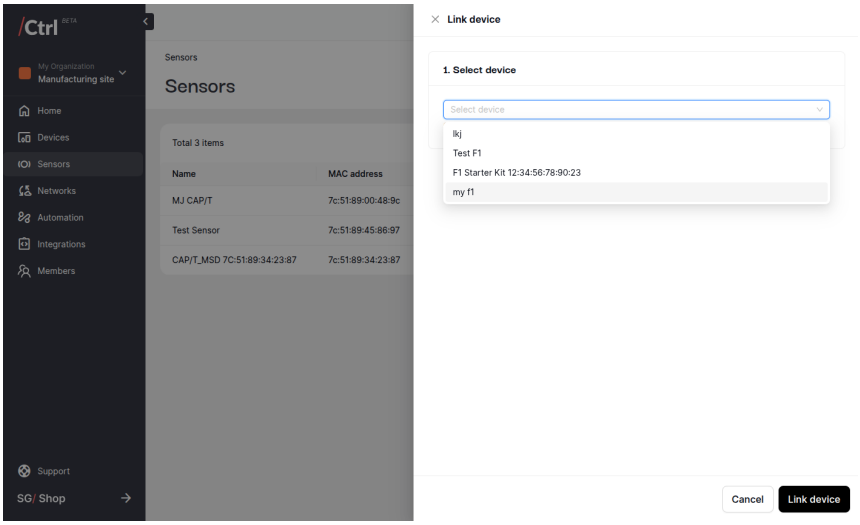
Through "Projects":

a. Go to the "Sensors" tab, and find the target Sensor. Click the "…" button.

b. Click "Link device", then select the target Kit.



c. Select a device to be linked.



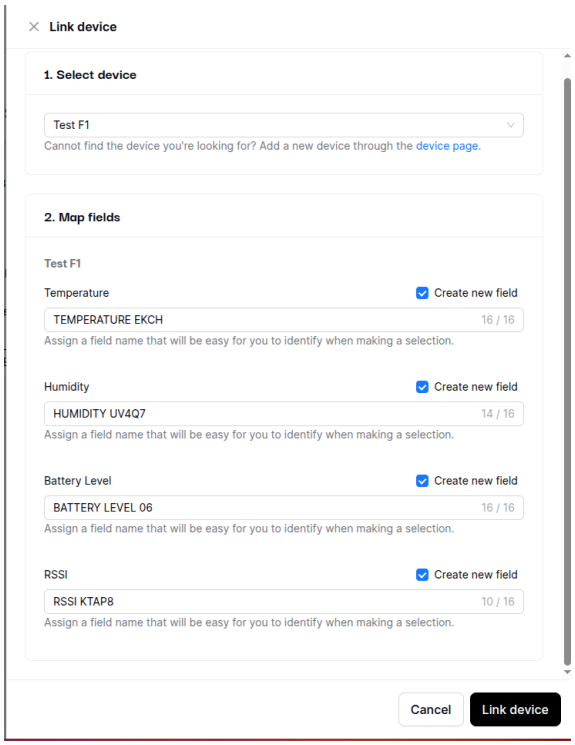
5.3 Map your CAP/T Sensor data

Sensor data is stored in "Device Fields" – consider them as individual buckets that store different types of sensor data.

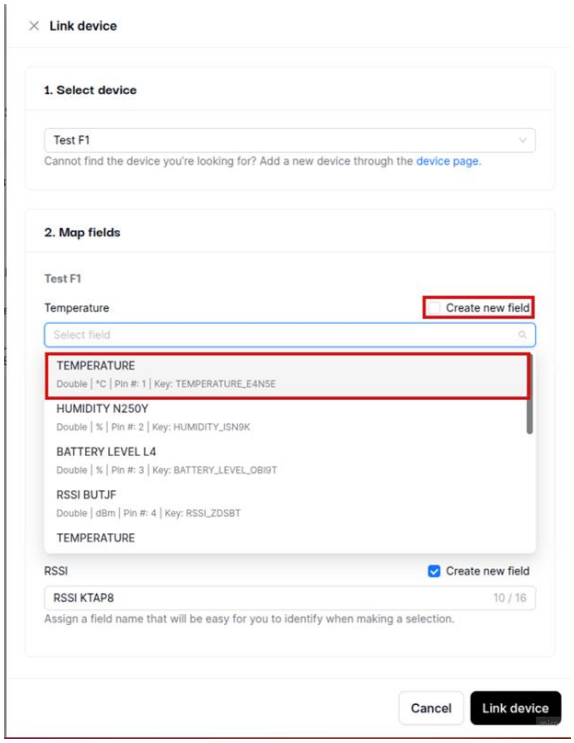
1. After you select a device or sensor to be linked, the field mapping options will be displayed.

   By default, a new Device Field is created for each type of telemetry data received from CAP/T sensors. You will see four Device Fields automatically created for the CAP/T Sensor:

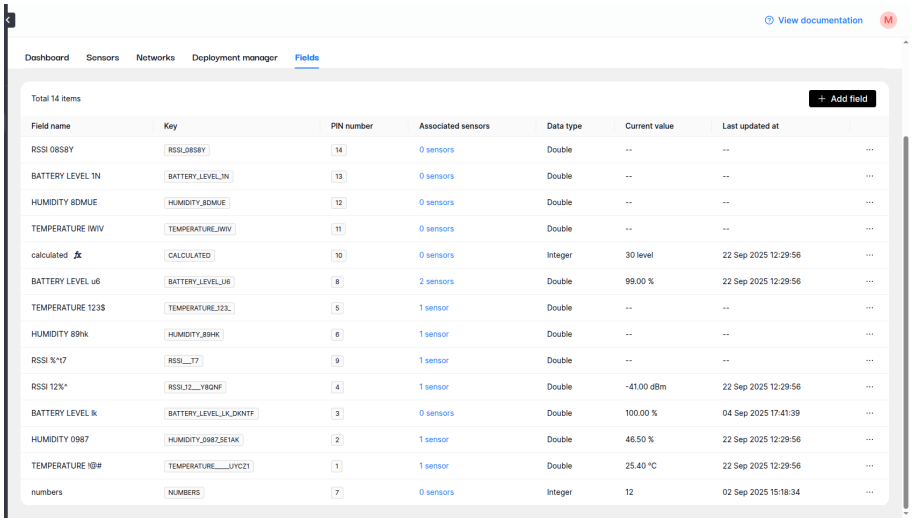
   - Temperature
   - Humidity
   - Battery Level
   - RSSI



2. You can also disable the "Create new field" option to map the data to existing compatible Device Fields.



3. Click “Link device” when you’re done.
4. If your device is not currently connected to the internet, *deploy the sensor configuration* by regenerating the activation code and running it through your CtrlR Visual Studio Code plugin.
5. View the incoming data by checking the latest value columns on your device’s “Fields” tab.



5.4 Next Steps

You can continue your IoT journey by either configuring your dashboards or *programming your F1 Starter Kit*.

CHAPTER

SIX

YOUR FIRST F1 CODE

The F1 smart module has MicroPython pre-installed as its operating system (OS), which is equipped with REPL. REPL stands for Read-Eval-Print Loop – an interactive interpreter mode that allows you to input code, execute it, and immediately see the results.

Using the CtrlR Plugin, open and connect a device – or use a serial terminal (PuTTY, screen, picocom, etc.). Upon connecting, there should be a blank screen with a flashing cursor. Press Enter and a MicroPython prompt should appear, i.e. `>>>`. Let's make sure it is working with the obligatory test:

```
>>> print("Hello F1!")
Hello F1!
```

In the example above, the `>>>` characters should not be typed. They are there to indicate that the text should be placed after the prompt. Once the text has been entered (`print("Hello F1!")`) and Enter has been pressed, the output should appear on screen, identical to the example above.

Basic Python commands can be tested out in a similar fashion.

If this is not working, try either a hard reset or a soft reset; see below.

6.1 Resetting the Device

If something goes wrong, the device can be reset with two methods: hard reset and soft reboot.

6.1.1 Hard reset

Press the RESET button on the F1 Starter Kit (or apply a high signal to the F1 module reset signal). The F1 module will reset itself.

Please notice that any serial/COM port connection will reset and may need to be manually reconnected if the auto-connect option is not enabled in the CtrlR plugin.

After reset, the normal MicroPython boot message will appear in the terminal of the CtrlR plugin or other serial terminals that have been connected, as shown below:

```
>>> ESP-ROM:esp32s3-20210327
Build:Mar 27 2021
rst:0x1 (POWERON),boot:0x8 (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:2
load:0x3fce3830,len:0xfac
load:0x403c9700,len:0xd3c
load:0x403cc700,len:0x2dd8
entry 0x403c9964
== micropython normal mode
-- start ctrl client
MicroPython v1.19.1-796-gf4811b0b4 on 2024-05-02; SGWireless SGW3501-F1-StarterKi
t with ESP32S3
Type "help()" for more information.
>>>
```

6.1.2 Soft reboot

Press **Ctrl-D** at the MicroPython prompt to perform a soft reset. A soft reboot message will appear in the terminal of the CtrlR plugin or other serial terminals that have been connected, as shown below:

```
MPY: soft reboot
== micropython normal mode
-- start ctrl client
MicroPython v1.19.1-796-gf4811b0b4 on 2024-05-02; SGWireless SGW3501-F1-StarterKi
t with ESP32S3
Type "help()" for more information.
>>>
```

CHAPTER

SEVEN

HARDWARE

The following pages contain all information relating to each product: pinouts, spec sheets, relevant examples and notes.

7.1 F1 Smart Module

Datasheet — March 2024 V1.1

`Download PDF Datasheet`

7.1.1 Introduction

The F1 Smart Module is a compact OEM module equipped with BLE, Wi-Fi, LoRa(WAN), and LTE CAT-M1/NB1/NB2 to support various connectivity needs. Running on a MicroPython-programmable microcontroller with a no-barrier entry into the SG Wireless Ctrl. Cloud Platform, the module enables truly limitless IoT application development with multi-network creation flexibility and rapid scaling capacity.

Key Features

- Multi-connectivity:
  - WiFi 802.11b/g/n (2.4 GHz)
  - Bluetooth BLE 5.0
  - Cellular LTE-CAT M1/NB1/NB2
  - LoRa(WAN) 868 MHz (EU) and 915 MHz (US)
- Powerful Espressif ESP32-S3 CPU
- MicroPython programmable with 27 IOs on module pads
- SMT-friendly semi-hole pins at module edges
- Operating temperature: –40 °C to 85 °C
- Compact size: 42.67 mm × 17.73 mm × 3.50 mm


SG/
SGW3501
7C:51:89:00:00:00
FCCID: 2AS9406
IMEI: 015770001978932
F1
BLE WiFi LTE LoRa

Order Information

Part Number	Description
SGW3501	F1 Smart Module: BLE, WiFi, LoRa, LTE
SGW3401	F1/C Cellular Module: BLE, WiFi and LTE
SGW3201	F1/L LoRa Module: BLE, WiFi and LoRa

7.1.2 General Features

Feature Specifications

Component	Specifications
CPU	Xtensa® dual-core 32-bit LX7 microprocessor, up to 240 MHz On-chip 384 KB ROM and 512 KB SRAM, on-board 8 MB PSRAM and 16 MB Flash Deep Sleep Mode: 10 µA
WiFi/BLE	Espressif ESP32-S3 on-chip RF frontend WiFi: IEEE 802.11b/g/n (2.4 GHz band); Data Rate: 1 M up to 54 Mbps (MCS7); Max Tx Power: 20 dBm BLE: Bluetooth LE 5.0, Bluetooth mesh; Data Rate: 125 kbps to 2 Mbps; Max Tx Power: 20 dBm
LTE	Sequans Monarch2 GM02S for CAT-M1, CAT-NB1 and CAT-NB2 support LTE CAT-M1/NB1/NB2 transmit power up to +23 dBm PTCRB and GCF 1.3 3GPP release 13 compliant; Operator Approval: Verizon, AT&T, T-Mobile, Vodafone, Orange
LoRa(WAN)	Semtech SX1262 RF transceiver, 868/915 MHz LPWAN Module TX Power: Up to +22 dBm; Sensitivity: −127 dBm LoRaWAN stack – Class A and Class C Device

7.1.3 Pinout & Pin Definitions

`Module Pinout Diagram - PDF`

Kindly refer to the *Mechanical Specifications* section for more details on the physical dimensions.

Pin Definitions

Pin Number	Pin Name	MCU Pin	LTE Module Pin	Type	Description
R4	GND			Power	Ground signal
R6	GND			Power	Ground signal

continues on next page

Table 1 – continued from previous page

Pin Number	Pin Name	MCU Pin	LTE Module Pin	Type	Description
R7	GND			Power	Ground signal
R9	USIM_C		SIM0_CLK	Analog I/O	USIM interface I/O to GM02S
R10	USIM_IO		SIM0_IO	Digital I/O	USIM interface I/O to GM02S
R12	GND			Power	Ground signal
R21	GND			Power	Ground signal
R22	RESET	CHIP_PU		Analog I/O	Reset pin to ESP32-S3 for module reset
R23	P0	U0RXD		Analog I/O	UART0 RXD to ESP32-S3
R24	P1	U0TXD		Analog I/O	UART0 TXD to ESP32-S3
R25	P2	GPIO0		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
R26	P3	GPIO4		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
R27	P4	MTDO		Digital I/O	Digital I/O to ESP32-S3
R28	P5	GPIO5		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
*R29	P6	GPIO6			Reserved – Leave floating, do not connect
R30	P7	GPIO3		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
R31	P8	GPIO46		Digital I/O	Digital I/O to ESP32-S3
R32	P9	GPIO45		Digital I/O	Digital I/O to ESP32-S3
R33	P10	MTCK		Digital I/O	Digital I/O to ESP32-S3
R34	P11	GPIO11		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
R35	P12	GPIO21		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3

continues on next page

Table 1 – continued from previous page

Pin Number	Pin Name	MCU Pin	LTE Module Pin	Type	Description
R36	GND			Power	Ground signal
R37	PEXT1	GPIO1		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
R38	PEXT2	GPIO12		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
M39	GND			Power	Ground signal
L39	BLE/WI			RF I/O	RF interface to ESP32-S3 for BLE and/or Wi-Fi interface
K39	GND			Power	Ground signal
A38	PEXT3	GPIO14		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
A37	PEXT4	GPIO13		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
A36	GND			Power	Ground signal
A35	GND			Power	Ground signal
A34	P13	GPIO20		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3 / USB OTG D+
A33	P14	GPIO19		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3 / USB OTG D−
A32	P15	GPIO38		Digital I/O	Digital I/O to ESP32-S3
A31	P16	GPIO41		Digital I/O	Digital I/O to ESP32-S3
A30	P17	GPIO2		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
A29	P18	GPIO10		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3

continues on next page

Table 1 – continued from previous page

Pin Number	Pin Name	MCU Pin	LTE Module Pin	Type	Description
A28	P19	GPIO15		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
A27	P20	GPIO16		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
A26	P21	GPIO17		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
A25	P22	GPIO18		Analog I/O or Digital I/O	Analog I/O or Digital I/O to ESP32-S3
A24	P23	GPIO42		Digital I/O	Digital I/O to ESP32-S3
A23	+3.3V	VDD3P3 VDD3P3 VDD3P3 VDDA		Power	Voltage supply to ESP32-S3 and module main circuit
A22	GND			Power	Ground signal
A21	+1.8V_C	VDD_SF		Power	Voltage supply VDD_SPI to ESP32-S3 for SPI flash and PSRAM
A13	GND			Power	Ground signal
A12	GND			Power	Ground signal
A10	USIM_R		SIM0_RST	Digital I/O	USIM interface I/O to GM02S
A9	USIM_V		SIM0_VC(	Power	USIM voltage supply to GM02S
A7	GND			Power	Ground signal
A6	GND			Power	Ground signal
A4	+VBATT		VBAT	Power	Voltage supply to GM02S
E1	GND			Power	Ground signal
F1	LORA_A			RF I/O	RF interface to SX1262 for LoRa interface
G1	GND			Power	Ground signal
J1	GND			Power	Ground signal
K1	LTE_AN		LTE_ANT	RF I/O	RF interface to GM02S for LTE CAT-M1/CAT-NB1/CAT-NB2 interface
L1	GND			Power	Ground signal

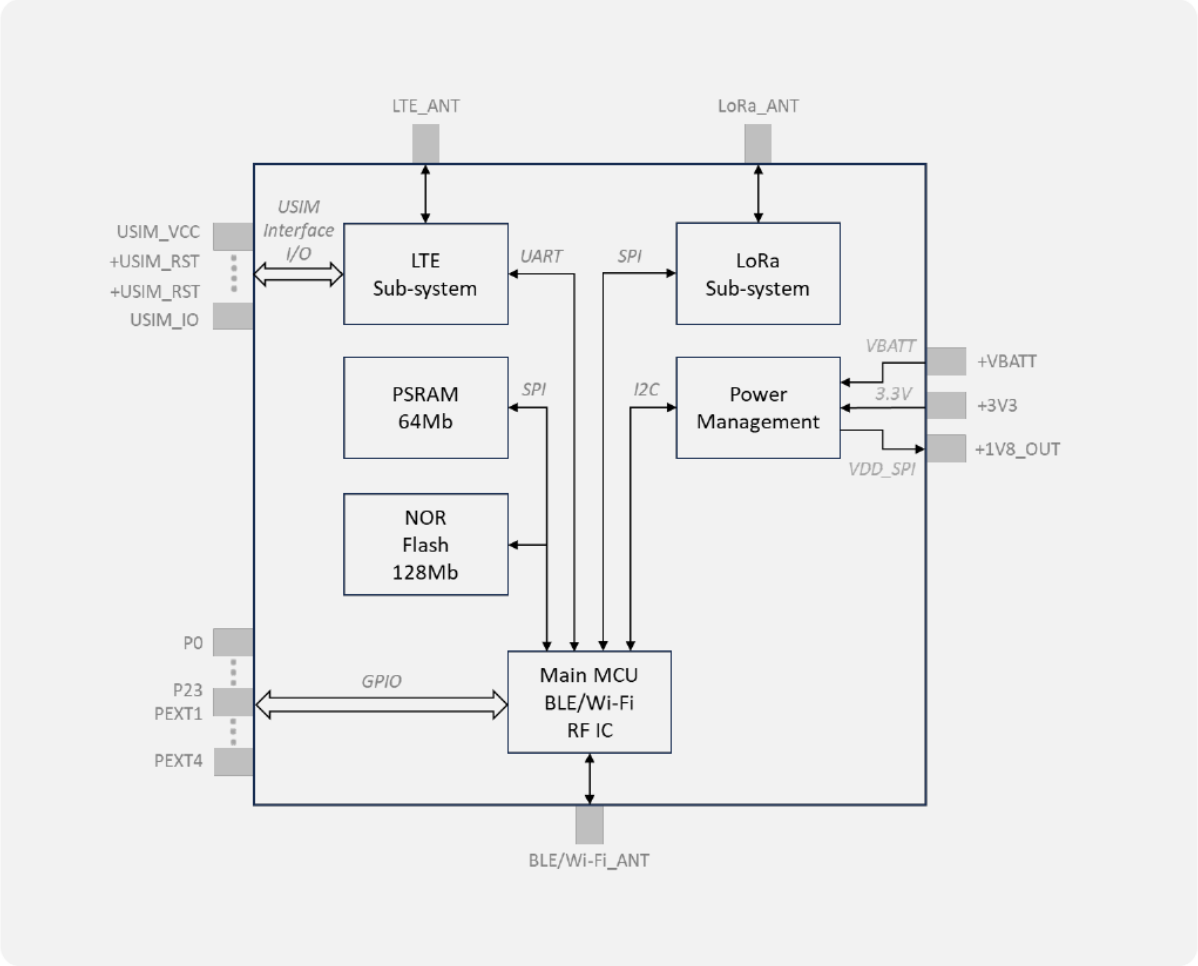


Fig. 1: *Figure 1: F1 Smart Module Block Diagram*

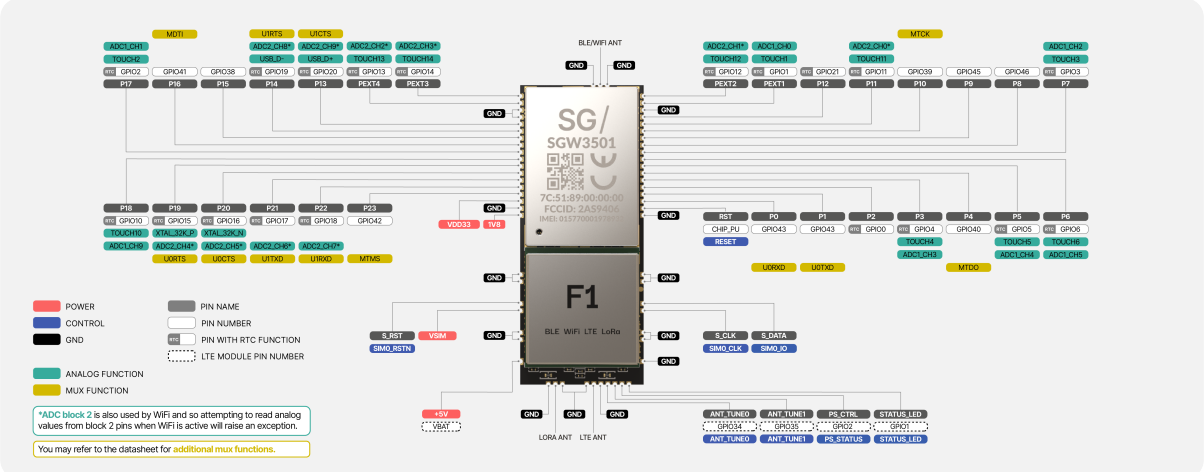


Fig. 2: *Figure 1: F1 Smart Module Pinout Diagram*

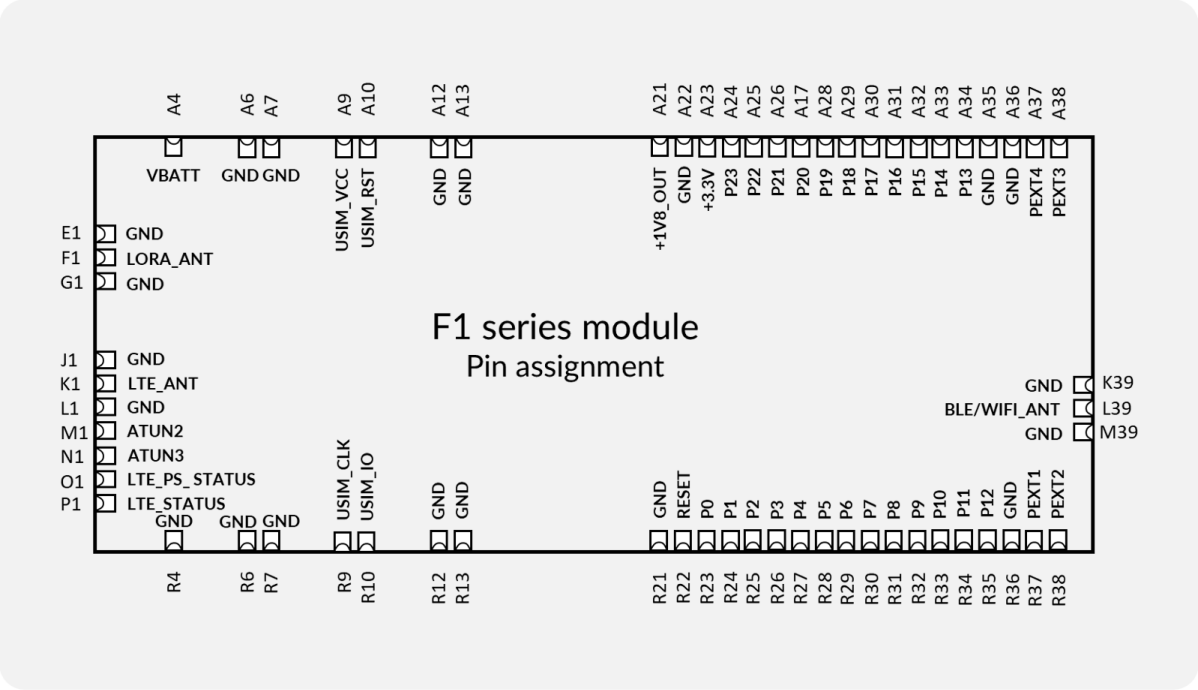


Fig. 3: *Figure 2: F1 Smart Module Pin-out (Top View)*

7.1.4 Electrical Specifications

Rating & Operating Conditions

Table 1: Absolute Rating and Operating Conditions Specifications

Symbol	Parameter	Min	Typ	Max	Unit
Absolute Rating					
+VBATT	Supply voltage to Sequans GM02S LTE module		5.0	5.8	V
+3V3	Supply voltage to Espressif ESP32-S3 and module main circuit	3.0	3.3	3.6	V
+1V8_OUT*	SPI supply voltage (output) of SPI flash and PSRAM for decoupling capacitor connection		1.8	2.3	V
T(OPR)	Operating temperature	–40		85	°C
Operating Conditions					
+VBATT	Supply voltage to Sequans GM02S LTE module	2.5	5.0	5.5	V
+3V3	Supply voltage to Espressif ESP32-S3 and module main circuit	3.2	3.3	3.4	V
+1V8_OUT*	SPI supply voltage (output) of SPI flash and PSRAM for decoupling capacitor connection	1.7	1.8	1.9	V
CPU IO *(3.3 V power domain, VDD = 3.3 V)*					
VIH	Input high voltage for GPIOs	0.75 × VDD		VDD + 0.3	V
VIL	Input low voltage for GPIOs	–0.3		0.25 × VDD	V
VOH	Output high voltage for GPIOs	0.8 × VDD			V
VOL	Output low voltage for GPIOs			0.1 × VDD	V
Radio IO *(1.8 V power domain)*					
VIH	Input high voltage for GPIOs	1.26		1.8	V
VIL	Input low voltage for GPIOs	0		0.54	V
VOH	Output high voltage for GPIOs	1.44		1.8	V
VOL	Output low voltage for GPIOs	0		0.36	V

* The +1V8_OUT pin is to connect an external capacitor to the module internal SPI flash and PSRAM for a more robust VDD_SPI supply. This pin should not be connected to any external circuits that may draw more than 20 mA. Voltage of this pin will vary in module light sleep mode and approach zero in module deep sleep mode.

WiFi

Standard: 802.11b/g/n (2.4 GHz ONLY) – 1T1R

Table 2: WiFi Specifications

Parameter	Description	Min	Typ	Max	Unit
General					
Freq. (EU)	Operating frequency (EU)	2.402		2.482	GHz
Ch. (EU)	Channel (EU)	1		13	
Freq. (US)	Operating frequency (US)	2.402		2.472	GHz
Ch. (US)	Channel (US)	1		11	
Power max. (EU/US)	Maximum power (EU/US)			20	dBm
Tx					
Tx Power @B – 1 Mbps	Tx power at B mode with data rate 1 Mbps		18	20	dBm
EVM (Peak) @B – 1 Mbps	EVM (Peak) at B mode with data rate 1 Mbps			8	%
Freq. Err. @B – 1 Mbps	Frequency error at B mode with data rate 1 Mbps	–40	0	40	kHz
Tx Power @G – 54 Mbps	Tx power at G mode with data rate 54 Mbps		16	20	dBm
EVM (RMS) @G – 54 Mbps	EVM (RMS) at G mode with data rate 54 Mbps			–25	dB
Freq. Err. @G – 54 Mbps	Frequency error at G mode with data rate 54 Mbps	–40	0	40	kHz
Tx Power @N20 – MCS7	Tx power at N mode with data rate MCS7 and 20 MHz bandwidth		15	20	dBm
EVM (RMS) @N20 – MCS7	EVM rms at N mode with data rate MCS7 and 20 MHz bandwidth			–27	dB
Freq. Err. @N20 – MCS7	Frequency error at N mode with data rate MCS7 and 20 MHz bandwidth	–40	0	40	kHz
Rx					
Rx Sens. @B – 1 Mbps	Rx sensitivity at B mode with data rate 1 Mbps		–92.0	–82.0	dBm
Rx Sens. @G – 54 Mbps	Rx sensitivity at G mode with data rate 54 Mbps		–76.5	–66.0	dBm
Rx Sens. @N20 – MCS7	Rx sensitivity at N mode with data rate MCS7 and 20 MHz bandwidth		–71.4	–64.0	dBm

Bluetooth

Standard: BLE 5.0 – 1T1R

Table 3: Bluetooth Specifications

Parameter	Description	Min	Typ	Max	Unit
General					
Freq.	Operating frequency	2.4000		2.4835	GHz
Ch.	Channel	0		39	
Power max.	Maximum power			20	dBm
Tx					
Tx Power @Ch.37 – 1 Mbps	Tx power at channel 37 (freq. = 2402 MHz) with data rate 1 Mbps		17	20	dBm
Freq. Err. @Ch.37 – 1 Mbps	Frequency error at channel 37 (freq. = 2402 MHz) with data rate 1 Mbps	−50	0	50	kHz
Tx Power @Ch.38 – 1 Mbps	Tx power at channel 38 (freq. = 2426 MHz) with data rate 1 Mbps		17	20	dBm
Freq. Err. @Ch.38 – 1 Mbps	Frequency error at channel 38 (freq. = 2426 MHz) with data rate 1 Mbps	−50	0	50	kHz
Tx Power @Ch.39 – 1 Mbps	Tx power at channel 39 (freq. = 2480 MHz) with data rate 1 Mbps		17	20	dBm
Freq. Err. @Ch.39 – 1 Mbps	Frequency error at channel 39 (freq. = 2480 MHz) with data rate 1 Mbps	−50	0	50	kHz
Rx					
Rx Sens. @Ch.38 – 2 Mbps	Rx sensitivity at channel 38 (freq. = 2426 MHz) with data rate 2 Mbps		−93.5		dBm
Rx Sens. @Ch.38 – 1 Mbps	Rx sensitivity at channel 38 (freq. = 2426 MHz) with data rate 1 Mbps		−97.5	−70.0	dBm
Rx Sens. @Ch.38 – 500 kbps	Rx sensitivity at channel 38 (freq. = 2426 MHz) with data rate 500 kbps		−100.0		dBm

LTE

Standard: CAT-M1, CAT-NB1, CAT-NB2

Table 4: LTE Frequency Bands (in MHz)

Band	Duplex	Uplink Freq. (MHz)	UL BW (MHz)	Downlink Freq. (MHz)	DL BW (MHz)	LTE-M	NB-IoT
1	FDD	1920 – 1980	60	2110 – 2170	60	Yes	Yes
2	FDD	1850 – 1910	60	1930 – 1990	60	Yes	Yes
3	FDD	1710 – 1785	75	1805 – 1880	75	Yes	Yes
4	FDD	1710 – 1755	45	2110 – 2155	45	Yes	Yes
5	FDD	824 – 849	25	869 – 894	25	Yes	Yes
8	FDD	880 – 915	35	925 – 960	35	Yes	Yes
12	FDD	699 – 716	17	729 – 746	17	Yes	Yes
13	FDD	777 – 787	10	746 – 756	10	Yes	Yes
14	FDD	788 – 798	10	758 – 768	10	Yes	Yes
17	FDD	704 – 716	12	734 – 746	12	No	Yes
18	FDD	815 – 830	15	860 – 875	15	Yes	Yes
19	FDD	830 – 845	15	875 – 890	15	Yes	Yes
20	FDD	832 – 862	30	791 – 821	30	Yes	Yes
25	FDD	1850 – 1915	65	1930 – 1995	65	Yes	Yes
26	FDD	814 – 849	35	859 – 894	35	Yes	Yes
28	FDD	703 – 748	45	758 – 803	45	Yes	Yes
66	FDD	1710 – 1780	70	2110 – 2200	90	Yes	Yes
85	FDD	698 – 716	18	728 – 746	18	Yes	Yes

Table 5: LTE Specifications

Parameter	Description	Min	Typ	Max	Unit
General					
Power max.	Maximum power			23	dBm
Tx					
Tx Power @Band 8 (900 MHz GSM)	Tx power at Band 8 (900 MHz GSM)		22	23	dBm
Tx Power @Band 2 (1900 MHz PCS)	Tx power at Band 2 (1900 MHz PCS)		22	23	dBm
Rx					
Rx sens. @Band 8 (900 MHz GSM)	Rx sensitivity at Band 8 (900 MHz GSM)		−103	−100	dBm
Rx sens. @Band 2 (1900 MHz PCS)	Rx sensitivity at Band 2 (1900 MHz PCS)		−103	−100	dBm

LoRa(WAN)

Mode: LoRa RAW mode and LoRa WAN mode
LoRaWAN Node Type: Class Type A, Class Type C
Frequency Band: EU868, US915

Table 6: LoRa(WAN) Specifications

Parameter	Description	Min	Typ	Max	Unit
General					
Freq. (EU)	Frequency band (EU)	863		870	MHz
Freq. (US)	Frequency band (US)	902		928	MHz
Power max. (EU)	Maximum power (EU)			15	dBm
Power max. (US)	Maximum power (US)			22	dBm
Tx					
Tx power @866.4 MHz [EU868]	Tx power (Tx tone) at 866.4 MHz		14	15	dBm
Tx power @918.2 MHz [US915]	Tx power (Tx tone) at 918.2 MHz		21	22	dBm
Rx					
Rx Sens. @866.4 MHz, BW=500 kHz, SF=12	Rx sensitivity at 866.4 MHz, 500 kHz bandwidth and SF=12		−127		dBm

7.1.5 Module Interface

Power Management

Table 7: Power Consumption by Mode of Operation

Mode of Operation	Min	Typ	Max	Unit
Idle (no radio but MicroPython is running)		30		mA
Light sleep (wake up or restart is required for MicroPython to run)		800		µA
Deep sleep (wake up or restart is required for MicroPython to run)		10		µA

Memory Allocation

Module OS firmware, OTA and user space sizes:

- Module OS firmware: 2,560 KB
- OTA1 space: 2,560 KB
- OTA2 space: 2,560 KB
- User space: 8 MB

7.1.6 Mechanical Specifications

Module Dimensions

All pins have a pin width of 0.7 mm with the exception of pin VBATT (pin #A4) with a 1.0 mm width.

Recommended PCB Landing Pattern

Recommended Soldering Profile

7.1.7 MicroPython

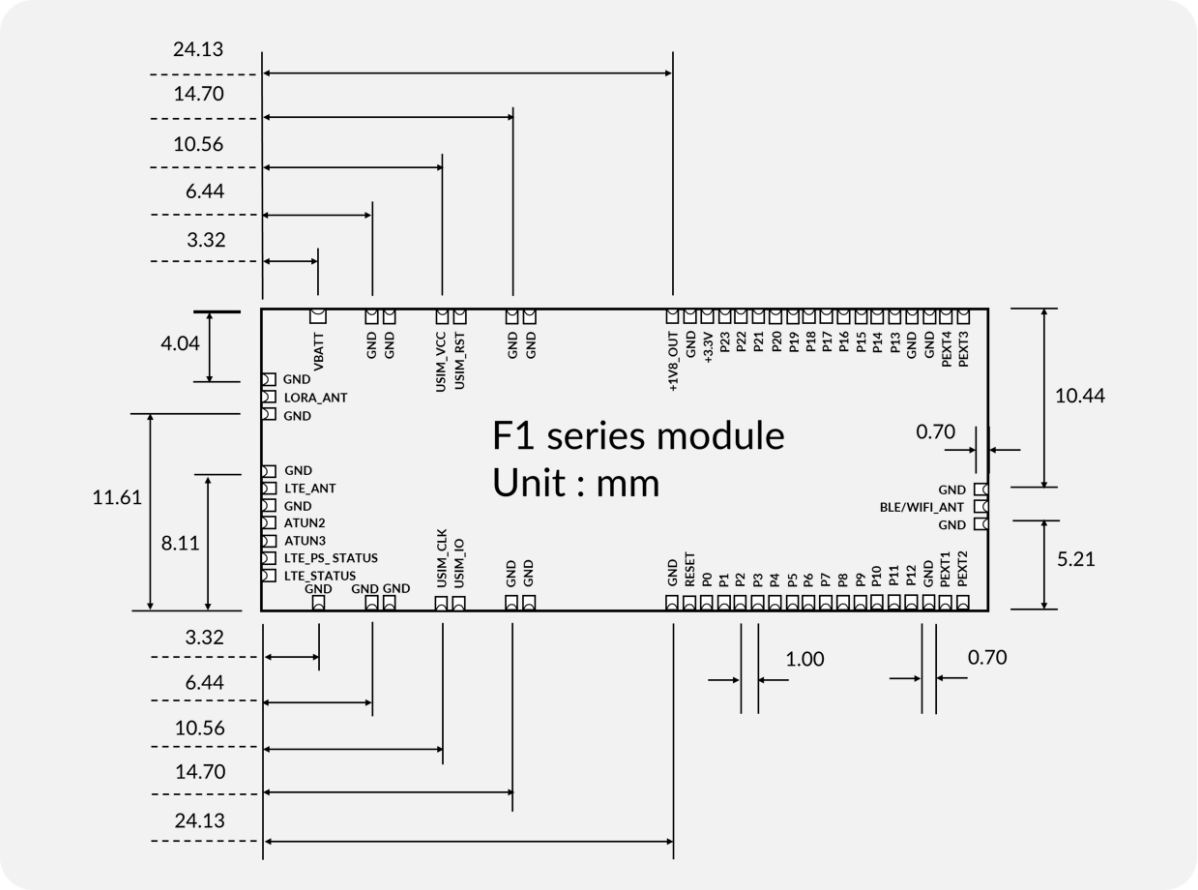


Fig. 4: *Figure 4: F1 Smart Module Dimensions*

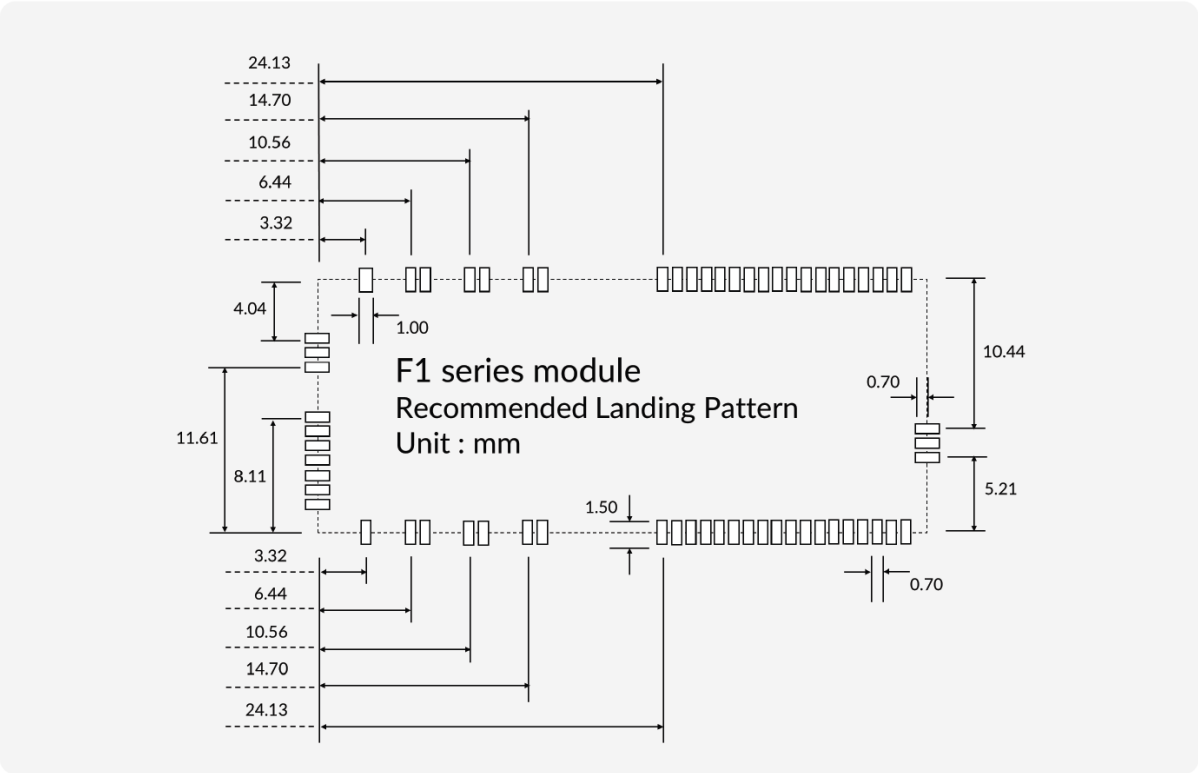


Fig. 5: *Figure 5: Recommended PCB Landing Pattern (Top View)*

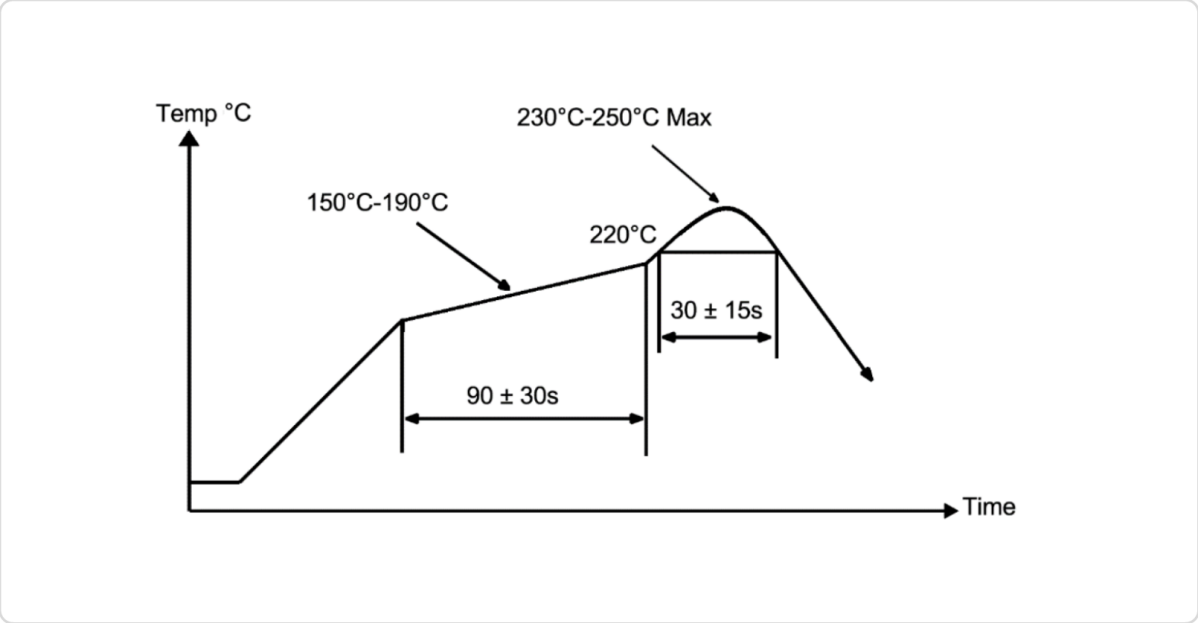


Fig. 6: *Figure 6: Recommended Soldering Profile*

Device Programming via UART

- By default, the F1 Smart Module runs an interactive Python REPL (Read-Eval-Print Loop) on UART0 which is connected to P0 (RX) and P1 (TX) running at 115200 baud.
- The module can be connected via a development board or any USB UART adapter. Code can be run via the REPL and the SG Wireless Ctrl. Visual Studio Code plug-in can also be used to upload code to the board.

Module-supported Libraries

Table 9: F1 Smart Module Supported Libraries

Library	Description
Python Standard Libraries	array, asyncio, binascii, builtins, cmath, collections, errno, gc, gzip, hashlib, heapq, io, json, math, os, platform, random, re, select, socket, ssl, struct, sys, time, zlib, _thread
MicroPython-specific Libraries	Bluetooth, btree, cryptolib, deflate, framebuf, machine, micropython, neopixel, network, uctypes, esp, esp32
SGW3501 Module-specific Libraries	**lte**: Ready-to-use LTE CAT-M1/NB1/NB2 library **lora**: Ready-to-use LoRa RAW and full stack LoRa WAN device Class A, Class C library **ctrl**: Ready-to-use Ctrl Cloud Platform client library

Note

MicroPython documentation library with API function calls: https://docs.micropython.org/. Libraries may vary by version – please check the latest F1 Module firmware release note for the required library

(pre-loaded onto Module).

7.1.8 Product Packaging

Modules are packed in tape-and-reel packaging and shipped out in carton boxes.

- **Tape –** MSL (Moisture Sensitivity Level): 3
- **Reel –** 250 pcs per reel

Note

All units are in millimetres.

7.1.9 Revision History

Version	Released Date	Description
1.0	Feb 7, 2024	Initial document release
2.0	Aug 8, 2024	Add Certification information

7.1.10 Certification

Model	Certification Type	Description
F1	CE Certificate	CE certificate of F1 (Type designation: SGW3501)
F1/C	CE Certificate	CE certificate of F1/C (Type designation: SGW3401)
F1/L	CE Certificate	CE certificate of F1/L (Type designation: SGW3201)
F1	FCC Certificates	FCC certificates of F1 (Type designation: SGW3501)

7.2 F1 Starter Kit

7.2.1 Introduction

The F1 Starter Kit is the evaluation board for the F1 Smart Module.

With Wi-Fi, BLE, LTE, and LoRa in one Arduino-compatible package, this development platform enables rapid prototyping and deployment of IoT applications requiring connectivity flexibility.

Varieties

Based on the cellular technologies and region, there are 4 varieties of F1 Starter Kit:

- **F1 Starter Kit NA, CAT-M1** (SGW3501-EVK-NA-CATM1-8130D)
- **F1 Starter Kit EU, CAT-M1** (SGW3501-EVK-NA-CATM1-8130D)
- **F1 Starter Kit Global, CAT-M1** (SGW3501-EVK-NA-CATM1-8130D)
- **F1 Starter Kit Global, NB-IoT** (SGW3501-EVK-NA-CATM1-8130D)

Each Starter Kit comes with the following:

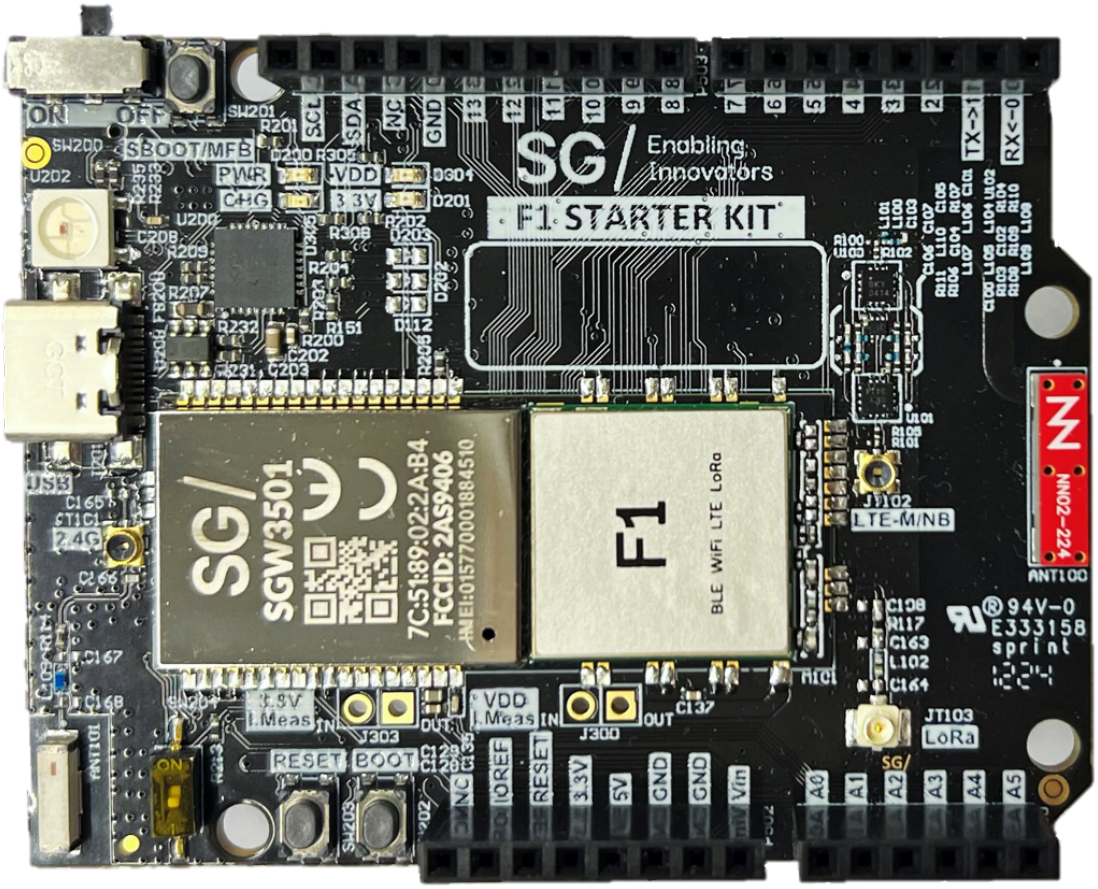


Fig. 7: *F1 Starter Kit*

- F1 EVK
- CAP/T sensor
- 1-hour consultation

What Makes the F1 Starter Kit Unique

The F1 Starter Kit has four wireless protocols on a single board:

- **Wi-Fi** for high-bandwidth local connectivity
- **BLE** for low-power device interactions
- **LTE** for reliable cellular communication
- **LoRa** for long-range, low-power wide-area networking

Many connectivity decisions get made too early in development, locking you into assumptions without real-world data to back them up.

The multi-protocol approach with the F1 Starter Kit gives you the flexibility to choose the optimal connectivity solution for each specific use case, and test different protocols as your projects evolve. Need Wi-Fi for configuration but cellular for deployment? Both are right here. Want to add LoRa for remote sensors later? Already built in.

What to Expect in Development

Built with developer productivity in mind, look out for:

- **Arduino compatibility** means your existing shields work without modifications. Same form factor, same pinout, same ecosystem you already know.
- **MicroPython support** lets you prototype quickly without diving into Embedded C. Write readable code, iterate fast, and optimize later.
- **SG Ctrl Cloud Platform** integration handles the device management, over-the-air updates, and data collection — everything to propel your prototype to production.

7.2.2 Features

Connectivity

- LTE Cat-M1/NB
- LoRa(WAN)
- Wi-Fi 4 802.11 b/g/n standard at 2.4 GHz
- Bluetooth LE

Microprocessor

- Xtensa® Dual-core 32-bit LX7 Microprocessor
- Up to 240 MHz
- 384 kB ROM
- 512 kB SRAM
- 16 kB RTC SRAM

Memory

- Additional 8 MB PSRAM
- Additional 16 MB NOR Flash

Peripherals

- UART ×2 (one is used as USB-UART debug on Starter Kit)
- SPI ×2
- I2C ×2

On-board Peripherals

- USB-C Device connector
- USB-C OTG connector
- Nano-SIM holder
- Micro-SD card holder
- RGB LED

Arduino UNO Shield Compatible Expansion

- Up to 6 Analog Input
- Up to 14 Digital IO (all configurable to PWM)
- Qwiic connector

Power

- Power by USB-C at 5 V
- Power by VIN at 5 V
- Power by Li-Ion / Li-Polymer battery at 3.7 V to 4.2 V (battery not included)
- Li-Polymer Battery Connector
- Battery charging and fuel gauge for accurate battery capacity measurement

7.2.3 Starter Kit Overview

Block Diagram

Starter Kit Topology — Top View

Major Component List (Top)

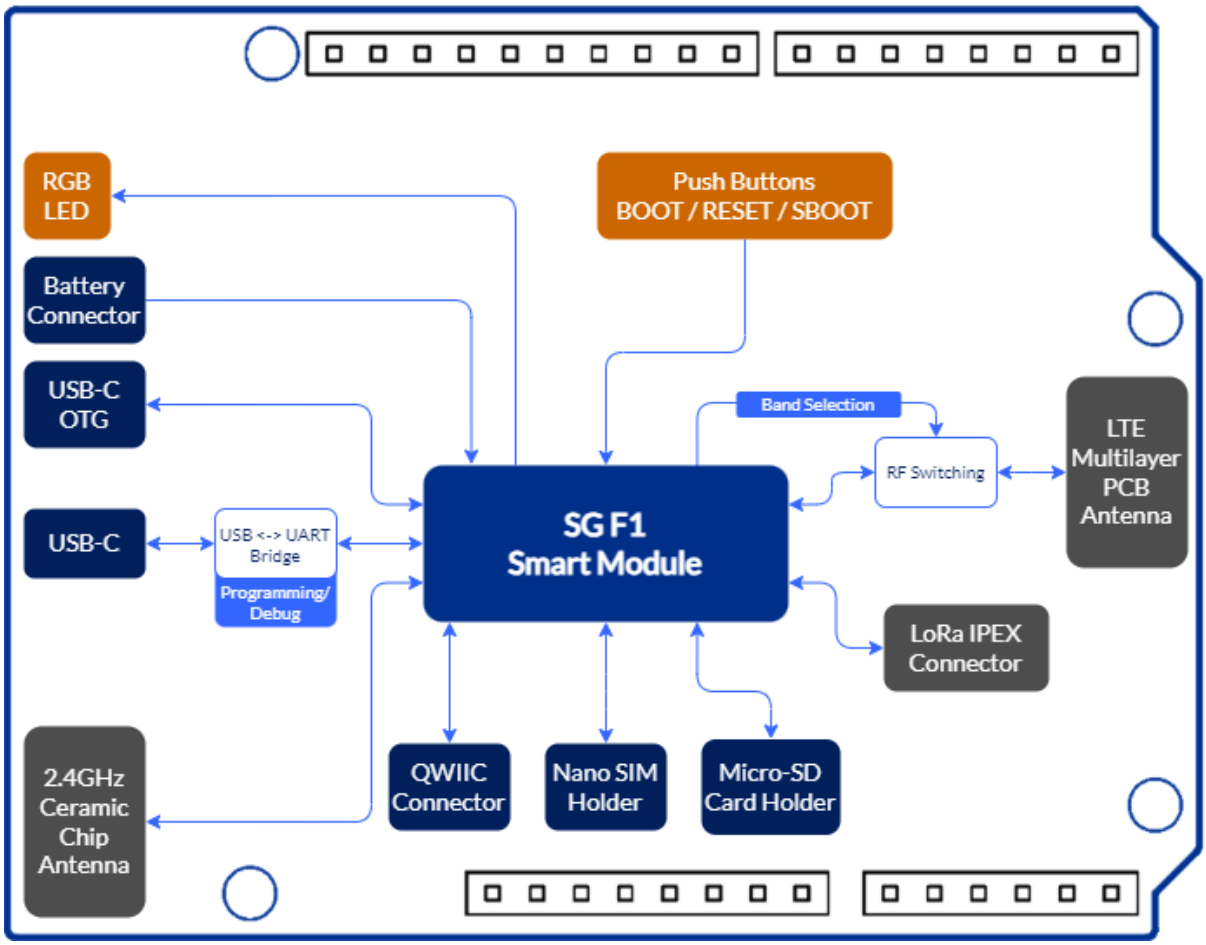


Fig. 8: *F1 Starter Kit Block Diagram*

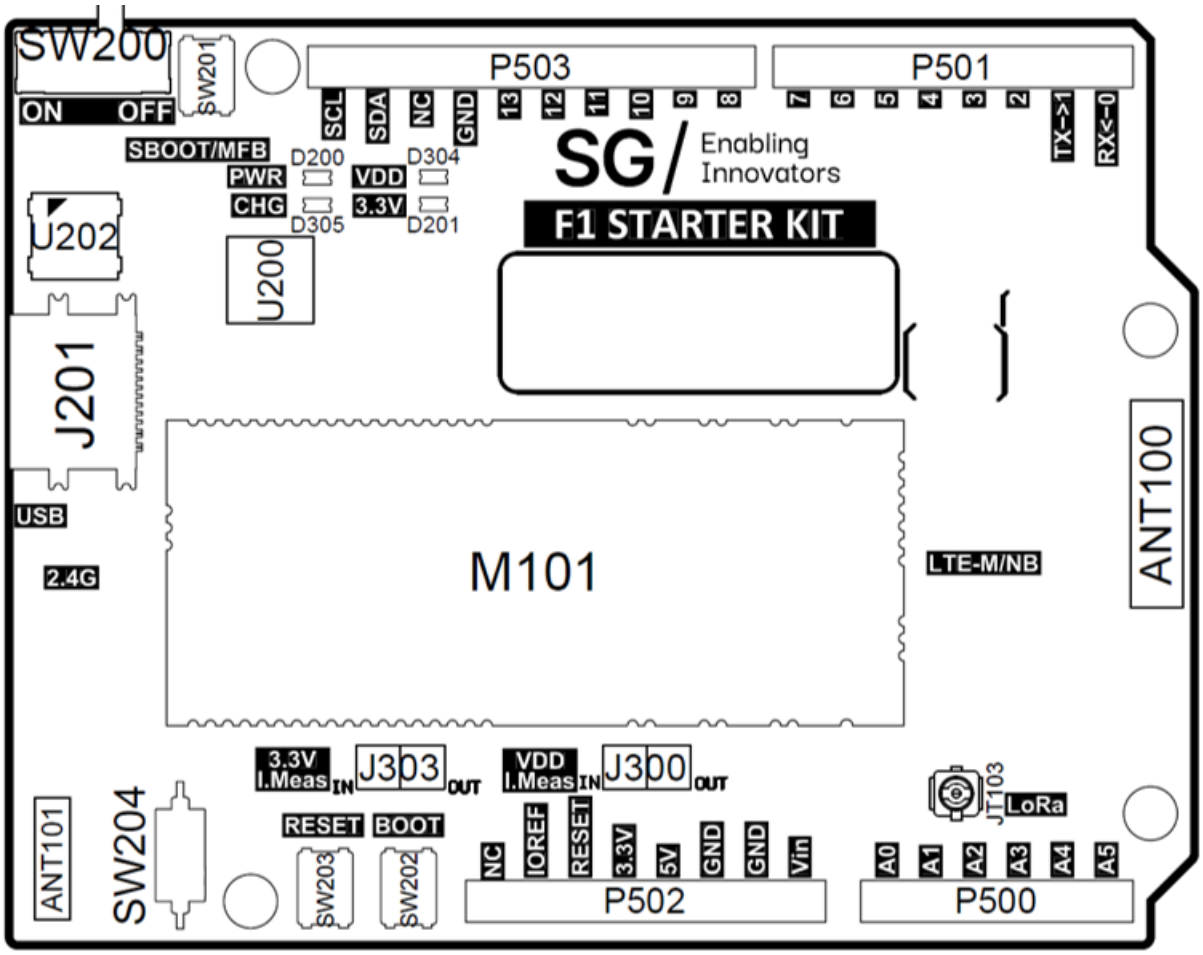


Fig. 9: *F1 Starter Kit — Top View*

Ref.	Description	Ref.	Description
M101	SG F1 Smart Module	D200	Power LED
J201	16P USB-C Connector	D305	Battery Charging LED
SW200	Main Power Switch	D304	VDD ON LED
ANT100	LTE Multilayer PCB Antenna	D201	3.3V ON LED
ANT101	2.4 GHz Ceramic Chip Antenna	J303	3.3V Current Measurement Jumper
JT103	IPEX Connector (LoRa Antenna)	J300	VDD Current Measurement Jumper
SW201	Secure Boot Button / Multi-Function Button	U202	RGB LED
SW202	Boot Button		
SW203	Reset Button		
SW204	Digital IO 12 Switch		

Starter Kit Topology — Bottom View

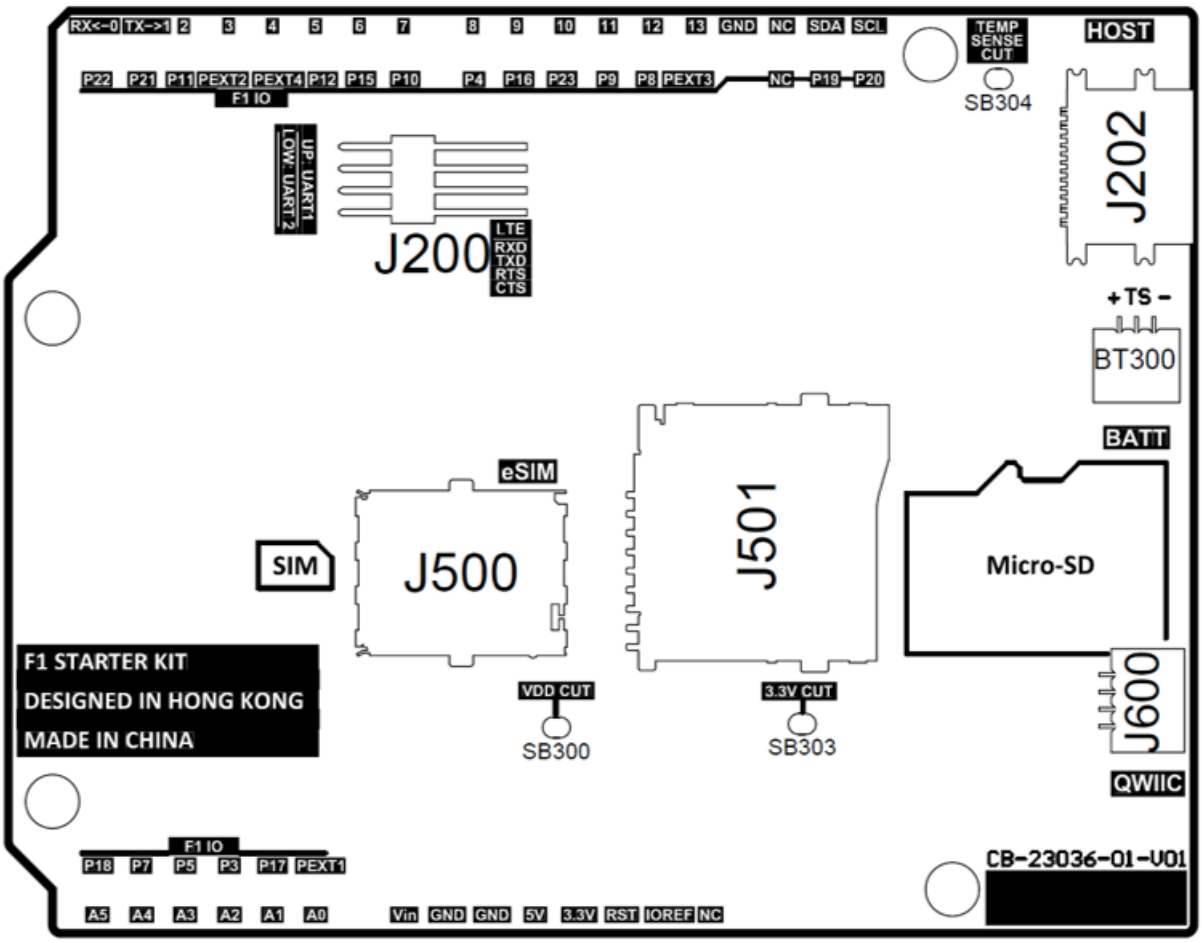


Fig. 10: *F1 Starter Kit — Bottom View*

Major Component List (Bottom)

Ref.	Description
J202	16-P Host USB-C Connector
BT300	1 mm Pitch Battery Connector
J500	Nano SIM Card Slot
J501	Micro-SD Card Slot
SB300	VDD Shorted Jumper
SB303	3.3V Shorted Jumper
SB304	Battery Temperature Sense Shorted Jumper
J600	Qwiic Connector

7.2.4 Pinout & Pin Definitions

All IO pins on the F1 Starter Kit can be MUXed to any digital function, while P19 and P20 are connected to the I2C lane of the on-board fuel gauge.

Analog and touch inputs should only be used on analog pins.

The mechanical arrangement of the pinout is compatible with standard Arduino UNO shields.

`Starter Kit Pinout Diagram - PDF`

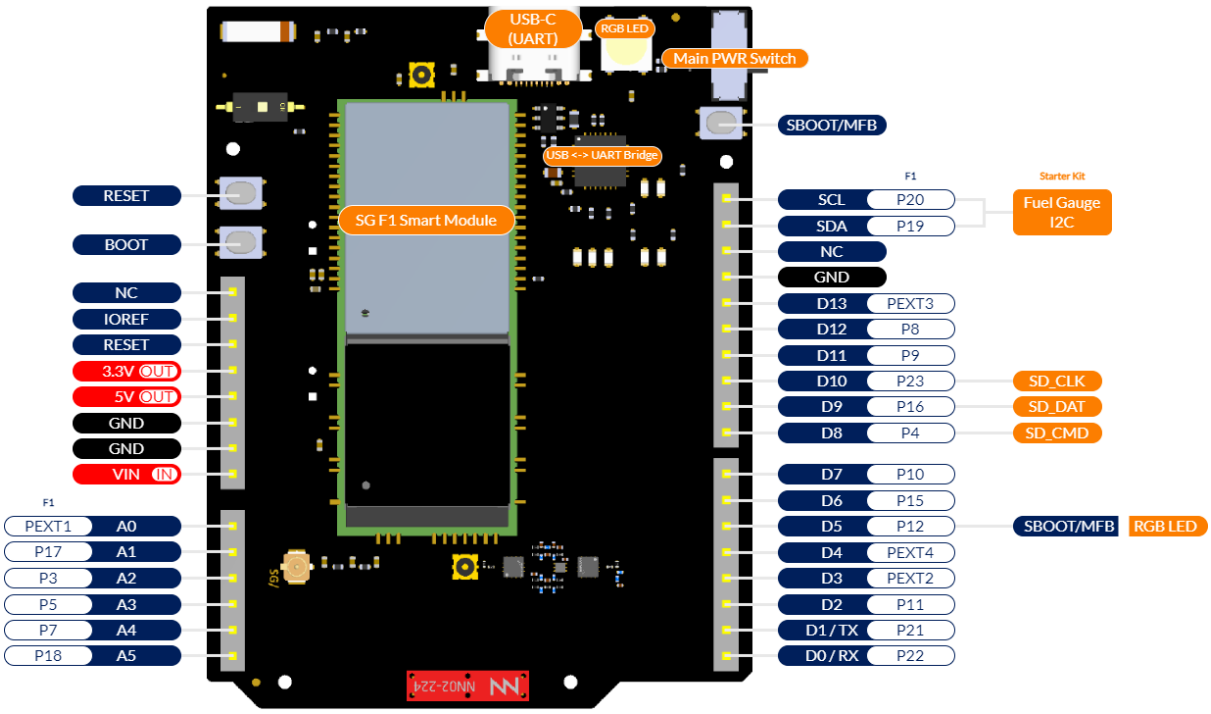


Fig. 11: *Figure 1: F1 Starter Kit Pinout Diagram*

Pin Definitions

Table 1: Analog / Power Pin Table

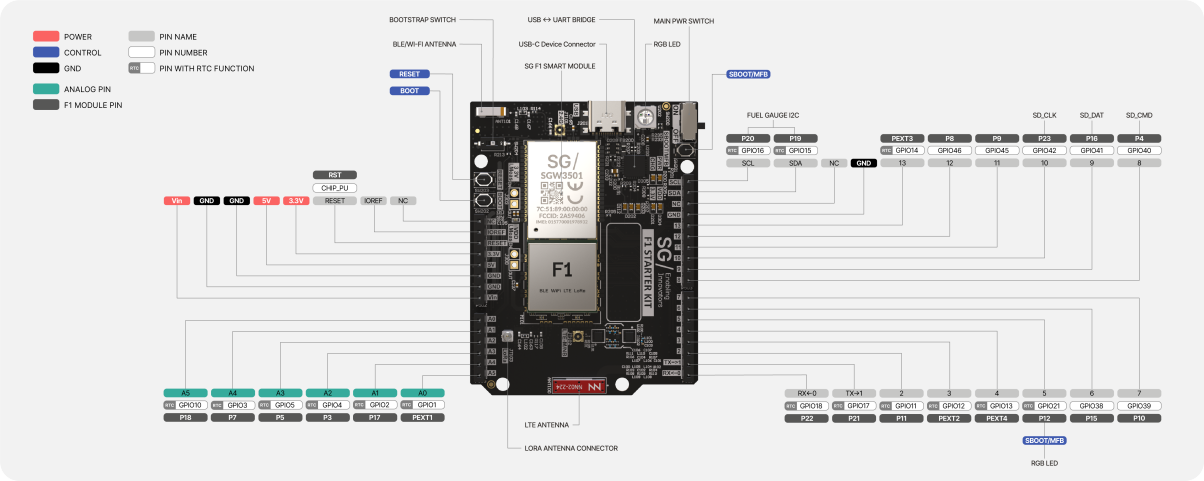


Fig. 12: *Figure 2: Top View of F1 Starter Kit*

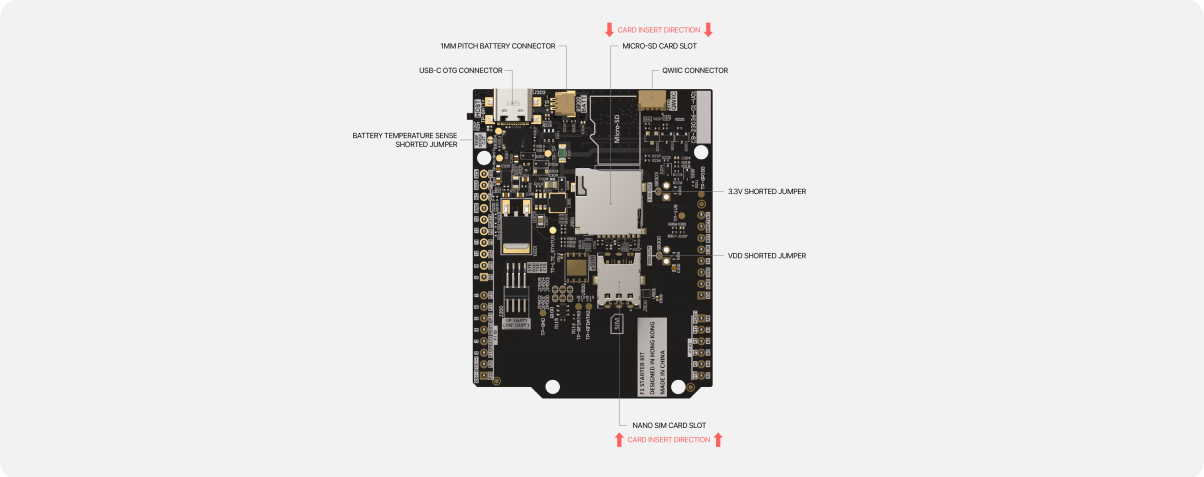


Fig. 13: *Figure 3: Bottom View of F1 Starter Kit*

Pin	Function	Type	Description	F1 Pin
1	NC	NC	Not Connected	/
2	IOREF	IOREF	Reference for digital logic — connected to 3.3 V	/
3	Reset	Reset	Reset	RESET
4	3V3	Power	3.3 V Power Rail	/
5	5V	Power	5 V Output Power Rail	/
6	GND	Power	Ground	/
7	GND	Power	Ground	/
8	VIN	Power	5 V Voltage Input	/
9	A0	Analog	Analog input 0	PEXT1
10	A1	Analog	Analog input 1	P17
11	A2	Analog	Analog input 2	P3
12	A3	Analog	Analog input 3	P5
13	A4	Analog	Analog input 4	P7
14	A5	Analog	Analog input 5	P18

Table 2: Digital Pin Table

Pin	Function	Type	Description	F1 Pin
1	SCL	Digital	I2C Serial Clock (SCL)	P20
2	SDA	Digital	I2C Serial Data (SDA)	P19
3	NC	NC	Not Connected	/
4	GND	Power	Ground	/
5	D13	Digital	Digital IO 13	PEXT3
6	D12	Digital	Digital IO 12*	P8
7	D11	Digital	Digital IO 11	P9
8	D10	Digital	Digital IO 10	P23
9	D9	Digital	Digital IO 9	P16
10	D8	Digital	Digital IO 8	P4
11	D7	Digital	Digital IO 7	P10
12	D6	Digital	Digital IO 6	P15
13	D5	Digital	Digital IO 5	P12
14	D4	Digital	Digital IO 4	PEXT4
15	D3	Digital	Digital IO 3	PEXT2
16	D2	Digital	Digital IO 2	P11
17	D1 / TX	Digital	Digital IO 1 / UART TX	P21
18	D0 / RX	Digital	Digital IO 0 / UART RX	P22

* You may need to switch OFF SW204 when using Digital IO 12.

7.2.5 Board Operation

- *Getting Started* — F1 platform setup guide
- *Programming References* — MicroPython references for F1 platform
- Additional online resources at https://www.sgwireless.com

See *Getting Started* for setup instructions.

CHAPTER

EIGHT

SOFTWARE

8.1 Ctrl Web Platform

The all-in-one Cloud Platform that lets you configure, deploy and manage your devices to give you full Ctrl over your IoT networks.

- Visualizes your sensor data.
- Checks the status of your entire deployment.
- Distributes firmware updates on a scalable approach.

In a nutshell, Ctrl is an environment designed to optimize your IoT experience when using F1 smart modules.

8.1.1 Projects

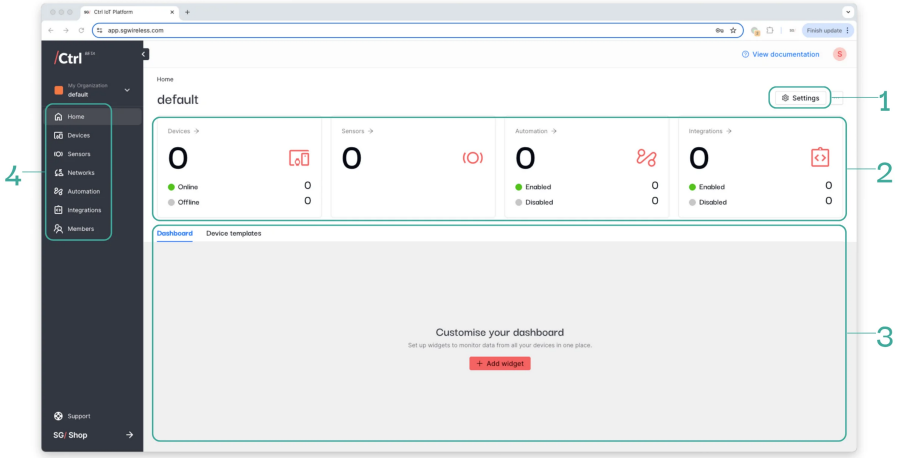
Your initial project is automatically generated when you create your account. You can establish multiple projects to organize different groups of IoT assets based on your requirements. Each project stores its IoT assets along with relevant information, such as templates, devices, sensors, network profiles, automation rules, integrations, and member management.

- *Project components*
- *Add project*
- *Switch project*
- *Edit project*
- *Delete project*

Project Components

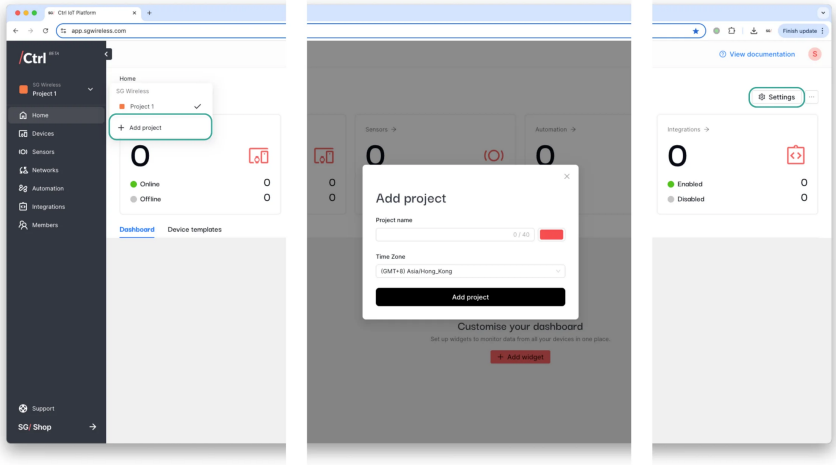
Your first project "default" is automatically created when you sign up for a Ctrl account. This is the project home page. Below are the components of a project:

- **Settings:** Name your project, pick a colour, and change the time zone if needed.
- **Set-up:** Overview of your project set-up, from connected assets to configured workflows.
- **Project Dashboard:** Create a centralized view of devices under the same project.
- **Device Templates:** Manage device templates to scale up your deployment and maintenance.
- **Menu Bar:** Quick links to configure each Ctrl building block.



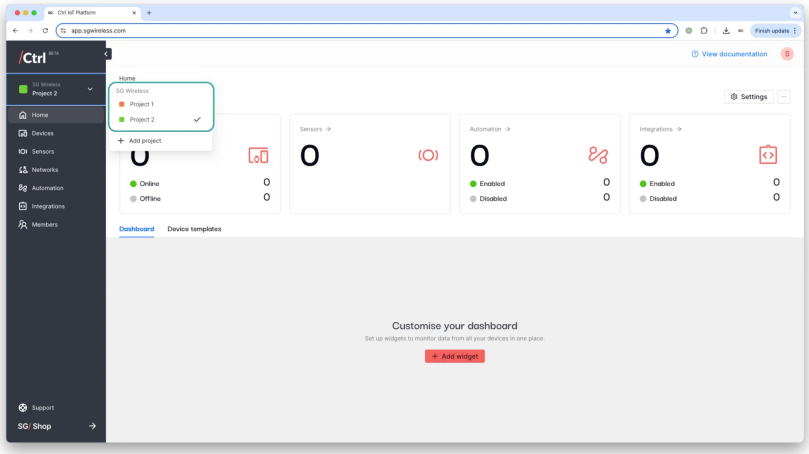
Add Project

To set up new projects, click on “Add project” and follow the prompts. These can always be changed later in “Settings”.



Switch Project

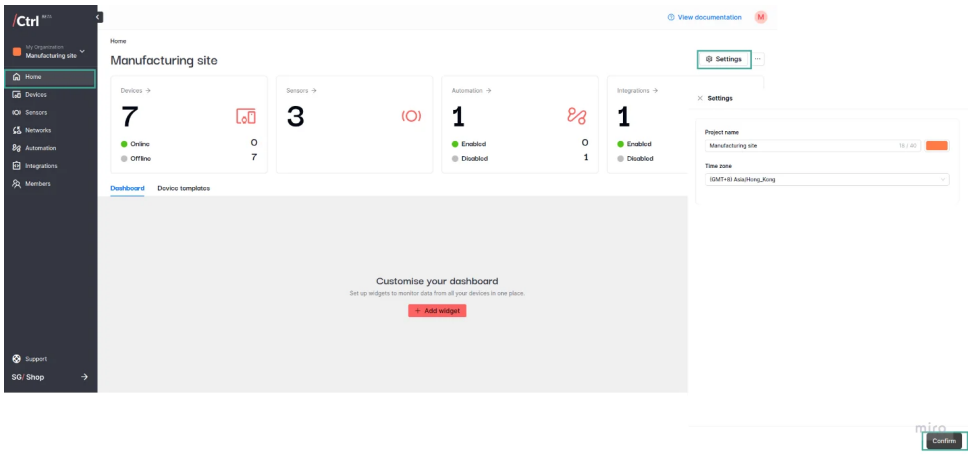
Switch between different projects simply by selecting it. The checkmark indicates the selected project.



Edit Project

Update your project settings through the following steps:

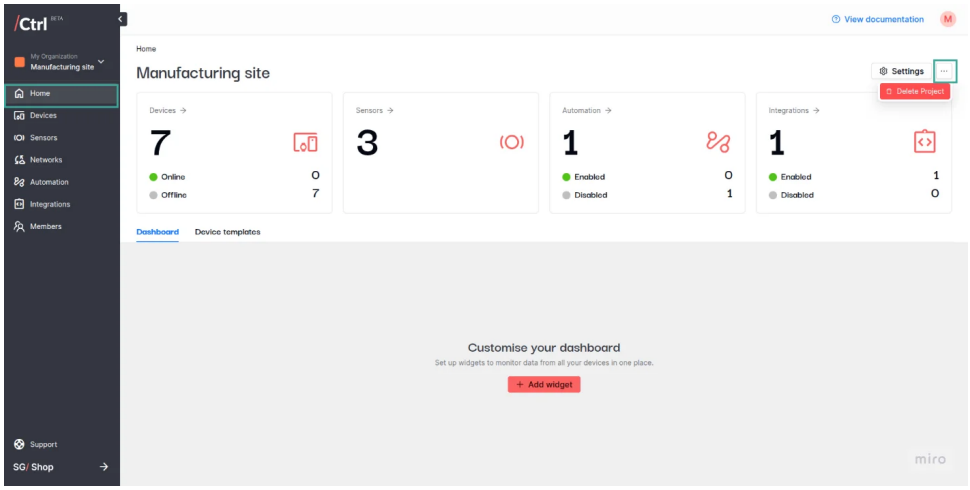
1. On the target project, go to the Home tab.
2. Click the “Settings” button on the top right corner.
3. Make necessary changes.
4. Click “Confirm” to apply the changes.



Delete Project

To delete a project, click on “…”, then “Delete project”. You’ll be asked to confirm this action by inputting your project name.

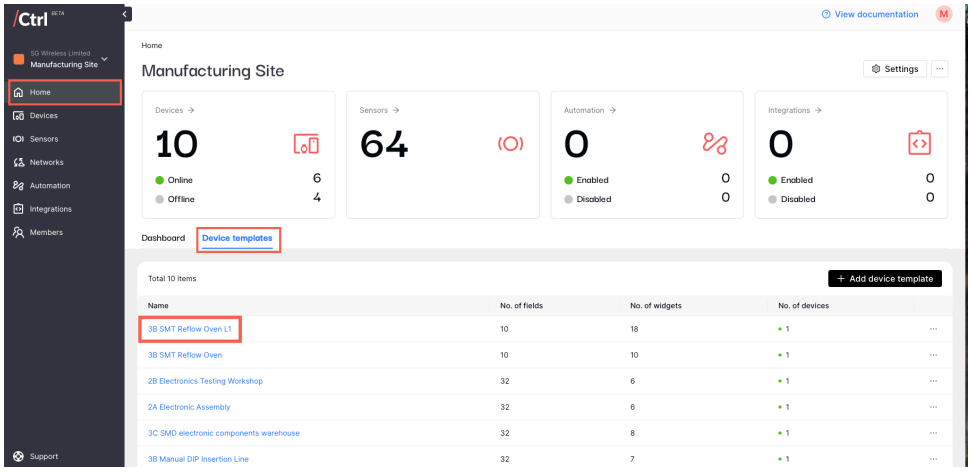
Upon project deletion, all related information will be deleted. **This action cannot be undone.**



8.1.2 Device Templates

Scaling IoT projects is great! Until you realize that it means there'll be hundreds and thousands of devices that you have to configure. By hand.

A Device Template solves this problem: you set up one template that tells a device how to collect and display its data, then apply to all.



You can expect:

- Consistent data handling - All devices using the template store and display data identically
- Unified dashboards - Shared dashboard layouts and widgets
- Bulk management - Update the template to change all associated devices
- Scalability - Essential for managing multiple devices

See how each component comes together to set up your own Device Template:

- *Device Fields*
- *Dashboard Layout*
- *Associated Devices*
- *Software releases*
- *Automation*

Fields

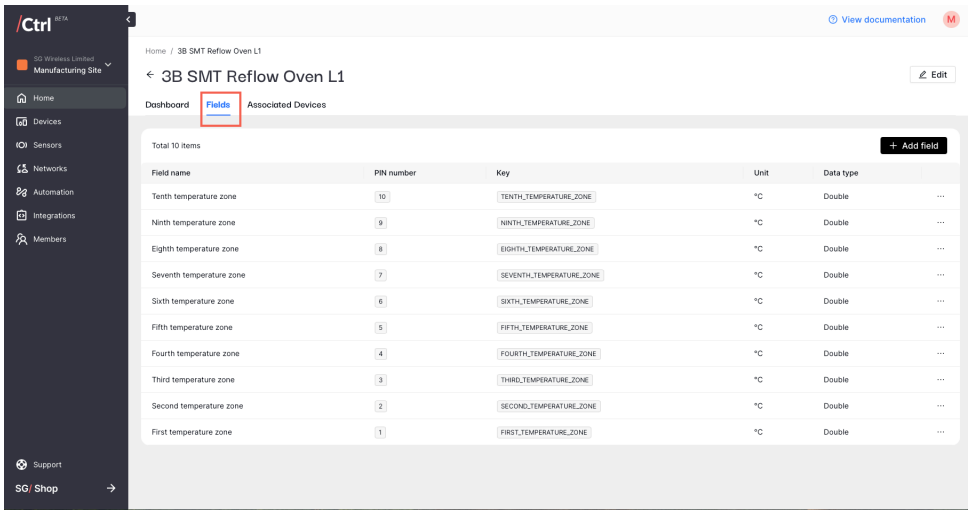
Uplink data received from your device is stored in Ctrl's 'Device Fields' - consider them as individual buckets that store different types of data. For example:

- A TEMPERATURE field stores temperature readings from the temperature sensor connected to your F1 Starter Kit
- A HUMIDITY field stores humidity measurements from the humidity sensor

The device field also stores downlink values sent from Ctrl to your device. For example:

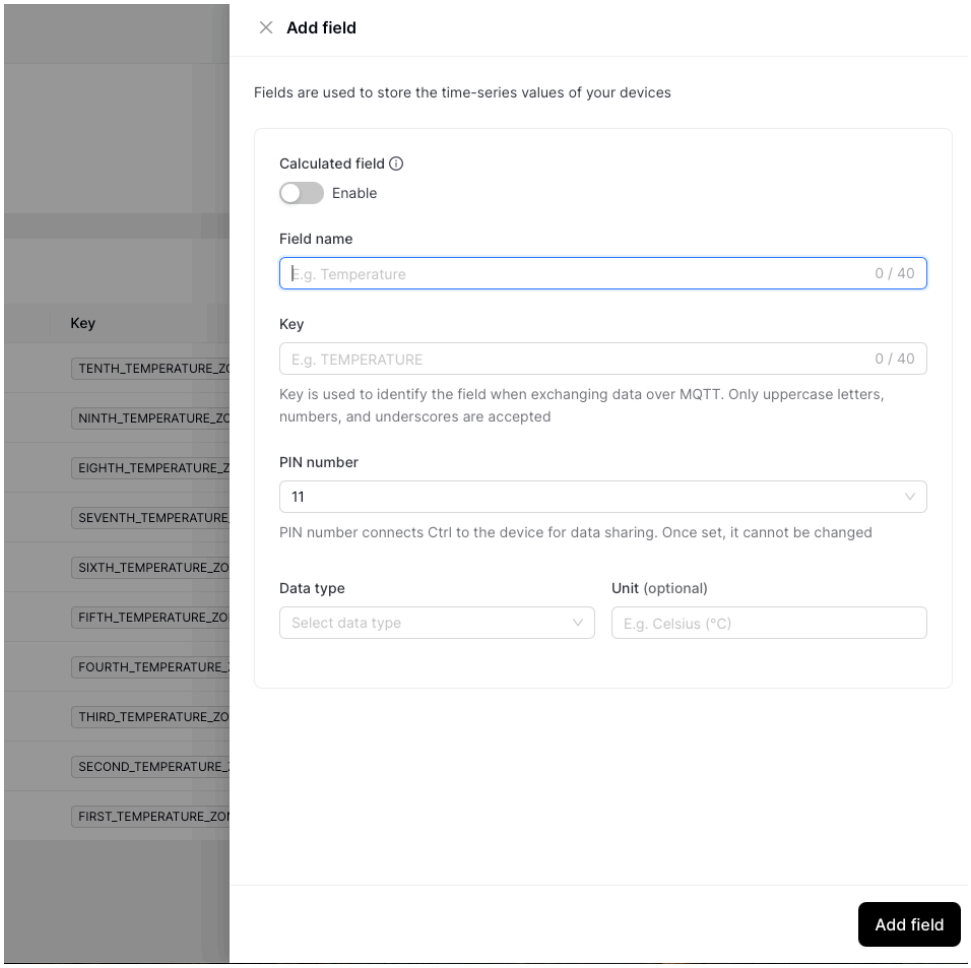
- A SET TEMPERATURE field stores the temperature value adjustment done through your Ctrl dashboard to your device.

Note that each field can be used for both uplink and downlink data.



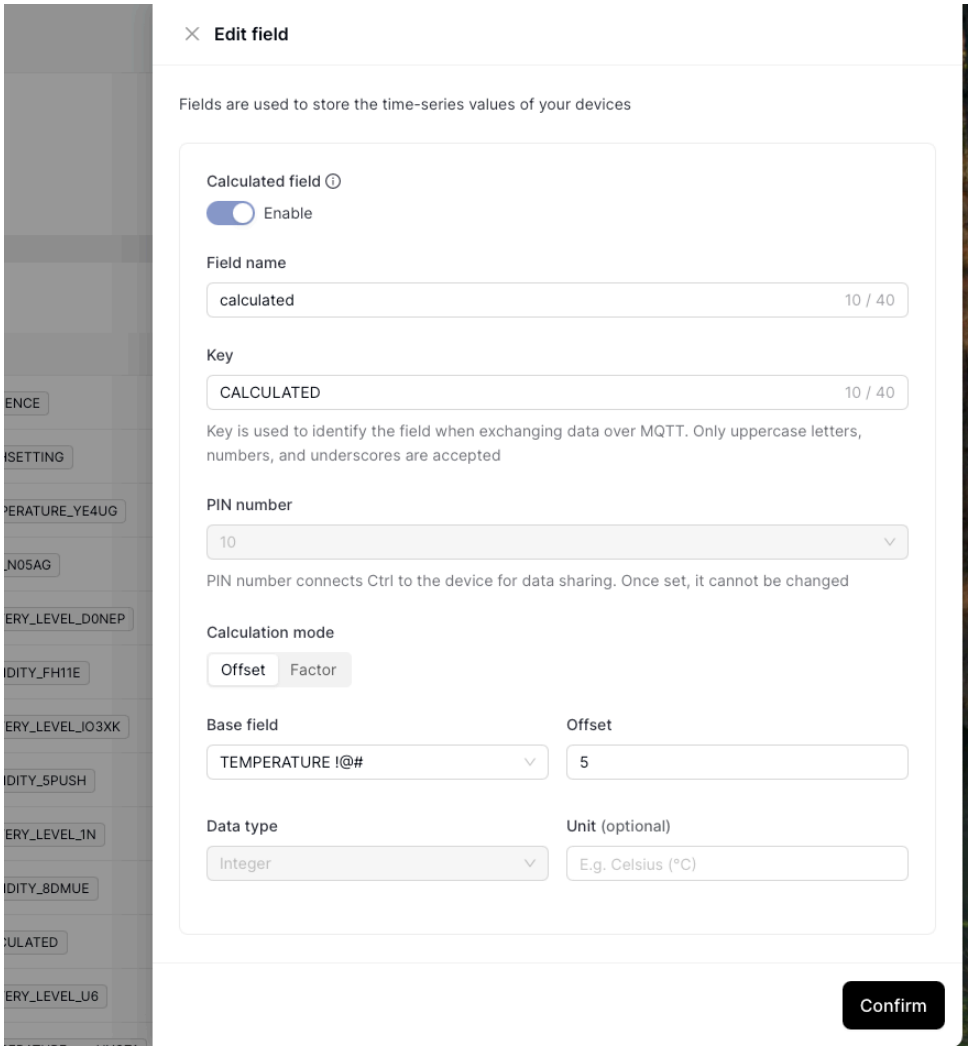
Field Elements

- **Field name:** User-friendly name to identify the fields
- **Key:** Used to identify the field when exchanging data over MQTT. Only uppercase letters, numbers, and underscores are accepted
- **PIN number:** Connects Ctrl to the device for data sharing. It is non-editable.
- **Data type:** Defines the data type of field data. It is non-editable.
- **Unit:** Identifies the quantification unit of the field data.



Field Types

- **Standard Field: Stores direct sensor readings as they come from your device, such as:**
  - Numbers (temperature: 23.5°C)
  - Text (status: “online”)
  - Boolean (door_open: true/false)
  - Location (GPS coordinates: 22.123,144.123)
- **Calculated Field: Can enabled by activating the ‘enable’ toggle during field creation. It stores calculated values of the standard field. Each calculated field can only be associated with one standard field. There are two options of operation available in calculated field:**
  - Offset: Add or subtract a constant. For example: HKT_Time calculated field stores GMT+8 timestamps by adding 8 to all the values of UTC_Time standard field.
  - Factor: Multiply or divide by a constant. For example, depth_feet calculated field stores the depth measurement in feet by dividing all values of depth_meter by 3,28084.

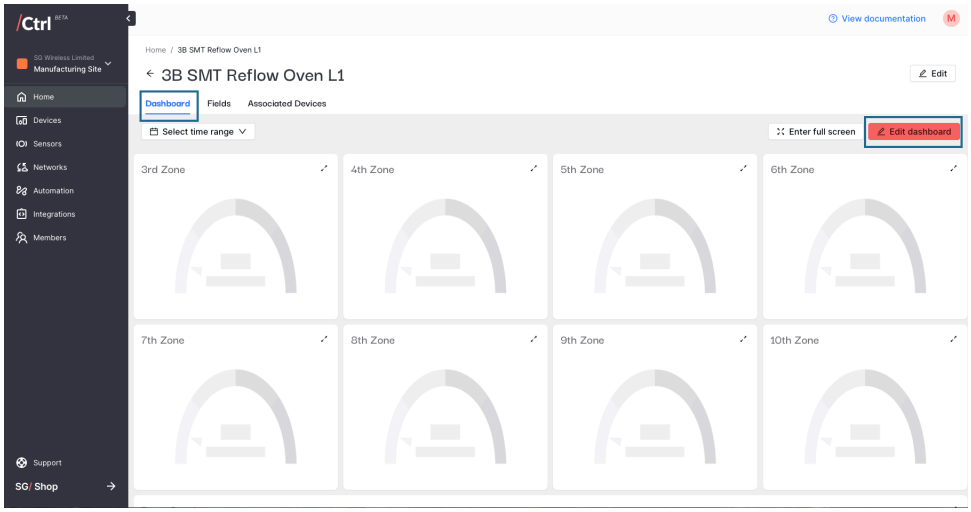


Dasboard Layout

All devices using the same template share identical dashboard configurations. This includes:

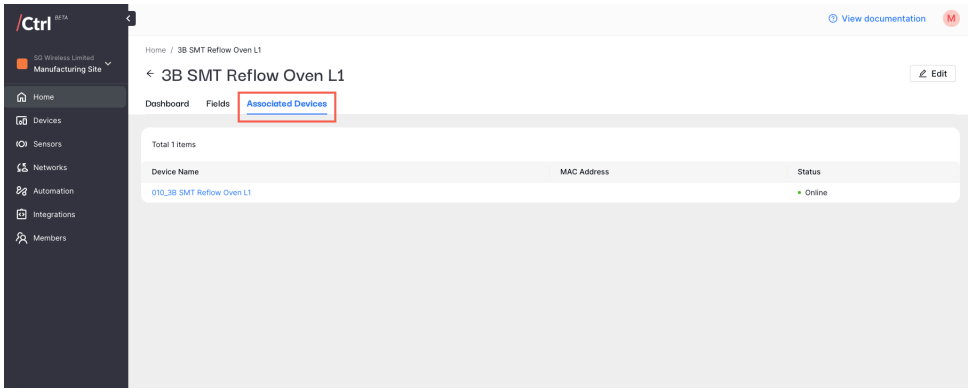
- **Dashboard layout:** Configure types, positions, and sizes of the widgets on each device dashboard. This dashboard is located on each device details.
- **Widget data sources:** Configure data souce fields of each widget.

Details regarding the data widget settings can be found on Dashboard & Widgets section



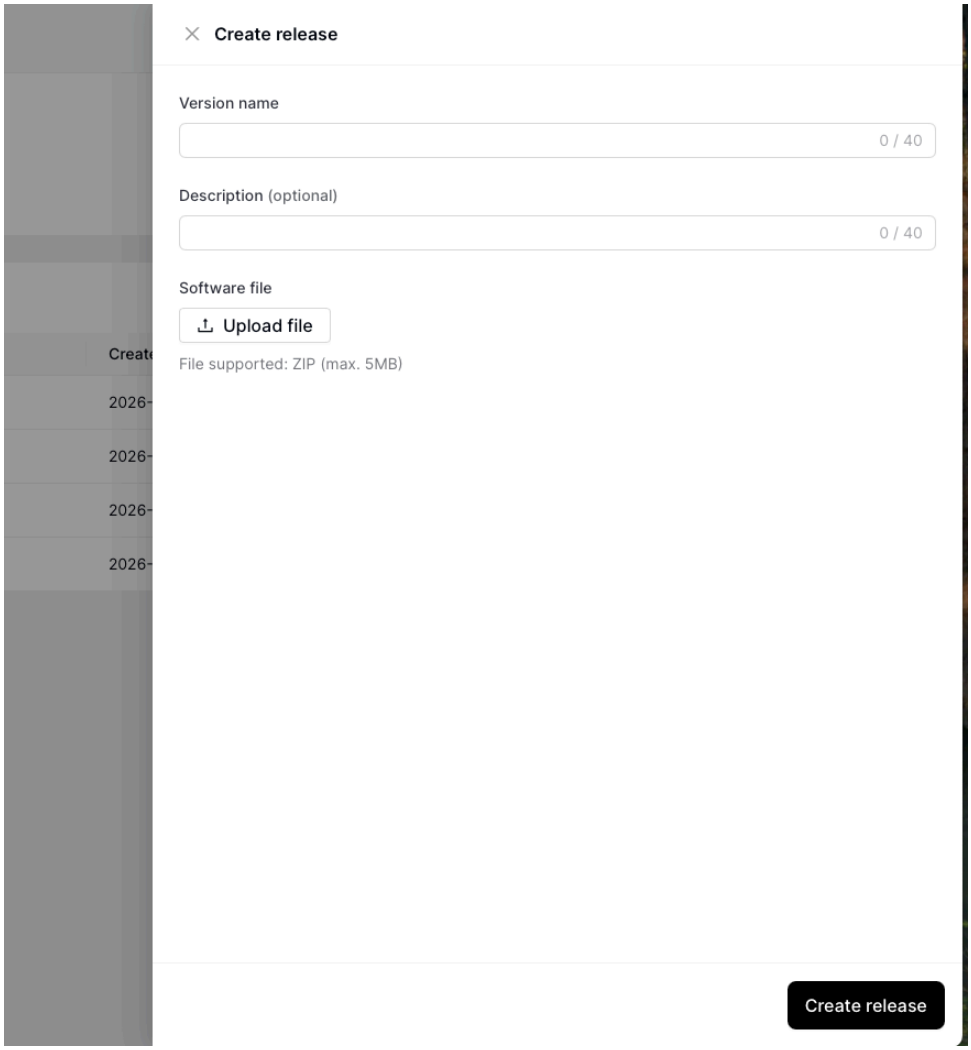
Associated Devices

Associated Devices tab lists all devices under the same template. Any changes made to the template elements, such as dashboard, fields, or software releases, will be reflected on all associated devices.



Software Releases

A repository of file content that can be uploaded over-the-air to the device. User needs to first create a new release and upload the zipped file content on this page before deploying it to the target device(s).



Automation

Automation triggers autonomous actions based on the specified condition. User can apply the automation rule to all devices under the same template or to a specific device.

Trigger

Determine what triggers the condition-checking mechanism:

- Immediately: Checks the condition on every incoming measurement
- Device status: Checks the condition based on the specified device status
- Interval: Checks the condition once every specified interval
- Fixed time: Periodically checks the condition at a specified time of the day. Make sure the time zone has been set correctly.

On top of the nature of triggers, user also can configure the applicable days of the week. For example: user can set the automation to only activated from Monday to Friday.

Source

Determine which device template the automation be applied to. The conditions checking will be applicable to devices of the specified template. However, user can also specify specific device(s) the automation will be applied to.

Condition

Set criteria for the device state or field value to execute the action. User can compare the field value with other field or with a constant. Logical operations OR and AND can be used to check for multiple conditions.

Furthermore, to prevent from repeating actions being executed, user can set a delay time before the next condition checking is executed.

8.1.3 Devices

In general, Ctrl communicates to your device through MQTT. Depending on the type of device, add your devices to Ctrl through one of the following steps:

- **Adding F1 devices:** *Zero-Touch Provisioning* or *Manual Provisioning*
- **Adding other devices:** Use the *Manual Provisioning* flow

Firmware Update

Note: this feature is only available for F1 Starter Kits

Once connected to Ctrl, it will detects the current firmware version of the Starter Kit. Information regarding the firmware version can be found in the following sections:

- Header part of each device details
- 'Firmware Update' section of the device's Deployment Management tab

You can perform Over-The-Air (OTA) firmware update by clicking the 'Update now' button on the 'Firmware Update' section of the device's Deployment Management tab. This button will be activated **if** the following conditions are fulfilled:

- The current firmware version is older than the latest version
- Device status is online

If you wish to deploy other firmware versions or if the device is currently offline, you can perform the update though USB connection using our Visual Studio Code CtrlR plugin.

File Management

Default File Structure

Following Micropython implementation on ESP 32, it uses a flat, Unix-like Virtual File System (VFS) where the primary flash storage is typically mounted at the root directory (/). By default, it consists of the following directories:

- **/ (Root): Contains two files that will be automatically searched and executed during boot up:**
  - boot.py: This script runs first upon power-on or reset. It is typically used for low-level system configuration, such as setting up network connections or mounting additional filesystems.

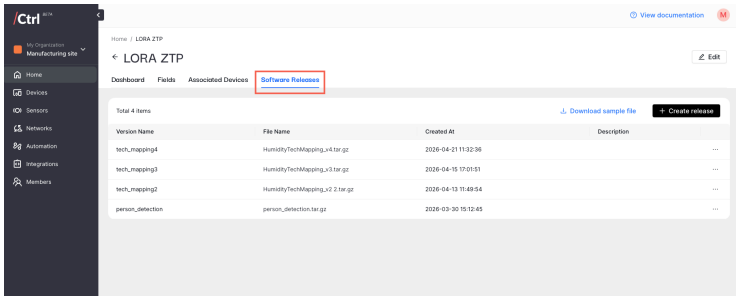
  - main.py: This script runs immediately after boot.py finishes. It contains your primary application logic and will restart automatically if it finishes execution.

- **/lib:** A standard location for third-party libraries and custom modules. Placing .py or .mpy files here allows them to be easily imported using the standard import statement.
- **/data (optional)** Often used by developers to store non-code assets like config.json, static web files, or log data.

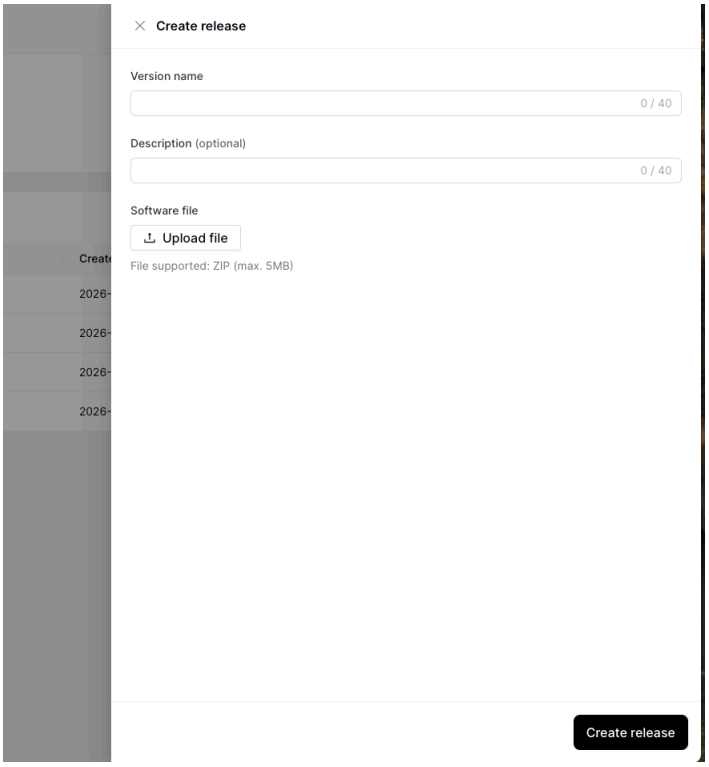
Software Releases (File Content Deployment)

You can remotely deploy new file(s) to the device's root directory through Ctrl through Software Release feature. File content management of F1 Starter Kit consists of two main parts:

- **Software releases management on Device Templates - one release for all devices under the same template.**
  - Go to your target device template, either by clicking 'device template in use' from the device details or by clicking one of the templates on the Home page's Device Template tab.
  - Go to the Software Releases tab. You will find a list of applicable release versions of the template.

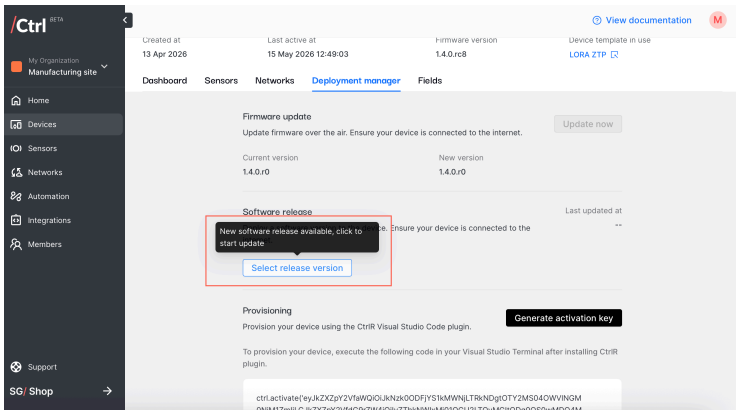


  - **To create a new release version:**
    - Compress all the file contents to be included into a zip file (maximum size: 1.2MB)
    - Click 'Create release' button
    - Type the version name and upload the zip file

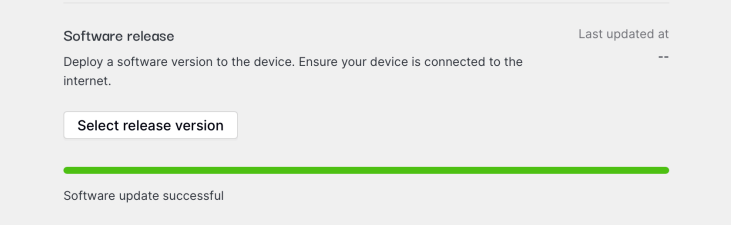


  - Select any devices under the Associated Devices tab to deploy the release.

- **Deployment manager tab on Device Details - over-the-air (OTA) software release**
  - Make sure that your device is connected to the internet over Wi-Fi or cellular connection
  - Go to Deployment Management tab of the target device details



  - Select the release version and click ‘Deploy release’ - device will be reset upon completion.



- **The OTA software release is not available under the following situations:**
  - Device is offline
  - A software release is in progress
  - Device is currently connected over LoRa - the file size is too large to be handled over LoRa connection

If OTA software release is not available, you can perform the deployment though USB connection using our Visual Studio Code CtrlR plugin.

Next Steps

- *Dashboards & Widgets*
- *Link sensors to your device*

8.1.4 Networks

For each project, you can save multiple LTE and Wi-Fi profiles for use across various devices through the Networks tab. This eliminates the need to repeatedly enter settings such as Wi-Fi passwords or LTE APNs during device provisioning.

Furthermore, you can implement these profiles remotely and manage your device's network settings remotely through Ctrl.

- For each project: manage your *network profiles*
- For each device: manage your *device network settings*

> **Note**
>
> Network settings through Ctrl are only enabled for SG devices at the moment.

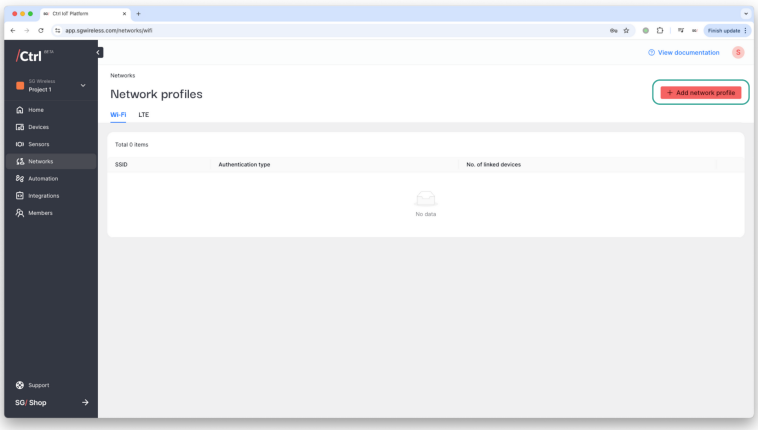
Manage Network Profiles

A *network profile* is a set of network configurations that can be implemented to multiple devices in the project.

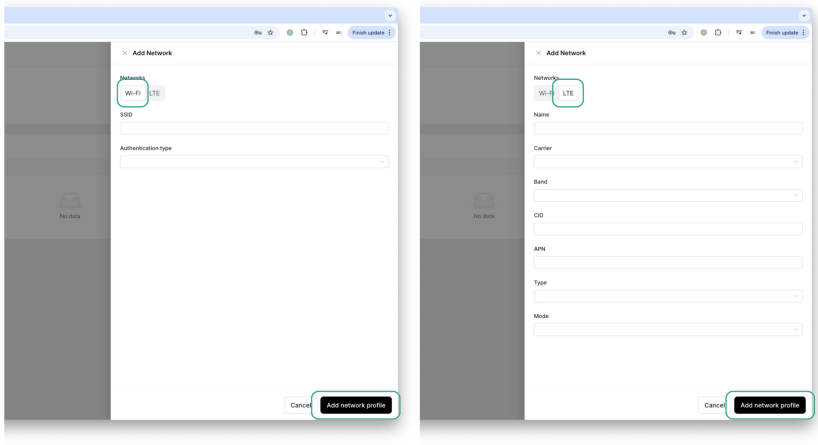
> **Note**
>
> If your device is provisioned via ZTP, an LTE profile will be automatically added for you, with name "1NCE".

Create new network profile

1. In the side menu, click “Networks”, then “Add network profile”.



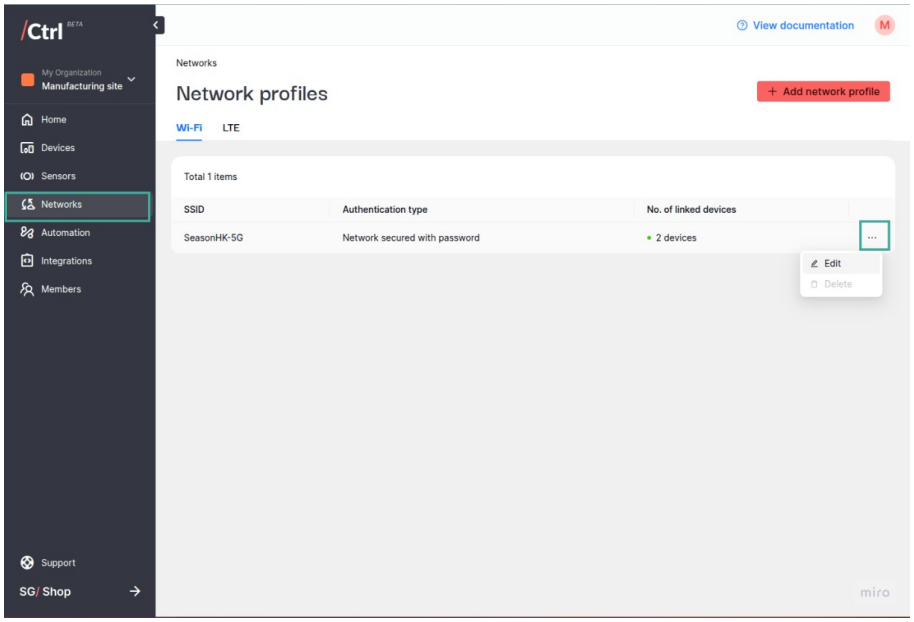
2. Choose “Wi-Fi” or “LTE”, and enter the required network credentials.



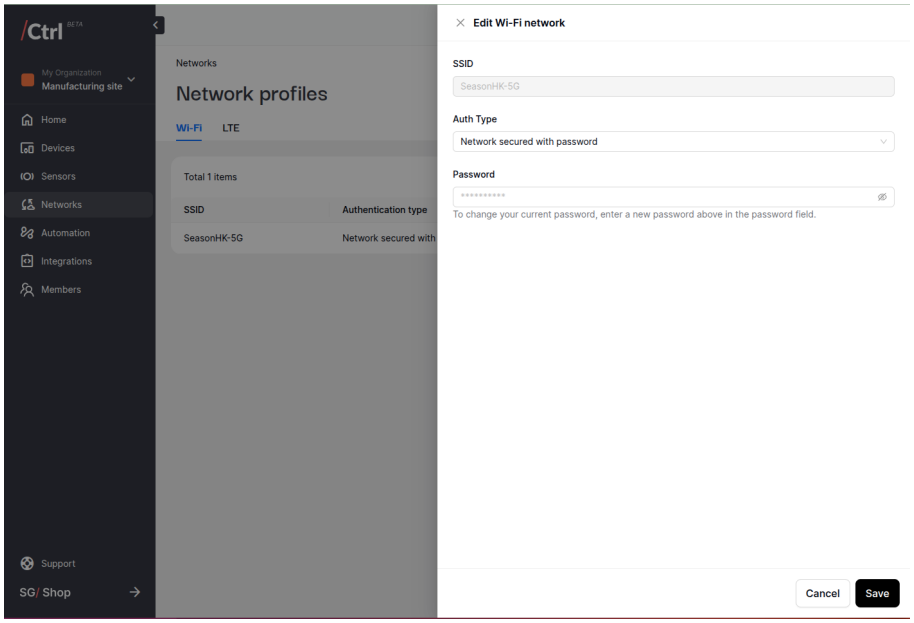
3. Click “Add network profile” once it’s done. Your new network profile will be available under the corresponding tab.

Edit network profile

1. In the side menu, click “Networks”.
2. Click the “…” icon on the right-most column of the target network, then “Edit”.



3. Make changes to the network credentials as needed. Note that **SSID** is **not editable for Wi-Fi** profiles while **name** is **not editable for LTE** profiles.



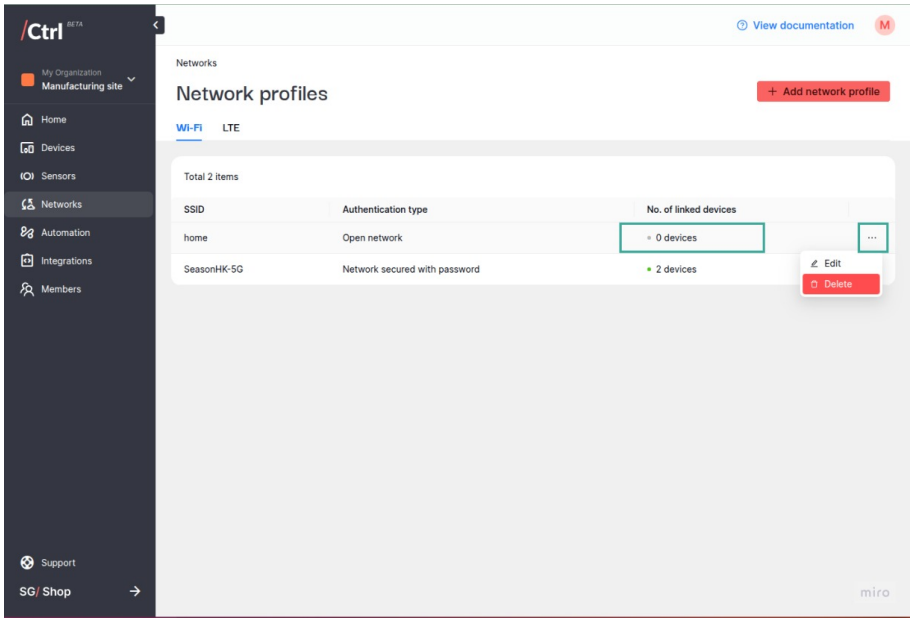
4. Click “Save” to apply changes.

> **Note**
>
> In order to implement the changes on your devices, you need to deploy the changes through each device’s network tab.

Delete network profile

A network can only be deleted if it is not being used by an existing device.

1. In the side menu, click “Networks”, then choose the target network.
2. Click the “…” icon on the right-most column of the target profile, then “Delete”.



Manage Device Network Settings

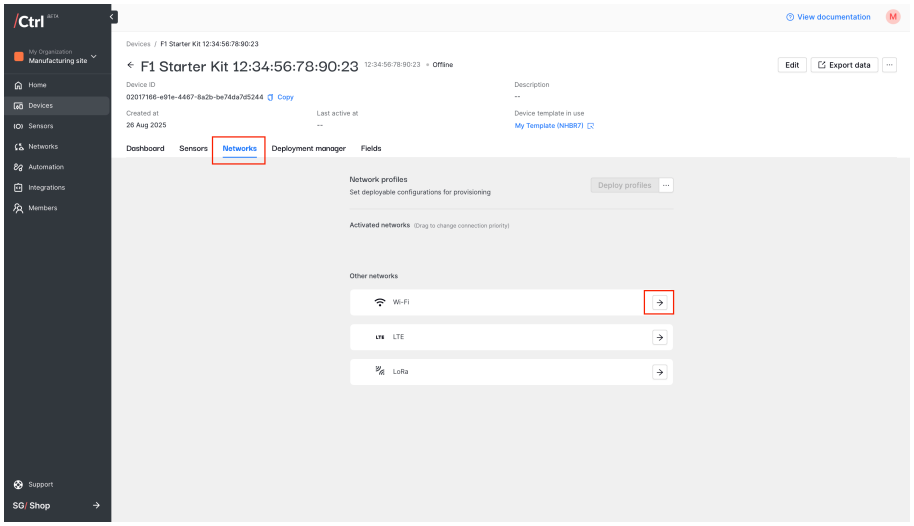
The F1 Starter Kit supports multiple network types that can be adjusted according to your need. You can *enable or disable* particular network types AND adjust the *network type priority* of your Starter Kit through Ctrl.

Accessing the device network settings

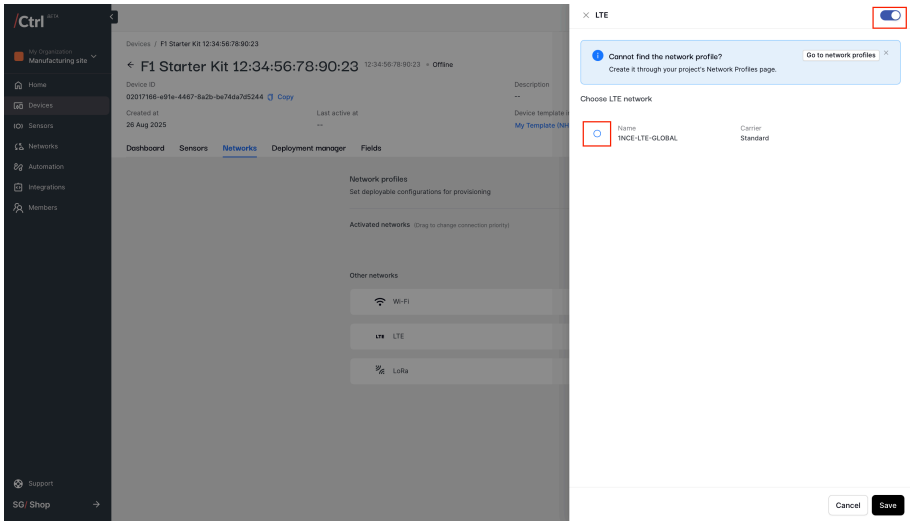
1. In the side menu, click “Devices”. Then, click on the target device.
2. Click on the device’s “Networks” tab.

Activate or deactivate a network

1. Click on the arrow icon on the target network.



2. Press the toggle on the top right corner.



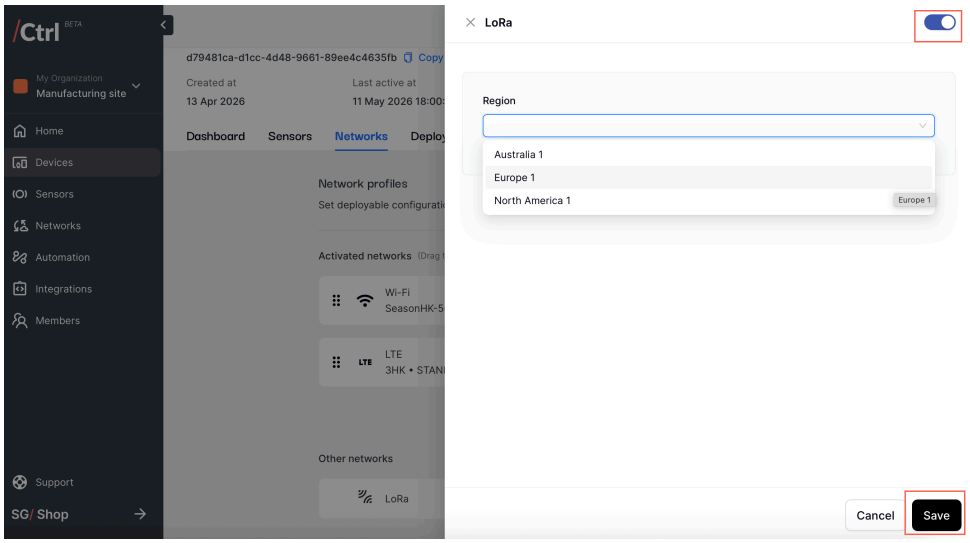
3. If you are activating the network, choose the preferred network profile before saving the update.
4. The activated networks are displayed under the “Activated networks” section, while the non-active networks are displayed under “Other networks” section.

Ctrl LoRaWAN integration

LoRa connectivity activation

Since Ctrl is integrated with SG Wireless TTN LoRaWAN, it will automatically register and connect your devices upon LoRa activation through Ctrl.

1. First, ensure you have a LoRaWAN gateway within accessible range of your device. Identify the region setting of the gateway and make sure that it’s also connected to the internet.
2. Once you activate the LoRa profile toggle, you will need to select the LoRaWAN region that matches with your LoRaWAN gateway region. Upon clicking the `save` button, Ctrl will automatically generate TTN device activation information, such as JoinEUI, DevEUI, AppKey, and NwkKey. This information is currently not visible by user in Ctrl. But it can be accessed by calling `ctr.print_config()` Ctrl Client endpoint through any IDE of your choice.



3. This device activation information will be passed to the device when you *deploy the network profile*.

If you wish to use your own LoRaWAN setup, you can use the programming resources on this page to configure the LoRa connection: Network Interfaces

LoRa Connectivity Limitations

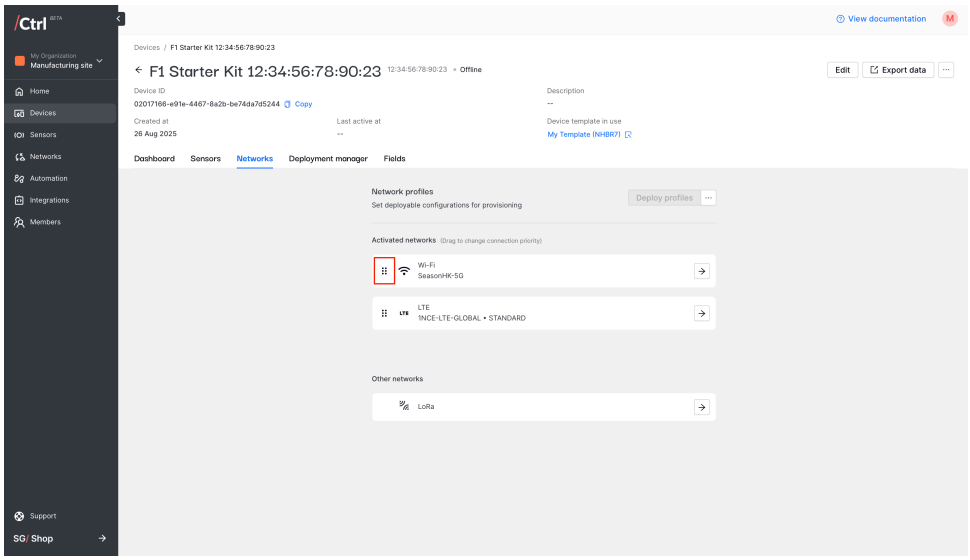
1. **Downlink data is not real-time.** As a LoRa Class A device, F1 Starter Kit always initiates communication from the device side and operates fully asynchronously, so downlink messages are not delivered instantly.
2. **LoRa's long-range, low-power design limits the size of data packets, so that these capabilities are not supported over LoRa:**
    a. Linking / unlinking sensors
    b. OTA firmware update
    c. OTA file content update
    d. OTA network settings update

Rejoining SGW LoRaWAN TTN Network

Supposedly, you only need to register your device to SG Wireless TTN network once during activation. However if your device looses connectivity from activities like firmware reflashing, the device needs to rejoin the network. While the device activation credentials remain unchanged, you might encounter errors like 'DevNonce is too small'. In this case, kindly contact us at info@sgwireless.com for support. We will help you to reset the DevNonce.

Adjust the device network priority

In the list of *activated networks*, click and hold the three lines on the left of the target network, and drag it to its desired priority.



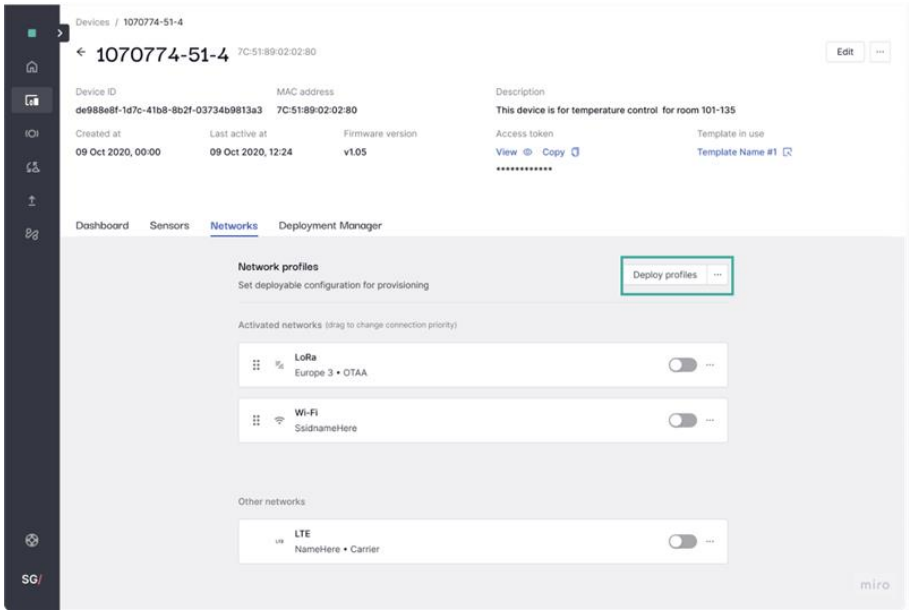
Note

This priority will only be used during the first-time connection. Automatic network switching is not supported after connection has been established.

Deploy the network setting

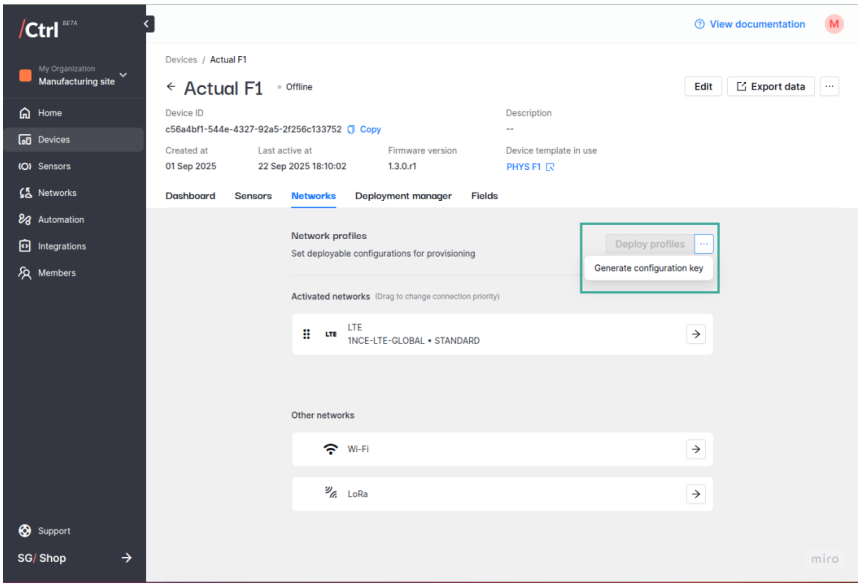
To implement the updated settings on your device, deploy it using one of the following approaches:

- **If your device is currently online:** press the “Deploy profiles” button to deploy the update over the air.



- **If your device is currently offline:**

1. Generate an activation key by clicking the button under the “…” icon of the device’s network tab.



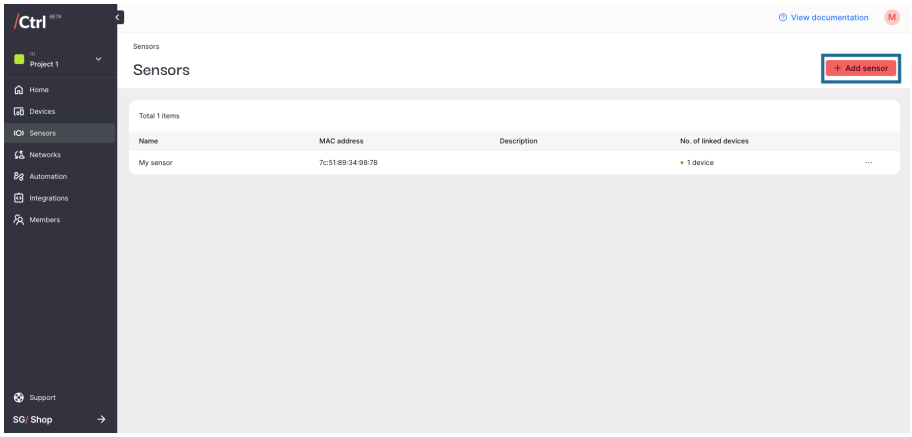
2. Copy the resulting activation code.
3. *Deploy the code through the CtrlR Visual Studio plugin*.

8.1.5 Sensors

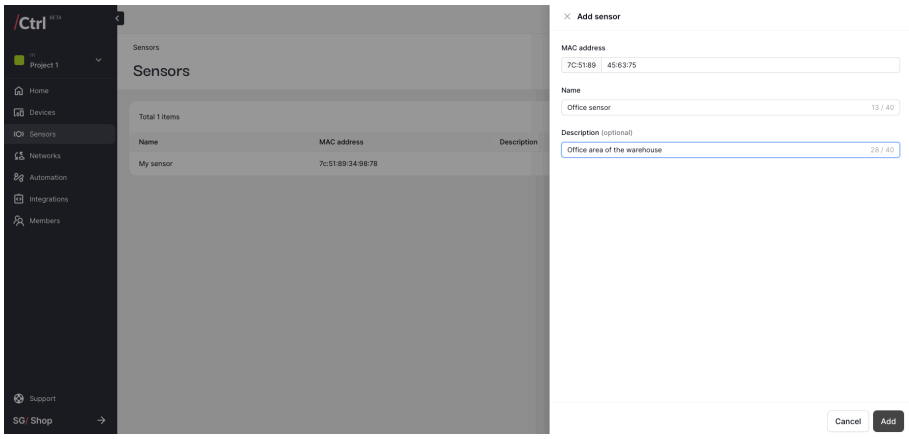
Add a Sensor to Ctrl

In this section, we will explain how to add an SG Wireless sensor to Ctrl.

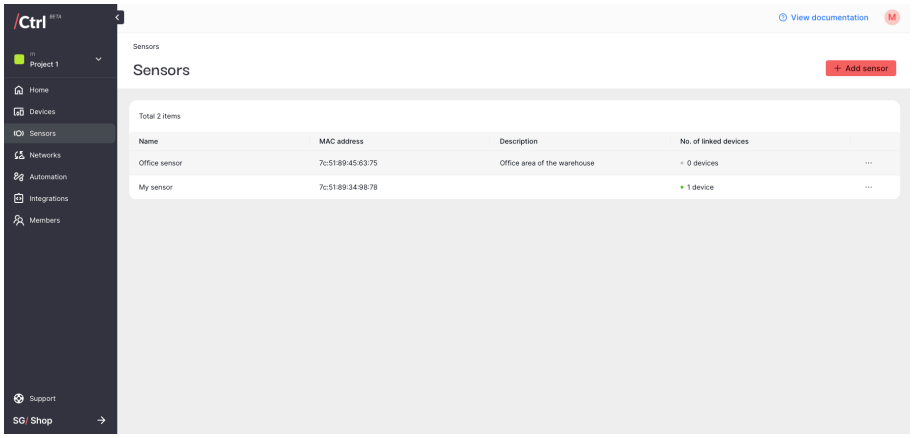
1. Navigate to **Sensors** and click on `Add sensor`.



2. Enter the Sensor's MAC address. You can enter just the last 3 segments.
3. Choose the device you want the sensor to be initially linked to.
4. Enter an appropriate `Name` and `Description` to better identify it.



5. The new sensor will appear on the `Sensors` page.



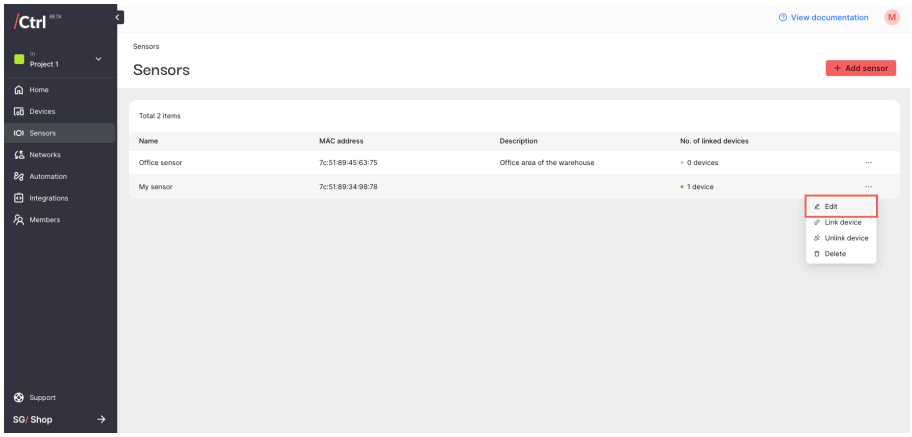
Each sensor can be linked to multiple devices, allowing for versatile integration. The data collected by the

sensor will flow through the connected devices and ultimately reach Ctrl.

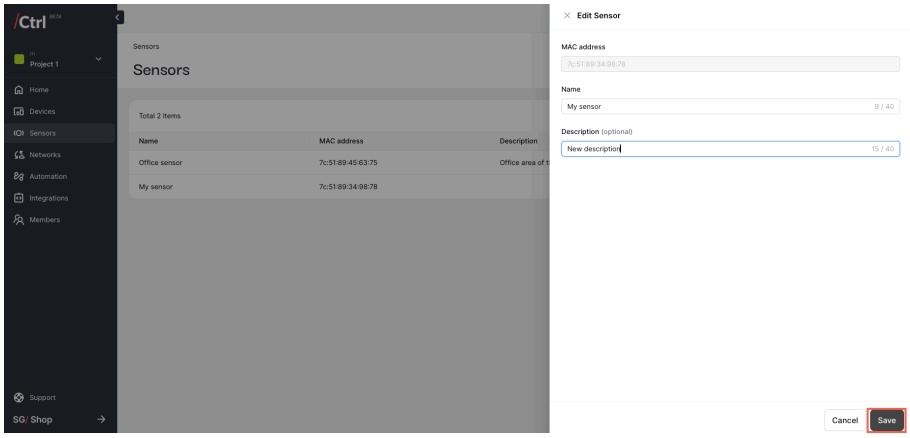
You can access Temperature and Humidity data from the sensor through the fields of the linked device.

Update Sensor Information

1. Navigate to the `Sensors` page.
2. Click on the "…" (ellipsis) icon next to the sensor you wish to edit.
3. Select the `Edit` option from the dropdown menu.

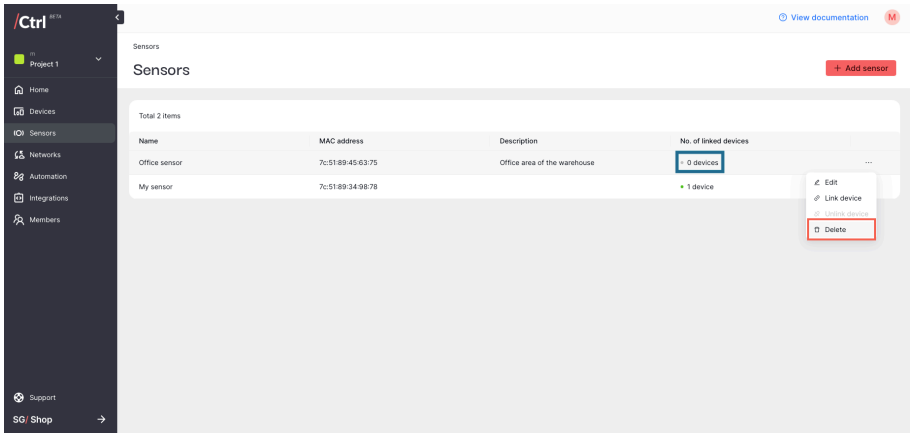


4. Update the `Name` or `Description` fields as needed.
5. Save your changes by clicking `Save`.

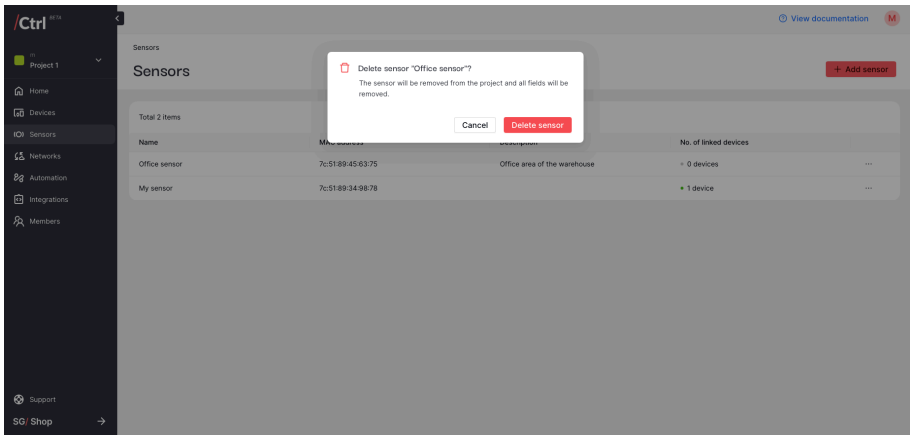


Remove a Sensor from Ctrl

1. Unlink all devices from the sensor.
2. Navigate to the `Sensors` page.
3. Click on the "…" (ellipsis) icon next to the sensor you wish to remove.
4. Select the `Delete` option from the dropdown menu. The option will only be enabled if there is no devices linked to it.



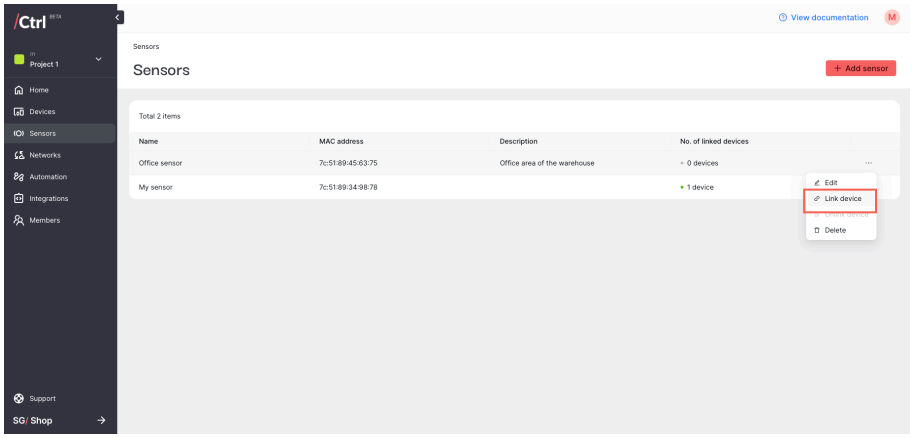
5. Confirm by clicking `Delete sensor`.
6. Deleting will unlink sensors from any linked devices and remove the sensor from the platform.



Link Sensor to a Device

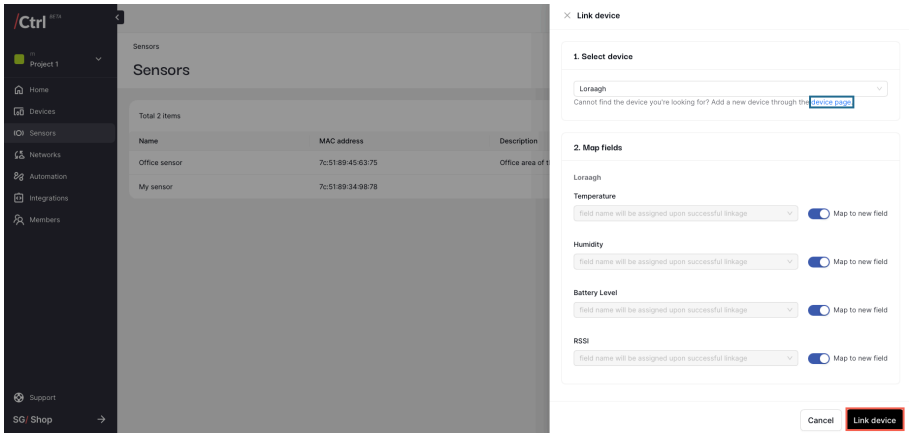
This action is necessary to send data from sensors to Ctrl. The linked device transmits the data readings with the sensor as the source.

1. Navigate to the `Device` page and select the device that you want to link to the sensor.
2. Go to the `Sensors` tab within the device page.
3. Click on `Link Sensor`.



4. Choose the sensor you want to link from the list provided.

5. Click the `Link device` button to confirm.

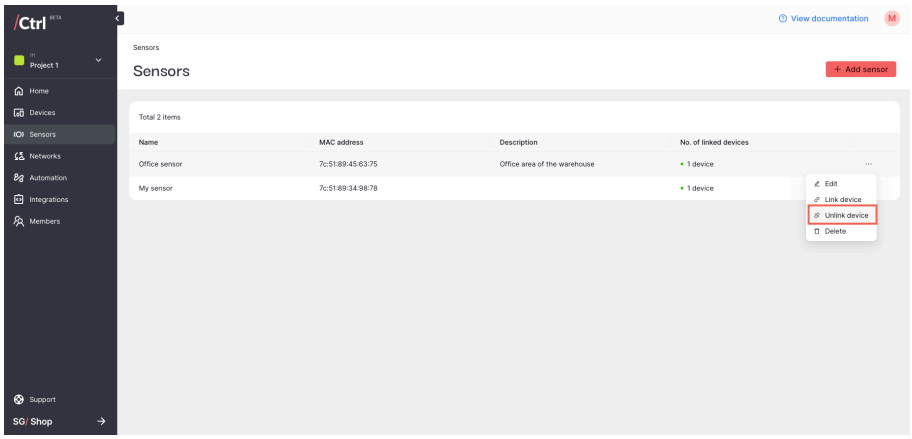


6. If the device is connected to the internet over Wi-Fi or LTE, Ctrl will send a message to the device over-the-air, instructing it to listen for the sensor's MAC address and establish the link. Otherwise, user need to update the configuration file through CtrlR.

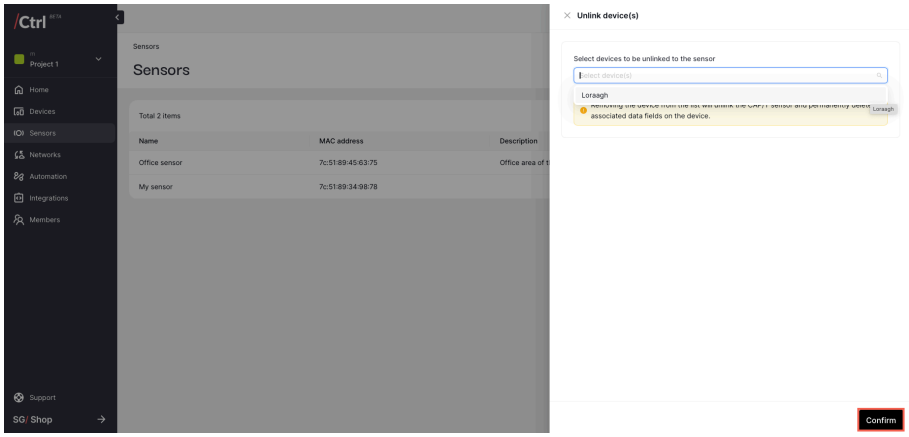
Unlink Sensor from a Device

After unlinking, no data readings will be sent to Ctrl from the device for this sensor.

1. Navigate to the `Device` page and select the device from which you want to unlink the sensor.
2. Go to the `Sensors` tab within the device page.
3. Click on the "…" (ellipsis) icon next to the sensor you wish to unlink.
4. Select the `Unlink device` option from the dropdown menu.



5. Confirm the action by clicking on the `Confirm` button in the confirmation dialog.



8.1.6 Dashboards & Widgets

Dashboards let you customize how to view incoming data exactly as you like it. After all, the true value of data lies in the insights that can be drawn from it, and having the data organized definitely helps!

> **Note**
>
> We're assuming that you have already connected your device to Ctrl. In case you haven't, check how to *add your device*. After you're done with that, you can proceed to the next example.

You can view data in various ways, using **widgets**. Below is an overview so you can see which widgets will be useful for your use case. Each widget can be configured, from the type of data it shows, to how this data is shown.

Once you have your widgets, you can arrange and rearrange them in your dashboard in **Edit Mode**, so the data that matters most is always on top.

- *Value Widget*
- *Boolean Widget*
- *Data Log Table Widget*
- *Bar Chart Widget*
- *Line Chart Widget*
- *Gauge Widget*
- *Map Widget (coming soon)*

Value Widget

Displays the latest data point of a specific data field.

By default, data points are displayed in black, but you can also set colour thresholds for specific values (such as green for normal readings, and red for abnormal readings).

Boolean Widget

Displays the binary state of a specified data field.

Useful for logical operations with a `True` or `False` condition, such as switch states (on/off) and device status (online/offline). You can customize the text and colours for each condition.

Data Log Table Widget

Presents incoming data in timestamped, chronological order.

Bar Chart Widget

Visualizes changes in data over time, useful for spotting patterns and trends that may not be obvious from individual readings.

Each bar chart supports up to 10 data fields. In each bar chart, you can customize:

- **Bar labels and colours**
- **Which Y-axis to use:** Helpful when comparing different types of data like temperature and humidity
- **Reference lines:** Up to 4 horizontal lines can be added for data comparison, such as average values or thresholds
- **Time range:** Choose from preset ranges (last hour, day, month) or set custom periods – shorter for fluctuations and seasonal patterns, longer for long-term trends. The minimum time interval is automatically set to keep the chart loading quickly.
- **Data aggregation:** Show maximum, minimum, or average values for each time interval
- **Display options:** On/off toggle for bar labels and axis labels

Line Chart Widget

Visualizes changes in data over time, useful for spotting patterns and trends that may not be obvious from individual readings.

Each line chart supports up to 10 data fields. In each line chart, you can customize:

- **Line labels and colours**
- **Which Y-axis to use:** Helpful when comparing different types of data like temperature and humidity
- **Reference lines:** Up to 4 horizontal lines can be added for data comparison, such as average values or thresholds
- **Time range:** Choose from preset ranges (last hour, day, month) or set custom periods – shorter for fluctuations and seasonal patterns, longer for long-term trends. The minimum time interval is automatically set to keep the chart loading quickly.
- **Data aggregation:** Show maximum, minimum, or average values for each time interval
- **Display options:** On/off toggle for line labels and axis labels

Gauge Widget

Displays the latest data point against a performance metric.

You can choose from radial, horizontal, or vertical styles, and set threshold ranges with custom labels and colours.

Map Widget (coming soon)

Displays geolocation data.

Arranging Your Widgets

Edit Mode lets you arrange your widgets for a custom dashboard that works for you.

- **Add widgets** to your dashboard
- **Edit widgets** to update their configuration
- **Delete widgets** you no longer need
- **Resize widgets** to emphasize the most important data
- **Reposition widgets** by dragging them to the desired location

8.1.7 Integrations

Ctrl offers a way to interact with external platforms and services, including the following:

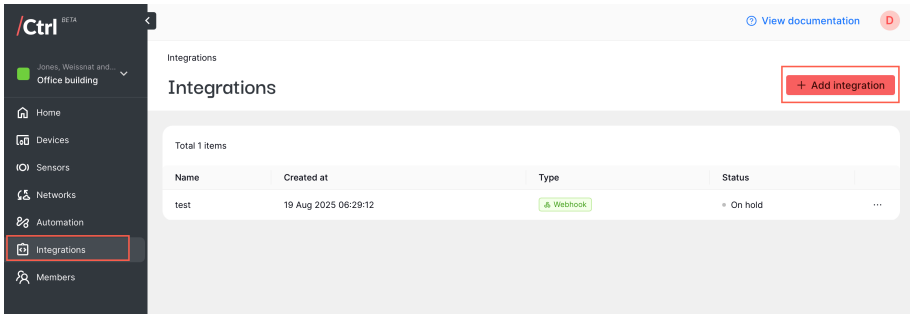
- *Webhook* – allows for user-defined HTTP callbacks to a defined remote destination. All elements of the requests (headers, query string parameters, etc.) are customisable.
- *AWS IoT Core* – a managed cloud platform that lets connected devices easily and securely interact with Cloud applications and other devices.
- *Azure IoT Hub* – a comprehensive collection of services and solutions designed to help you create end-to-end IoT applications on Azure.

Webhook

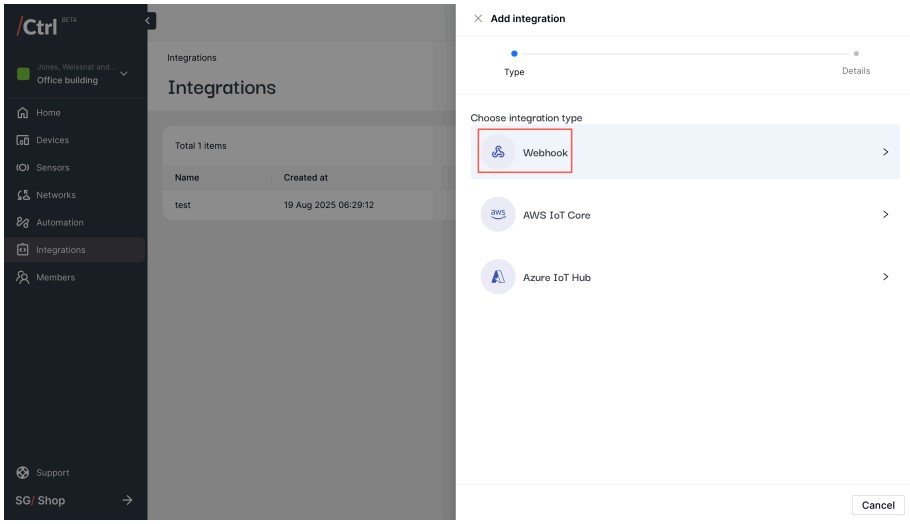
Whenever one of your integrated devices sends a signal to Ctrl, we perform an HTTP request defined by the user. You can use some presets (`DEVICE_TOKEN`, `USER_ID`, etc.), which will act like placeholders and will be dynamically replaced at the moment of performing the request with the relative content.

Setting up your integration

1. Click **Integrations** on the side menu, then click “Add integration” on the top right corner.

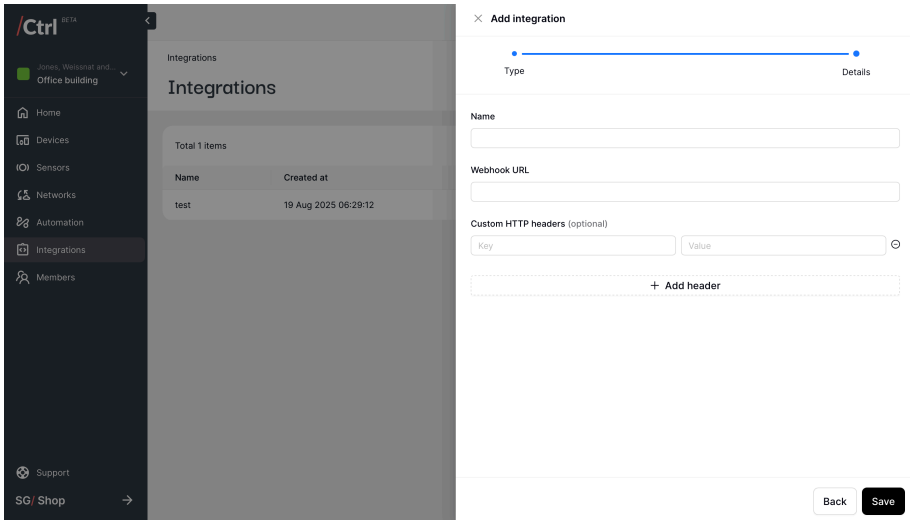


2. Select **Webhook** as the integration type.



3. Input a friendly name to identify the webhook integration and the webhook URL where the payloads will be sent to. Additionally, you can configure your custom HTTP headers.

> **Note**
>
> For testing, you can use the URL generated from https://webhook.site/.



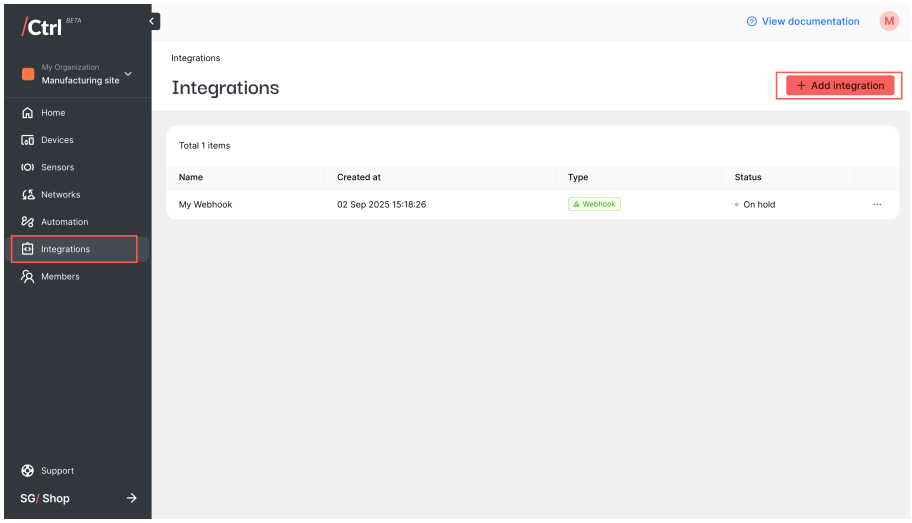
4. Click “Save” when you are done. You should be able to see the incoming Ctrl telemetry data on your destination system.

AWS IoT Core

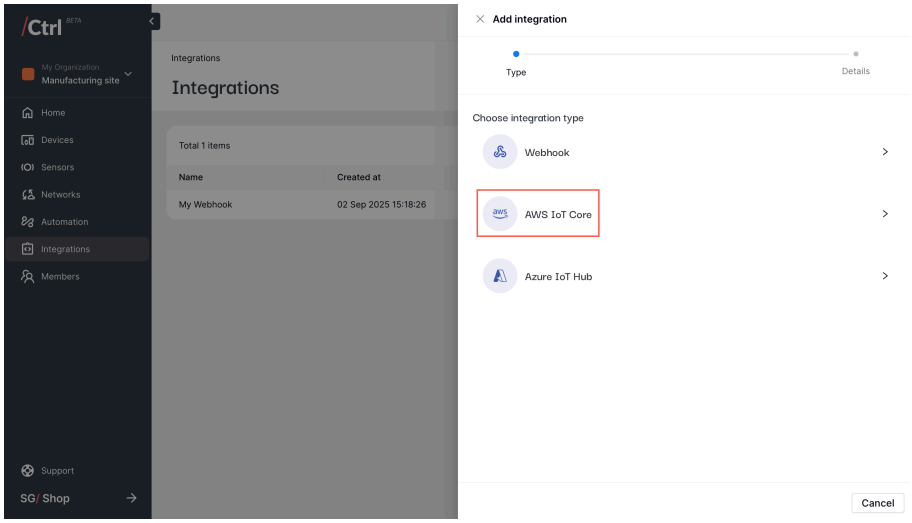
Whenever one of your integrated devices sends a signal to our broker, we republish the binary payload to the endpoint specified for its integration through MQTT connection.

Setting up your integration

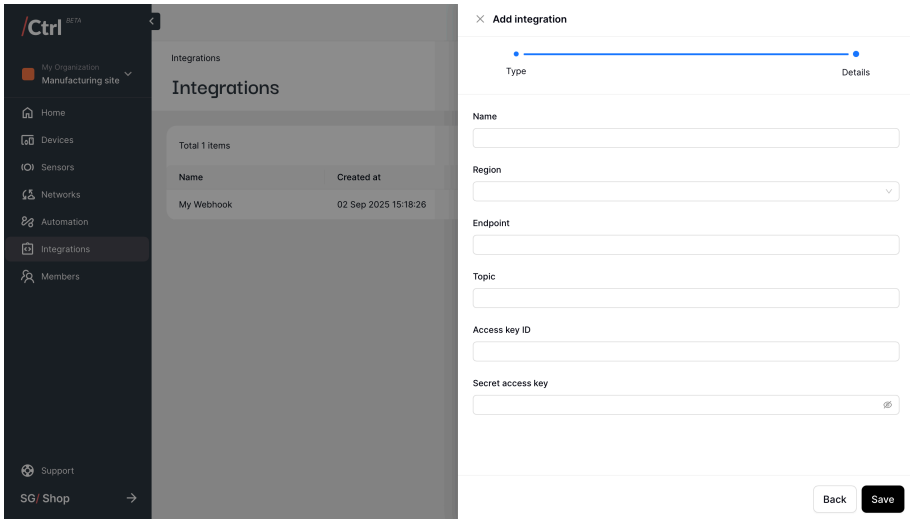
1. Click **Integrations** on the side menu, then click “Add integration” on the top right corner.



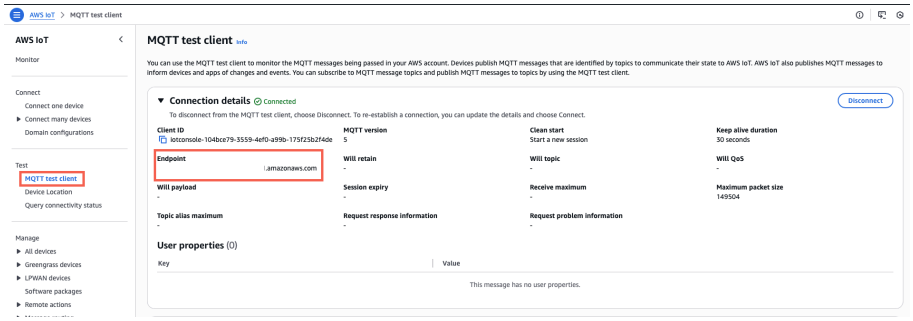
2. Select **AWS IoT Core** as the integration type.



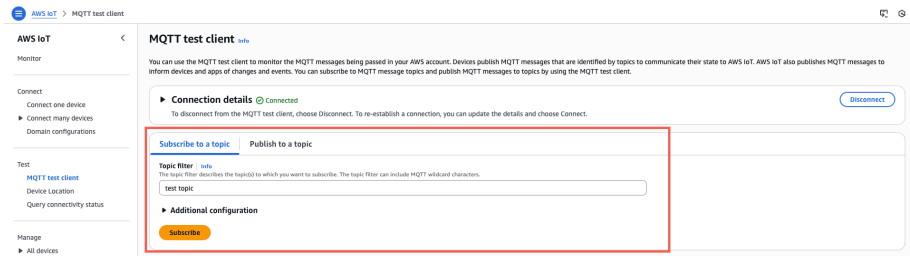
3. Complete your AWS IoT Core information.



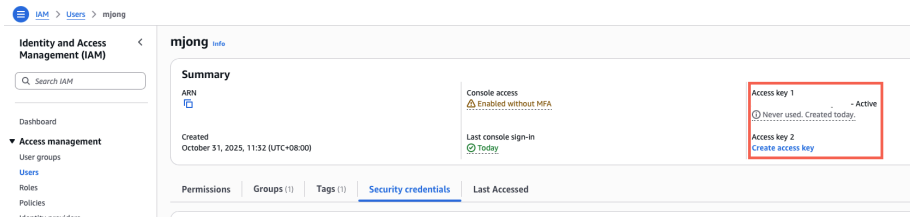
- **Name:** User-friendly name to identify a specific integration config.
- **Region:** Region of your AWS instance.
- **Endpoint:** AWS IoT Core MQTT endpoint found on the “Connection details” section on the upper part of the “MQTT test client” tab.



- **Topic:** MQTT subscription topic that you define on the MQTT test client page.



- **Access key ID and secret access key:** You can find the access key ID and secret through the user account details page of the IAM service. Steps to manage the access keys can be found here.



4. Click “Save” once you’re done. From now on, you can start seeing the incoming telemetry data through the MQTT test client page of your AWS IoT service page.

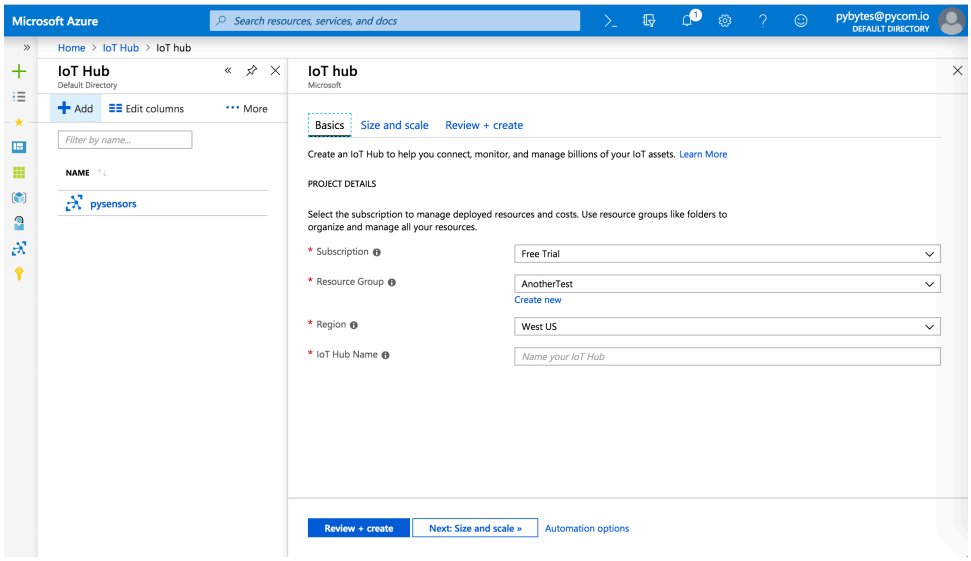
Azure IoT Hub

Whenever one of your integrated devices sends a signal to our broker, we republish the binary payload to the endpoint specified for its integration through the Azure IoT Hub SDK.

Set up your IoT Hub

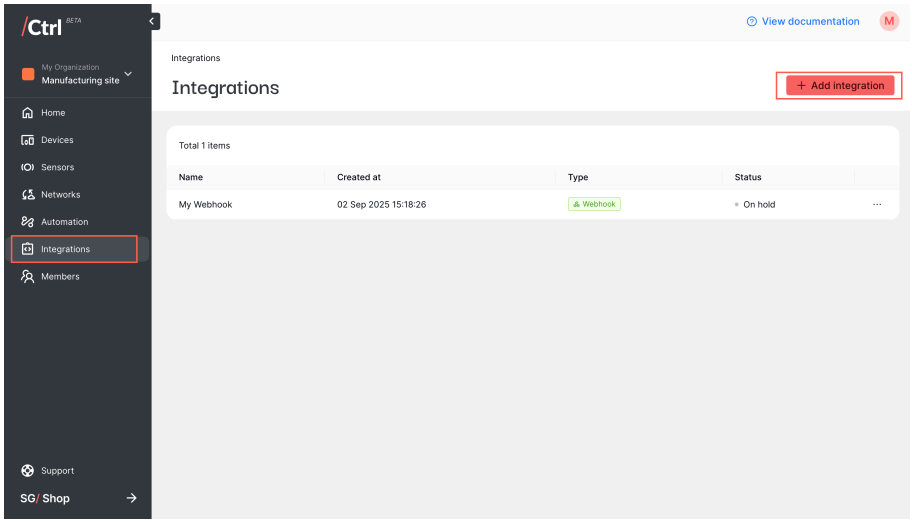
The first step requires you to create an IoT Hub. This is an Azure service that enables you to gather high volumes of telemetry data from your IoT devices. It then moves them into the cloud for storage or processing. In order to do that, follow the official documentation. To summarise, you'll need to:

- Specify your subscription plan.
- Create or choose a resource group. This contains resources that share the same lifecycle, permissions and policies. The name can contain alphanumeric characters, periods, underscores, hyphens and parentheses. It cannot end in a period.
- Choose a region.
- Choose an IoT Hub name (its length must be between 3 and 50, and it can use only alphanumeric characters and hyphens). It won't be possible to change this name later.
- Specify tier scaling and units.

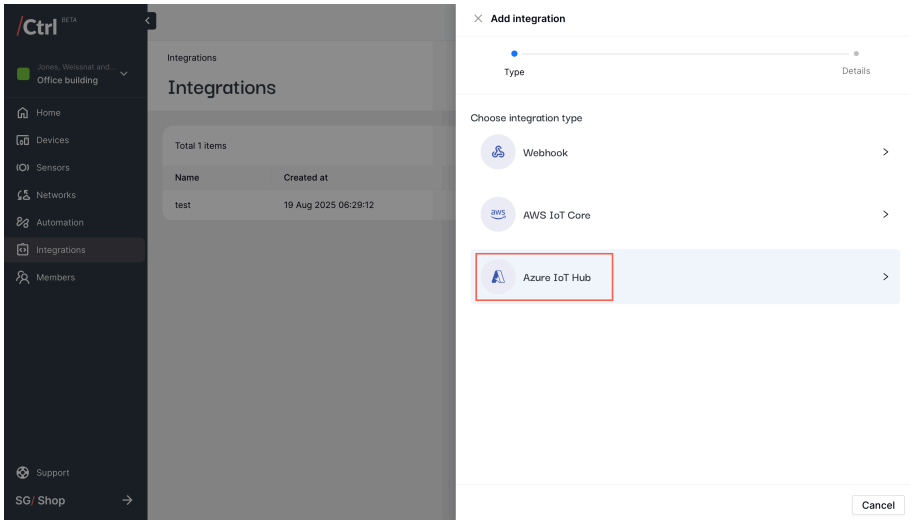


Setting up your Ctrl integration

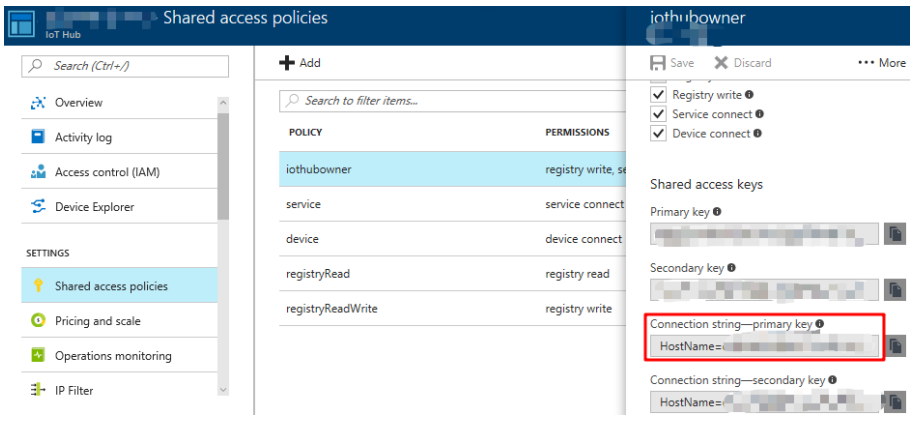
1. On your Ctrl page, click **Integrations** on the side menu, then click "Add integration" on the top right corner.



2. Select **Azure IoT Hub** as the integration type.

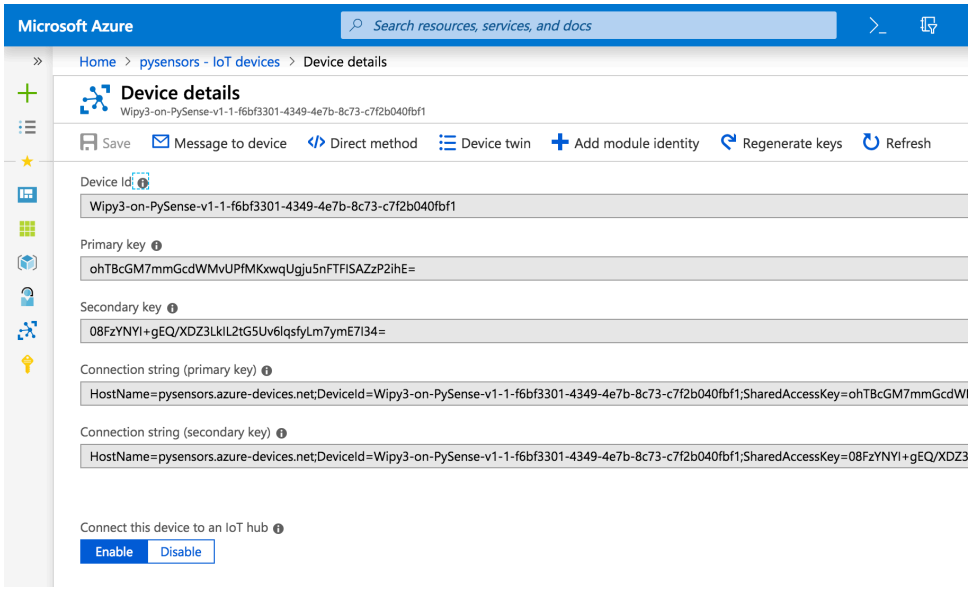


3. Add a user-friendly name to identify the integration instance, then copy and paste the connection string of your IoT Hub to the Ctrl form.



4. Click “Save” once you are done.

The corresponding device has been created in Azure as well. You just have to log in to the portal, click on its IoT Hub and then click on the device just created. You should be able to see all the device’s details, also the connection string which will be saved encrypted in our database and used to republish your data to your Azure IoT Hub.



Try to send some signal messages with your device. You should be able to see in the dashboard that the system has received them. More information on testing device connectivity can be found here.

> **Warning**
>
> Do not delete Azure devices directly from the Azure user interface, otherwise the integration with Ctrl will stop working. Always use the Ctrl interface to delete Azure devices.

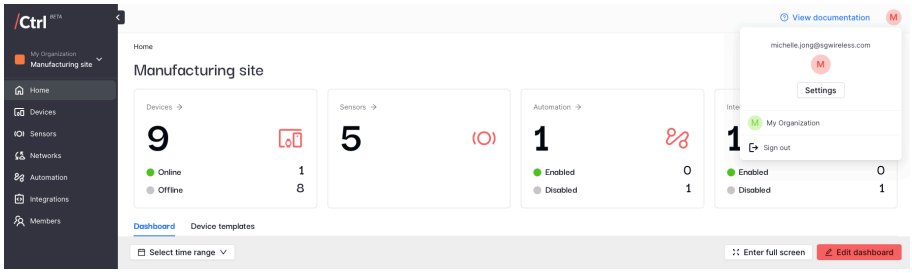
8.1.8 Account Settings

You can modify your personal account settings and/or organization settings on Ctrl. Personal account settings are accessible for everyone, while the organization settings are only open to users who have `Store Owner` or `Admin` role.

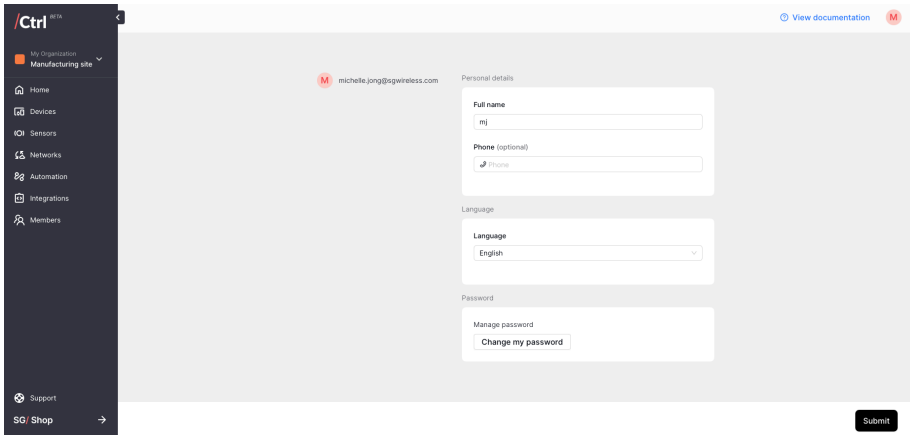
Update Personal Account Settings

Update Personal Profile

1. Click on the user icon at the top-right corner, and choose `My Profile`.

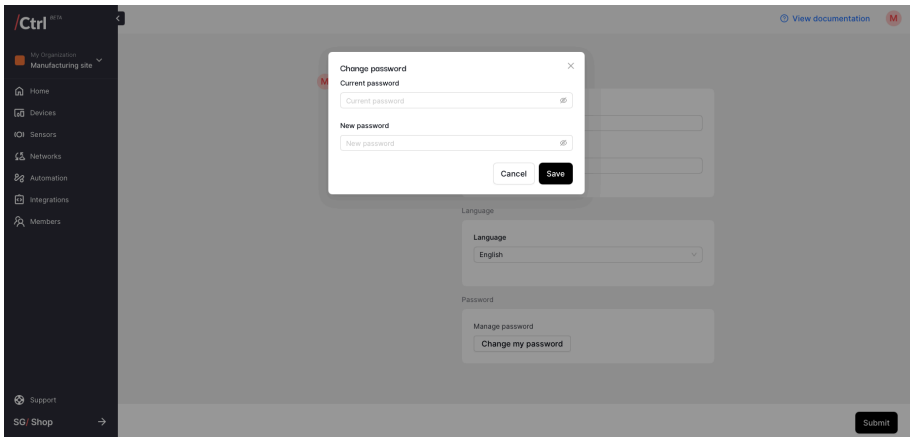


2. Under the `Account` tab, expand the `Profile` section. You can update your first and last name here. This change can be viewed by others in your organization as well.



Update Login Credentials

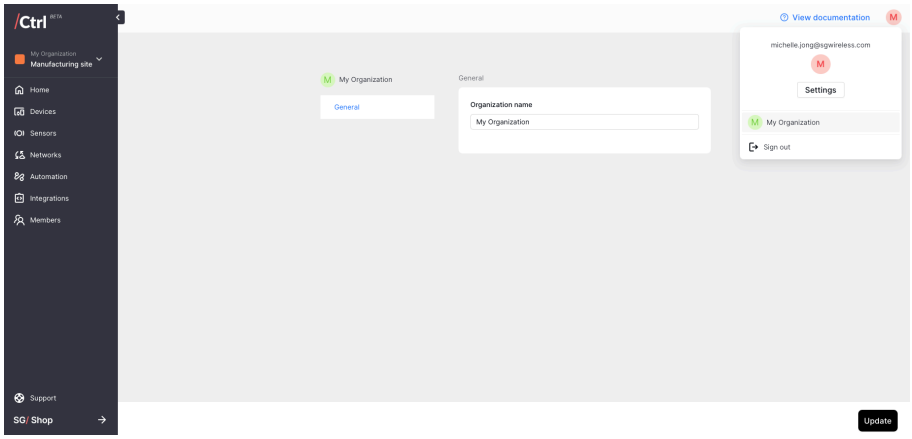
1. Click on the user icon at the top-right corner, and choose `My Profile`.
2. Under the `Account` tab, expand the `Credentials` section. You can update your password to Ctrl here. It is recommended for you to change your default password for safety reasons.



Update Organization Settings

Update Organization Profile

1. Click on the user icon at the top-right corner, and choose `My Profile`.
2. Under the `Organization` tab, expand the `Organization` section. You can update your company name, address, and phone number here.



8.2 CtrlR Visual Studio Plugin

A specialized extension for Microsoft Visual Studio Code (VS Code), which acts as a dedicated Integrated Development Environment (IDE) for SG Wireless devices.

- Allows your computer to easily detect and connect to the F1 Starter Kit via a USB-UART bridge.
- Provides a built-in REPL (Read-Eval-Print Loop) terminal.
- Enbles file management for your SG Wireless devices.

CtrlR is designed to bridge the gap between the physical F1 Starter Kit hardware and the Ctrl Cloud Platform.

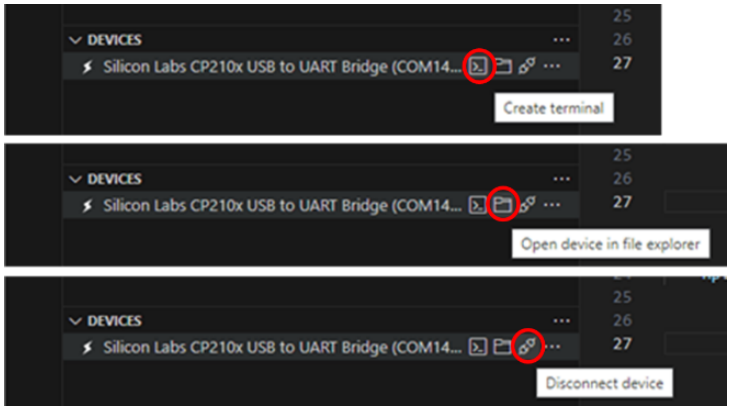
8.2.1 Programming with CtrlR

This guide shows how to run codes on SG Wireless devices through CtrlR.

Sending Commands Through Visual Studio Terminal

F1 smart module pre-installed MicroPython as operating system (OS), which equips with REPL. REPL stands for Read-Eval-Print Loop which is an Interactive Interpreter Mode that allows you to input code, execute it, and immediately see the results.

Using the CtrlR Plugin, click the left-most icon on your device to invoke the terminal:



There should be a blank screen with a flashing cursor. Press Enter and a MicroPython prompt should appear, i.e. >>>.

```
MicroPython v1.19.1-796-gf4811b0b4 on 2024-01-06; SGWireless SGW3501 with ESP32S3
Type "help()" for more information.
>>>
>>>
>>> os.uname()
(sysname='SGW3501', nodename='SGW3501', release='0.2.0.b0', version='v1.19.1-796-gf4811b0b4 on 2024-01-06', ma
chine='SGWireless SGW3501 with ESP32S3')
>>>
```

Let's make sure it is working with the obligatory test (you don't need to type the >>> symbol):

```
print("Hello F1!")
```

Once you type the code above, then press Enter and the following output should appear on screen:

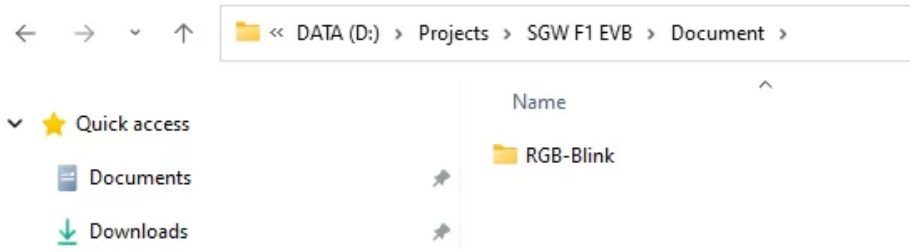
```
Hello F1!
```

Note that this command will only be executed once. It will not be stored anywhere on your device.

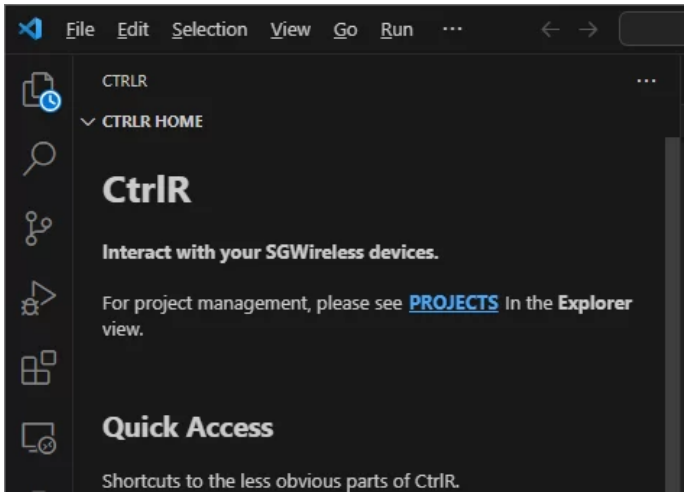
Creating a project in CtrlR

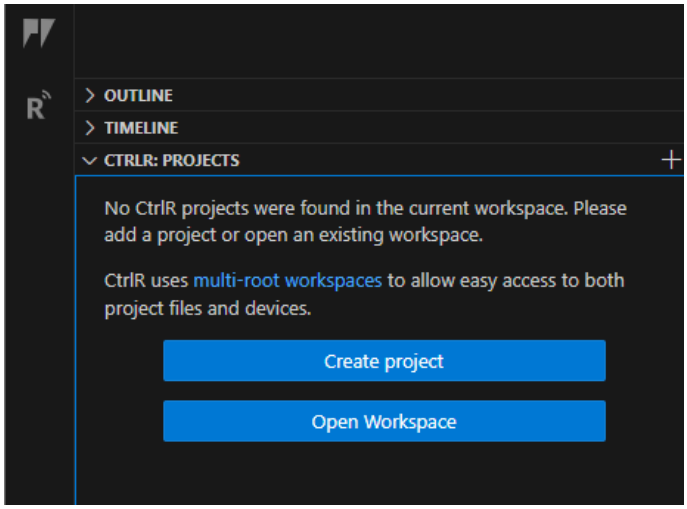
A project is simply a folder containing your application code (typically main.py) and optional supporting files.

1. In a known destination on your computer, create a new project folder called **RGB-Blink**.

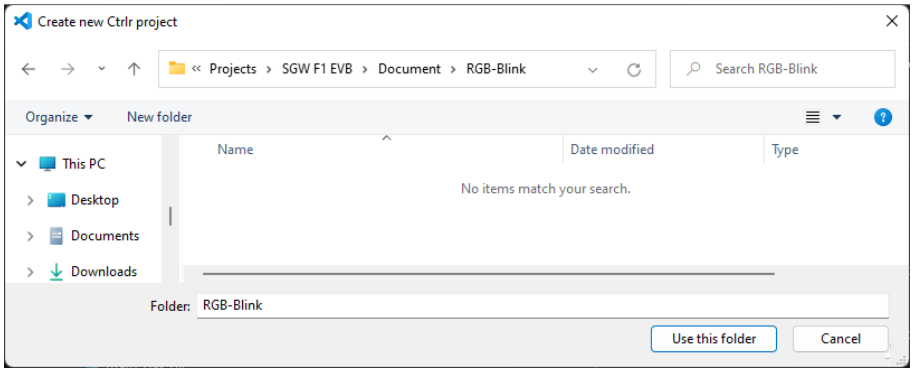


2. Launch VS Code and open the **RGB-Blink** project folder you created.



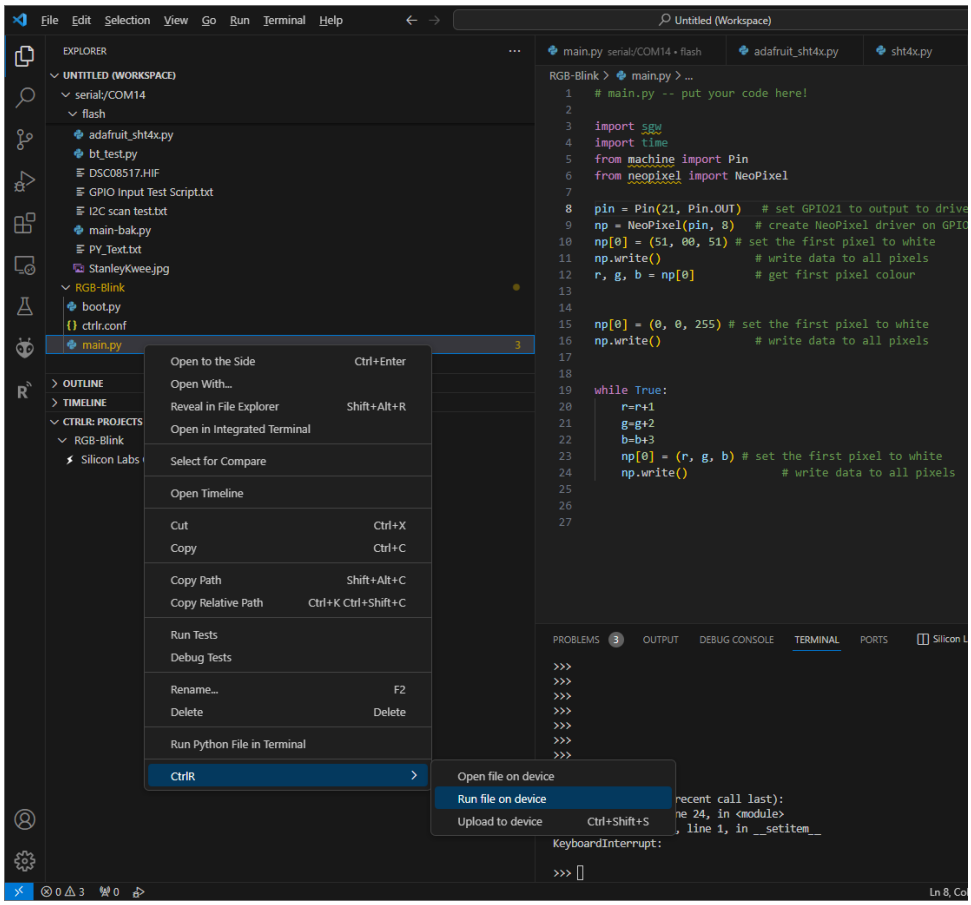


3. Create a new file called `main.py` and add your code. You can use the following code as an example:

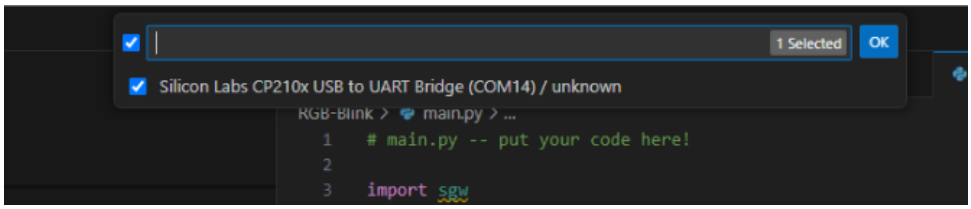



```
import time

colours = [0xff0000, 0x00ff00, 0x0000ff]  # Red, Green, Blue

while True:
    for colour in colours:
        rgbled.color(colour)
        time.sleep(1)
```

Running code on F1 Starter Kit before uploading



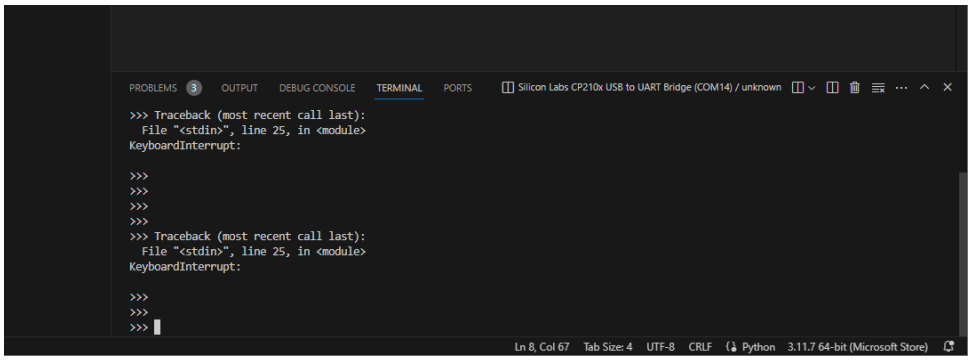
This action sends the script to the F1 Starter Kit and runs it immediately. The script is **not** stored on the device and will not persist after a reboot. This is useful for testing.



You should see that that on-board LED now blink in red, green and blue indefinitely.

Due to the infinite While-loop implemented on the script, it will run forever. Click onto the CtrlR terminal, and press `ctrl-c` on your keyboard to stop the script.

Uploading code to your F1 Starter Kit



This option uploads the script to the F1 Starter Kit's file system. The script will persist and run on boot. Use this when your code is ready for deployment.

CHAPTER

NINE

PROGRAMMING REFERENCES

This section documents the MicroPython modules available on the SG Wireless F1 platform. Each module is implemented as a built-in C extension and can be imported directly from the MicroPython REPL or from user scripts.

9.1 Application

9.1.1 CTRL API Documentation

Contents

- Sending Fields
- Configuration
- Connection
- Miscellaneous
- Examples

Fields

ctrl.send_field(pin_number, value, [timestamp=0, device_token=None])

Send a field value to CTRL. Arguments are:

- `pin_number`: The pin/field number in CTRL, can be any integer value
- `value`: The value you want to send, this can be any type (int, float, string, etc.)
- `timestamp`: Optional. Unix timestamp in seconds. If set to 0 (default), the server will use the current time
- `device_token`: Optional. Device token for sending data to a different device

ctrl.send_field_map(map, [timestamp=0, device_token=None])

Send multiple field values to CTRL in a single message using a dictionary/map. Arguments are:

- `map`: A dictionary where keys are pin/field numbers and values are the data to send. Example: `{1: 25.5, 2: 60.3, 3: "online"}`
- `timestamp`: Optional. Unix timestamp in seconds. If set to 0 (default), the server will use the current time
- `device_token`: Optional. Device token for sending data to a different device

ctrl.send_ping_message()

Sends a ping (is-alive) message to CTRL. The platform will answer with a `pong` message if connected via WiFi or LTE-M

ctrl.send_info_message()

Send an info message to CTRL containing the device type and firmware version.

ctrl.send_battery_level(battery_level)

Sends the battery level to Ctrl. The argument `battery_level` can be any integer.

You can define `battery_level` with a function depending on your hardware.

```
def battery_level():
    return 3.7
ctrl.send_battery_level(battery_level())
```

Configuration

If CTRL support is active in the current firmware (check `import ctrl_cfg; ctrl_cfg.ctrl_on_boot()`) it will load automatically. It will first look for a file `ctrl_config.json` in the file system.

If the file is found and the configuration looks valid, CTRL will try to connect to the cloud platform based on the configured parameters. The user can upload a file `ctrl_project.json` onto the local `/` to overwrite any of the parameters from `ctrl_config.json`. This allows for project specific settings to be configured such as forcing SSL to be enabled or to disable automatically starting the ctrl client on boot.

If no valid configuration is found, ctrl will load with an empty configuration to allow for the `ctrl.activate()` command to be executed. This will allow for the device to be activated via the python cli.

To manually load the CTRL client from your own scripts, the following code shows how it is loaded from the build-in frozen code.

```
import ctrl_cfg
if ctrl_cfg.ctrl_on_boot():
    import os
    import sys

    if 'ctrl_config' not in globals().keys():
        from ctrl_config import CtrlConfig
        from ctrl import Ctrl

        ctrl_config = CtrlConfig().read_config()

    if (not ctrl_config.get('ctrl_autostart', True)) and ctrl_config.get(
↪'cfg_msg') is not None:
        print(ctrl_config.get('cfg_msg'))
        print("Not starting CTRL as auto-start is disabled")
```

(continues on next page)

(continued from previous page)

```
    else:
        # Load CTRL if it is not already loaded
        if 'ctrl' not in globals().keys():
            ctrl = Ctrl(ctrl_config, ctrl_config.get('cfg_msg') is None,␣
↪True)
```

The CTRL API offers several helper functions to work with the configuration:

ctrl.read_config([filename='/ctrl_config.json', reconnect=False])

Load the CTRL configuration file. By default, this is loaded from `/ctrl_config.json` If reconnect=True, ctrl will disconnect and re-connect using the new configuration

ctrl.get_config([key=None])

Returns the configuration. If key is specified, only configuration for the given key is returned.

ctrl.update_config(key, [value=None, permanent=True, silent=False, reconnect=False])

Update a `key` and `value` of the default configuration file. This will **update** the existing configuration setting to add the new values.

additional options:

- `permanent`: will call `ctrl.write_config()`. If set `False`, the new value will not be stored in the configuration file and only used this session.
- `silent`: set `silent` to `True` to not print a message to REPL.
- `reconnect`: calls `ctrl.reconnect()`

ctrl.set_config(key, [value=None, permanent=True, silent=False, reconnect=False])

Set a `key` and `value` of the default configuration file. This will overwrite any existing settings for the specified key.

additional options:

- `permanent`: will call `ctrl.write_config()`. If set `False`, the new value will not be stored in the configuration file and only used this session.
- `silent`: set `silent` to `True` to not print to REPL.
- `reconnect`: calls `ctrl.reconnect()`

ctrl.write_config([file='/ctrl_config.json', silent=False])

Writes the updated configuration to the default configuration file. The parameters:

- `file`: The file name and location
- `silent`: set `silent` to `True` to not print to REPL.

ctrl.print_config()

Print the configuration settings to the REPL. This is easier to read for a human

ctrl.activate(activation_string)

Activate ctrl with the configuration pasted from the CTRL platform (under device/provisioning)

Connection

ctrl.start([autoconnect=True])

This will manually start the ctrl client, with the option to set `autoconnect`. Setting `autoconnect` to `False` will not start the connection immediately.

ctrl.connect()

Connect the device to CTRL following the loaded configuration file. You will need to load a configuration file before calling this. If you are using the WiFi or LTE-M connection, and it is already available, CTRL will use the existing connection.

ctrl.enable_lte(carrie, apn, [type='IP', cid=1, band=None, bands=None, mode=0, fallback=False])

Enable connecting via LTE-M connection to CTRL. Enter the paramters you would normally enter for an LTE connection. If fallback is True, will add LTE-M as the last option in the list of networks. Otherwise, it will be added as the first option and the device will connect via LTE-M after reset.

ctrl.enable_wifi(ssid, [password=None, fallback=False])

Enable connecting via WiFi to CTRL. Enter the paramters you would normally enter for a WiFi connection. If fallback is True, will add WiFi as the last option in the list of networks. Otherwise, it will be added as the first option and the device will connect via WiFi after reset.

ctrl.connect_lte()

Manually connect to CTRL using LTE and the settings from the configuration file.

ctrl.connect_wifi([timeout=120])

Manually connect to CTRL using WiFi and the settings from the configuration file. The `timeout` option is in seconds.

ctrl.connect_lora_otaa([timeout=120])

Manually connect to CTRL using LoRa OTAA and the settings from the configuration file. The `timeout` option is in seconds.

ctrl.disconnect()

Disconnect from CTRL gracefully. Closes the MQTT connection and socket.

ctrl.reconnect()

Calls `ctrl.disconnect()` followed by `ctrl.connect()`

ctrl.isconnected()

Returns the connection status to CTRL, can be `True` or `False`.

ctrl.ifconfig()

Returns a tuple with IP information when connected over WiFi or LTE-M

ctrl.enable_ssl()

Enable SSL on the CTRL connection

> Note that SSL might not be supported by your LTE connection
>
> **Note that SSL is not currently supported by the CTRL platform**

ctrl.dump_ca([file='/cert/sgw-ca.pem'])

Write CTRL ROOT CA certificate to file. In order for the firmware to load the certificate, it needs to be present in the file system. While the firmware has this CA embedded, it needs to be written to the file system in order to be used.

Miscellaneous

ctrl.deepsleep(ms)

This will disconnect the current connection before going to deepsleep. See machine.deepsleep() for more details.

ctrl.print_cfg_msg()

This prints the configuration status message on the REPL.

ctrl.message_queue_len()

Returns the length of the message queue

ctrl.get_network_type()

Returns the network type currently in use

ctrl.debug(new_level, [update_nvs=True])

Sets the debug level at new_level [0-65565] update_nvs will preserve the setting after reset

ctrl.ztp([new_status=None])

If no parameter is given, will return if ztp is currently enabled. If new_status is True, will enable ztp during next boot If new_status is False, will disable ztp during next boot and stop the current ztp activation process if running

---

Examples

Example 1:

Assuming your device has been activated with one of the provisioning tools available, the following code will send data regularly to the CTRL cloud:

```
# Import what is necessary to create a thread
import time
import math

# Send data continuously to CTRL
while True:
    for i in range(0,20):
        ctrl.send_field(1, math.sin(i/10*math.pi))
        print('sent field {}'.format(i))
        time.sleep(10)
```

Optionally, you can send a timestamp:

```
# Import what is necessary to create a thread
import time
import math

# Send data continuously to CTRL
while True:
    for i in range(0,20):
        ctrl.send_field(1, math.sin(i/10*math.pi), time.time())
        print('sent signal {}'.format(i))
        time.sleep(10)
```

Deprecated API

ctrl.send_signal(signal_number, value)

Deprecated: ``send_signal`` has been removed. Use ``send_field`` instead.

9.1.2 Logging Library

Contents

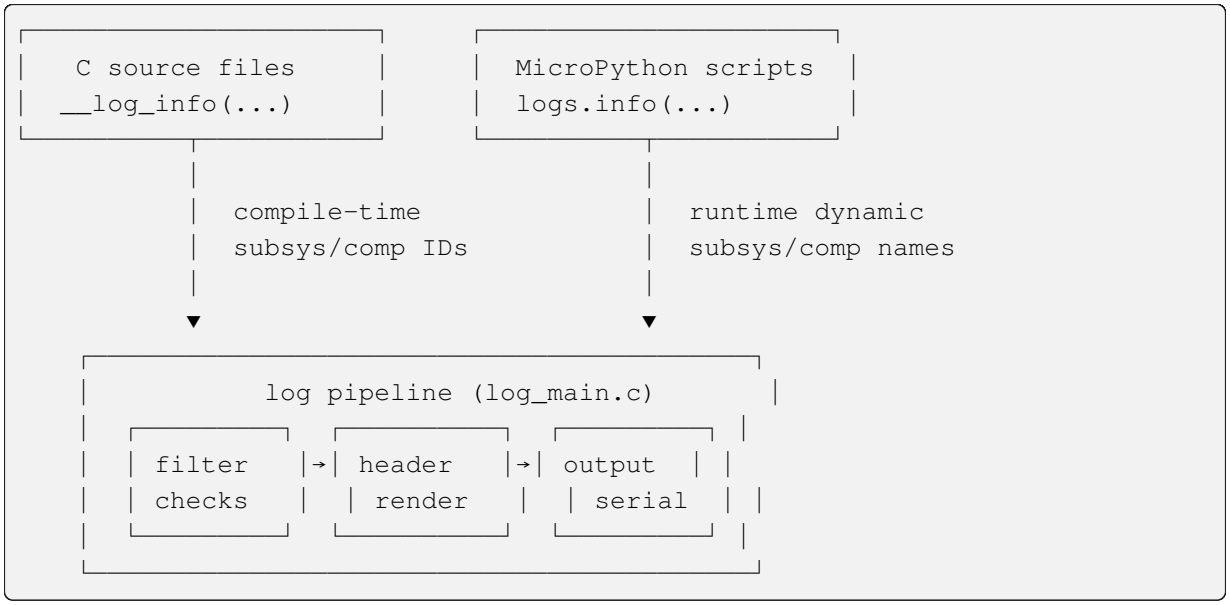
Introduction

The logging library provides a unified, structured logging system for both C native code and MicroPython applications. All log messages pass through the same pipeline and appear in a consistent columnar format with timestamp, log type, OS context, source location, subsystem, component, and message fields.

Key features:

- **Structured output** — fixed-width columns with configurable widths
- **Compile-time definitions** — subsystems and components defined via macros, enabling per-module compile-time gating
- **Runtime filtering** — enable/disable output by subsystem, component, log type, or header column at runtime
- **Terminal coloring** — ANSI color support with per-subsystem and per-component colors
- **Dynamic registration** — MicroPython code can register subsystems and components at runtime and produce log output identical to native C logs
- **Soft-reset safe** — dynamic registrations are cleared on MicroPython soft reset, allowing clean re-registration

Architecture



The static registry holds subsystems and components defined at compile time using `__log_subsystem_def` and `__log_component_def`. The dynamic registry holds subsystems and components registered at runtime from MicroPython via `log_register_subsystem()` and `log_register_component()`. Both registries participate in the same filtering and output pipeline.

C API — Subsystem and Component Definitions

Each C source file selects its subsystem and component before including the log header:

```
#define __log_subsystem     mysubsys
#define __log_component     mycomp
#include "log_lib.h"
```

Subsystems and components are defined (typically once per compilation unit) using the following macros:

- `__log_subsystem_def(name, color, compile_flag, on)` — define a subsystem.

Parameter	Description
`name`	Subsystem identifier (used as a C token)
`colc`	**Color identifier:** **`default`, `red`, `green`, `yellow`,** `blue`, `purple`, `cyan`, `white`, `black`
`compile`	`1` to compile log calls, `0` to strip them at compile time
`on`	`1` to enable output at startup, `0` to start disabled

- `__log_component_def(subsystem, name, color, compile_flag, on)` — define a component under an existing subsystem. Same parameters as above, with an additional `subsystem` parameter naming the parent.

```
// mydriver.c
#define __log_subsystem     drivers
#define __log_component     spi
#include "log_lib.h"

__log_subsystem_def(drivers, blue, 1, 1)
__log_component_def(drivers, spi, yellow, 1, 1)
```

C API — Logging Macros

All macros use `printf`-style format strings. Output is gated by the compile flag and runtime filter state of the current subsystem and component.

Macro	Log Type	Description
`__log_info(fmt, ...)`	info	General status and progress
`__log_debug(fmt, ...)`	debug	Detailed tracing information
`__log_warn(fmt, ...)`	warn	Recoverable issues, unusual conditions
`__log_error(fmt, ...)`	error	Error conditions and failures
`__log_output(fmt, ...)`	output	Direct standard output (no header)
`__log_printf(fmt, ...)`	printf	Continuation of a previous log line
`__log_assert(cond, fmt, ...)`	assert	Fatal assertion (triggers crash if `cond` is false)
`__log_ptr(ptr)`	debug	Convenience: logs `"pointer: name = 0xADDR"`

```
__log_info("SPI transfer complete, %d bytes", len);
__log_debug("Register 0x%02X = 0x%02X", reg, val);
__log_warn("Retry %d/%d", attempt, max_retries);
__log_error("SPI timeout after %d ms", timeout_ms);
__log_assert(buf != NULL, "buffer must not be NULL");
```

C API — Color Macros

Inline color escapes can be embedded in log format strings. These are stripped when `SDK_LOG_LIB_TERMINAL_COLORING_ENABLE` is disabled.

Macro	Color
`__red__`	Red
`__green__`	Green
`__yellow__`	Yellow
`__blue__`	Blue
`__purple__`	Purple
`__cyan__`	Cyan
`__white__`	White
`__black__`	Black
`__default__`	Terminal default

```
__log_info("Status: "__green__"OK"__default__);
__log_error("Status: "__red__"FAIL"__default__" (code %d)", err);
```

String wrapper macros are also available: `__red_str("text")`, `__green_str("text")`, etc. These automatically reset to the default color after the string.

C API — Filtering Functions

All filtering functions work on both static (compile-time) and dynamic (runtime) registrations.

- `void log_filter_subsystem(name, state, silent)` — enable (`true`) or disable (`false`) all output from a subsystem. When `silent` is `false`, a confirmation message is logged.
- `void log_filter_component(subsys_name, comp_name, state, silent)` — enable or disable a specific component.
- `void log_filter_type(type_name, state, silent)` — enable or disable a log type globally (e.g., `"debug"`, `"info"`).
- `void log_filter_header(column_name, state, silent)` — show or hide a header column (e.g., `"timestamp"`, `"func_name"`).
- `void log_filter_header_reorder(name, new_order)` — change the display order of a header column.
- `void log_filter_list_stats()` — print the current filter state for all subsystems, components, log types, and header columns.
- `bool log_filter_subsystem_get_state(name)` — query the current enable state of a subsystem.
- `bool log_filter_component_get_state(subsys_name, comp_name)` — query the current enable state of a component.
- `void log_filter_save_state(p_state, new_state)` — save the current filter state for a subsystem/component pair and set a new state. Useful for temporarily enabling output around a specific operation.
- `void log_filter_restore_state(p_state)` — restore a previously saved filter state.

C API — Dynamic Registration

These functions are the C-level API backing the MicroPython `logs` module. They can also be called directly from C code that needs runtime registration.

- `int log_register_subsystem(name, color_name, enabled, silent)` — register a dynamic subsystem. Returns `0` on success, `-1` if the name is a duplicate, clashes with a static subsystem, or the registry is full.
- `int log_register_component(subsys_name, comp_name, color_name, enabled, silent)` — register a dynamic component under an existing dynamic subsystem. Returns `0` on success, `-1` on failure.
- `void log_dynamic_message(subsys_name, comp_name, p_type_info, msg)` — push a message through the log pipeline using dynamic subsystem/component names.
- `void log_dynamic_registry_clear()` — clear all dynamic registrations. Called automatically on MicroPython soft reset to allow re-registration.

MicroPython API — `logs` Module

```
import logs
```

The `logs` module is a built-in C module enabled by `SDK_LOG_LIB_MPY_CMOD_ENABLE`. It provides Python access to log registration, message output, and filtering.

MicroPython API — Registration

Before logging, at least one subsystem and one component must be registered.

- `logs.register_subsystem(name, *, color='default', enabled=True, silent=True)`

  Register a new dynamic subsystem.

Parameter	Type	Default	Description
`name`	str	*(required)*	Subsystem name (max 23 characters)
`color`	str	`'default'`	Display color for the subsystem column
`enabled`	bool	`True`	Whether the subsystem starts enabled
`silent`	bool	`True`	If `False`, logs a registration confirmation

  Raises `ValueError` if the registry is full or the name is a duplicate.

- `logs.register_component(subsystem, component, *, color='default', enabled=True, silent=True)`

  Register a dynamic component under an existing subsystem.

Parameter	Type	Default	Description
`subsystem`	str	*(required)*	Parent subsystem name (must exist)
`component`	str	*(required)*	Component name (max 23 characters)
`color`	str	`'default'`	Display color for the component column
`enabled`	bool	`True`	Whether the component starts enabled
`silent`	bool	`True`	If `False`, logs a registration confirmation

  Raises `ValueError` if the subsystem is not found or the component is a duplicate.

MicroPython API — Logging Functions

All logging functions share the same signature:

```
logs.<level>(subsystem, component, message)
```

Function	Log Type	Typical Use
`logs.info(subsystem, component, message)`	info	General status and progress
`logs.debug(subsystem, component, message)`	debug	Detailed tracing information
`logs.warn(subsystem, component, message)`	warn	Recoverable issues
`logs.error(subsystem, component, message)`	error	Error conditions and failures
`logs.output(subsystem, component, message)`	output	Direct output (no log-type decoration)

All three arguments are required strings. The subsystem and component must have been previously registered.

```
logs.info("myapp", "network", "Connected to broker")
logs.debug("myapp", "sensors", "Raw value: {}".format(val))
logs.warn("myapp", "network", "Connection timeout, retrying...")
logs.error("myapp", "network", "Failed: {}".format(err))
logs.output("myapp", "sensors", "Temperature: 24.5°C")
```

MicroPython API — Filtering

- `logs.filter_subsystem(subsystem, state, *, silent=False)` — enable or disable a subsystem. Works on both static and dynamic subsystems.
- `logs.filter_component(subsystem, component, state, *, silent=False)` — enable or disable a specific component.
- `logs.filter_header(header_item, state, *, silent=False)` — show or hide a header column. Valid names: `"timestamp"`, `"log_type"`, `"os_info"`, `"subsystem"`, `"component"`, `"filename"`, `"line_num"`, `"func_name"`.
- `logs.filter_log_type(log_type, state, *, silent=False)` — enable or disable a log type globally. Valid names: `"info"`, `"debug"`, `"warn"`, `"error"`, `"output"`, `"printf"`, `"assert"`.
- `logs.header_reorder(header_item, new_order)` — change the display position of a header column.
- `logs.filter_stats()` — print the current filter state for all subsystems, components, types, and header columns.

```
# Disable all debug output
logs.filter_log_type("debug", False)

# Hide the function name column
logs.filter_header("func_name", False)

# Disable a specific component
```

(continues on next page)

(continued from previous page)

```
logs.filter_component("ctrl", "sensors", False)

# Print full filter status
logs.filter_stats()
```

Log Output Format

All log messages (C and MicroPython) appear in the same structured format:

```
|000:00:06-822|  info  |1:mp_task      |mpy             |  ctrl   | ␣
↪connection | Connected to MQTT broker
```

The default column layout:

Column	Header Name	Default Width	Description
Timestamp	`timestamp`	13	Time since boot `HHH:MM:SS-mmm`
Log type	`log_type`	8	`info`, `debug`, `warn`, `error`, `output`
OS context	`os_info`	12	Core ID and task name (e.g. `1:mp_task`)
Function	`func_name`	15	Source function name (C) or `mpy` (Python)
Subsystem	`subsystem`	9	Registered subsystem name
Component	`component`	13	Registered component name
Message	—	—	The log message text

Columns can be enabled/disabled, reordered, and resized via Kconfig and runtime filter APIs. The `filename` and `line_num` columns are disabled by default.

Kconfig Options

All options are under the `SDK_LOG_LIB_ENABLE` menu.

Option	Default	Description
`SDK_LOG_LIB_ENABLE`	y	Master enable for the logging library
`SDK_LOG_LIB_MEMORY_BUFFER_SIZE_KB`	4	Internal buffer size in KB
`SDK_LOG_LIB_TERMINAL_COLORING_ENABLE`	y	ANSI color output
`SDK_LOG_LIB_MPY_CMOD_ENABLE`	y	Enable MicroPython `logs` module

Log types — each can be individually compiled in or out:

Option	Default
`SDK_LOG_LIB_TYPE_INFO`	y
`SDK_LOG_LIB_TYPE_DEBUG`	y
`SDK_LOG_LIB_TYPE_WARN`	y
`SDK_LOG_LIB_TYPE_ERROR`	y
`SDK_LOG_LIB_TYPE_PRINTF`	y
`SDK_LOG_LIB_TYPE_ASSERT`	y
`SDK_LOG_LIB_TYPE_MEM_DUMP`	y
`SDK_LOG_LIB_TYPE_ENFORCE`	y

Header fields — each can be enabled/disabled and has a configurable width:

Field	Enable Option	Width Option	Default Width
Timestamp	`SDK_LOG_LIB_HEADER_TIMESTAMP` (y)	`..._TIMESTAMP_WIDTH`	13
Log type	`SDK_LOG_LIB_HEADER_LOG_TYPE` (y)	`..._LOG_TYPE_WIDTH`	8
OS context	`SDK_LOG_LIB_HEADER_OS_CONTEXT_IN` (y)	`..._OS_CONTEXT_INFO_WIDTH`	12
File name	`SDK_LOG_LIB_HEADER_FILENAME` (n)	`..._FILENAME_WIDTH`	12
Line number	`SDK_LOG_LIB_HEADER_LINE_NUM` (n)	`..._LINE_NUM_WIDTH`	6
Function	`SDK_LOG_LIB_HEADER_FUNC_NAME` (y)	`..._FUNC_NAME_WIDTH`	15
Subsystem	`SDK_LOG_LIB_HEADER_SUBSYSTEM` (y)	`..._SUBSYSTEM_WIDTH`	9
Component	`SDK_LOG_LIB_HEADER_COMPONENT` (y)	`..._COMPONENT_WIDTH`	13

Line wrapping is optionally enabled with `SDK_LOG_LIB_TOTAL_LINE_WRAPPING_ENABLE` (default: off) and `SDK_LOG_LIB_TOTAL_LINE_WIDTH` (default: 86).

Capacity Limits

Dynamic registry (runtime registrations from MicroPython or C):

Resource	Limit
Dynamic subsystems	8
Dynamic components (shared across all subsystems)	32
Name length	23 characters

These limits are separate from compile-time static subsystems and components, which have no practical limit beyond available flash.

The dynamic registry is cleared on every MicroPython soft reset (`log_dynamic_registry_clear()`), allowing modules to re-register cleanly.

Example — C Component

```
// spi_driver.c
#define __log_subsystem     drivers
#define __log_component     spi
#include "log_lib.h"

__log_subsystem_def(drivers, blue, 1, 1)
__log_component_def(drivers, spi, yellow, 1, 1)

esp_err_t spi_transfer(uint8_t* buf, size_t len) {
    __log_debug("SPI transfer: %d bytes at %p", len, buf);

    esp_err_t err = /* ... actual transfer ... */;
```

(continues on next page)

(continued from previous page)

```
    if (err != ESP_OK) {
        __log_error("SPI transfer failed: %s", esp_err_to_name(err));
        return err;
    }

    __log_info("SPI transfer complete: %d bytes", len);
    return ESP_OK;
}
```

Output:

```
|000:00:01-234|  debug |0:spi_task    |spi_driver.c    | drivers |     spi ␣
↪   | SPI transfer: 64 bytes at 0x3FC90000
|000:00:01-235|  info  |0:spi_task    |spi_driver.c    | drivers |     spi ␣
↪   | SPI transfer complete: 64 bytes
```

Example — MicroPython

```
import logs

# Register subsystem and components
logs.register_subsystem("myapp", color='cyan', enabled=True, silent=False)
logs.register_component("myapp", "main", color='green', enabled=True)
logs.register_component("myapp", "network", color='blue', enabled=True)
logs.register_component("myapp", "sensors", color='yellow', enabled=True)

# Log messages
logs.info("myapp", "main", "Application starting")
logs.info("myapp", "network", "Connecting to WiFi...")
logs.debug("myapp", "network", "SSID: MyNetwork, RSSI: -45")
logs.warn("myapp", "sensors", "Battery level low: 15%")
logs.error("myapp", "network", "MQTT connection lost")

# Runtime filtering
logs.filter_component("myapp", "sensors", False)     # silence sensors
logs.filter_component("myapp", "sensors", True)      # re-enable sensors
logs.filter_stats()                                   # show all filter␣
↪states
```

Output:

```
|000:00:06-100|  info  |1:mp_task     |mpy             |  myapp  |    main ␣
↪   | Application starting
|000:00:06-110|  info  |1:mp_task     |mpy             |  myapp  |  ␣
↪network   | Connecting to WiFi...
|000:00:06-120|  debug |1:mp_task     |mpy             |  myapp  |  ␣
↪network   | SSID: MyNetwork, RSSI: -45
|000:00:06-130|  warn  |1:mp_task     |mpy             |  myapp  |  ␣
```

(continues on next page)

(continued from previous page)

```
↪sensors    | Battery level low: 15%
|000:00:06-140|  error |1:mp_task      |mpy              |  myapp   |  ␣
↪network    | MQTT connection lost
```

9.2 Network Interfaces

9.2.1 LoRa API Documentation

Contents

- Initialization
- LoRa Modes
- LoRa Test Stub
- LoRa Events and Callback
- LoRa RAW APIs
- LoRa WAN APIs
- LoRa Certification Mode
- Examples
  - Example - LoRa-WAN End-Device Commissioning:
    - OTAA - LoRaWAN Specs v1.0.x
    - OTAA - LoRaWAN Specs v1.1.x
    - ABP - LoRaWAN Specs v1.0.x
    - ABP - LoRaWAN Specs v1.1.x

Initialization

To initialize the LoRa stack, you need to do `import lora` to be able to call any lora utility and to automatically initialize the saved operating lora mode:

```
import lora      # mandatory before any lora function call
                 # automatically initialize lora for current operating mode

lora.deinit()    # deinit the stack
                 # all lora calls will be ignored after it

lora.initialize()
                 # initialize the stack again and back to normal operation
```

LoRa Modes

There are two available modes for lora; *LoRa-RAW* and *LoRa-WAN*.

```
lora.mode()                                 # returns the current operating␣
↪LoRa mode
lora.mode(lora._mode.RAW)                  # switch mode to LoRa-RAW
lora.mode(lora._mode.WAN)                  # switch mode to LoRa-WAN (ADR␣
↪enabled by default)
lora.mode(lora._mode.WAN, adr=False)       # switch to LoRa-WAN with ADR␣
↪disabled
lora.mode(lora._mode.WAN, adr=True)        # switch to LoRa-WAN with ADR␣
↪explicitly enabled
```

The optional `adr` keyword argument controls Adaptive Data Rate (ADR) when switching to WAN mode. ADR allows the network server to optimise the device's data rate and transmission power. Disable it when the device is mobile or the RF environment is expected to change frequently.

> **NOTE**: The `adr` argument is only meaningful when switching to `lora._mode.WAN`. It has no effect when querying the current mode or switching to `lora._mode.RAW`.

LoRa Test stub

In case of testing and no need to connect a user level callback, lora-stack provides an internal stub for callbacks to be used while testing.

```
lora.callback_stub_connect()     # connect the internal lora-stack callback␣
↪stub
lora.callback_stub_disconnect() # to disconnect it and connect user␣
↪provided one
```

9.2.2 LoRa WAN API Documentation

Available LoRa WAN APIs Summary

API Call	Brief description
`lora.stats()`	displays the current stats of lora WAN
`lora.wan_params()`	set the lora WAN regional parameters
`lora.commission()`	set the LoRa-WAN commissioning parameters
`lora.join()`	start performing join procedure
`lora.send()`	transmit a LoRa-WAN packet
`lora.recv()`	receive a LoRa-WAN packet
`lora.port_open()`	open a lora-wan port to be able to tx/rx over it
`lora.port_close()`	close a lora-wan port, tx/rx on it will be discarded
`lora.callback()`	set a user level callback to listen to specifc events
`lora.duty_get()`	get the current duty-cycle in milliseconds
`lora.duty_set()`	set the the duty-cycle to a specific value
`lora.duty_start()`	start duty-cycle operation
`lora.duty_stop()`	stop duty-cycle operation
`lora.enable_rx_listening()`	perform class-a cycle to fetch pending DL msg
`lora.disable_rx_listening()`	if no pending UL msg, discard class-a cycle
`lora.mode(adr=)`	enable or disable Adaptive Data Rate (ADR)
`lora.tx_airtime()`	get last TX time-on-air in milliseconds
`lora.last_rx_at()`	get timestamp (ms since boot) of last network reception

LoRa WAN Stats

Displays useful information about the current lora-WAN settings such as: enabled `region`, current working `class`, Device EUI `DevEUI`, Join EUI, current assigned `DevAddr` after the joined procedure, lora-wan operating `version` and the type of `activation` whether `NONE`, `OTAA`, or `ABP`

Example:

```
lora.stats()
    # outputs
    # - region               : EU-868
    # - class                : class-A
    # - dev eui              : 39 2C 39 D1 5D 3E 12 10
    # - join eui             : EA 68 DE 1C 4B E0 20 F4
    # - dev addr             : 01 88 DB B8
    # - lorawan version      : val: 16778240 ( 1.0.4.0 )
    # - activation           : OTAA
```

Setting LoraWAN parameters

To change the current operating `region` and forces the device to be in `class` `A`, `B` or `C`

Example:

```
lora.wan_params(region = lora._region.REGION_EU868, lwclass = lora._class.
↪CLASS_A)
    # sets the lora region to EU868
    # sets the default working class to class A which is the default
```

Device commissioning

Commissioning a device means preparing a new lora-wan end-device, hence if the LoRa-Stack is prepared with previous end-device credentials, it will be cleared and will be configured with the new credentials as if it is a completely new end-device. In other words, the LoRa MAC layer state will start clean for a new end-device session and previous session will be cleared.

If the provided credentials are same as previousely commissioned parameters, the commissioning will be ignored.

the end-device commissioning credentials are as follows:

- `version=<version>` to specify the end-device LoRa standard. It takes one of the following:
  - `version=lora._version.VERSION_1_0_x` LoRa version 1.0.x
  - `version=lora._version.VERSION_1_1_x` LoRa version 1.1.x
- `type=lora._commission.OTAA` Device will be commissioned using OTAA procedure and the device shall perform the Join procedure before tx/rx with the network in this activation method, the following keys shall be provided along with:
  - `DevEUI` The device EUI
  - `JoinEUI` The Join EUI
  - `AppKey` The AppKey
  - `NwkKey` The NwkKey if version 1.1.x
- `type=lora._commission.ABP` The device will not perform the join procedure and it will send directly an UL message. The following parameters shall be provided:
  - `DevEUI` The device EUI
  - `DevAddr` The device network address
  - `AppSKey` The application security key
  - `NwkSKey` The network security key
- `verify=True` To check the provided parameters are same as the current commissioned parameters or not without doing any commissioning processing.

> **Note**
>
> If the device is already joined the network, after this commissioning operation the device will not be considered joined and need to rejoin again using the new provided commissioning parameters.

Example

```
import lora
import ubinascii

# verify the existing commissioning
if lora.commission(
    verify  = True,
    type    = lora._commission.OTAA,
```

(continues on next page)

(continued from previous page)

```
    version = lora._version.VERSION_1_0_X,
    DevEUI  = ubinascii.unhexlify('0000000000000000'),
    JoinEUI = ubinascii.unhexlify('0000000000000000'),
    AppKey  = ubinascii.unhexlify('00000000000000000000000000000000')
    ) == True:
    print('end-device is already commissioned')
else:
    print('end-device is not commissioned')

# OTAA Version 1.0.x Example
lora.commission(
    type    = lora._commission.OTAA,
    version = lora._version.VERSION_1_0_X,
    DevEUI  = ubinascii.unhexlify('0000000000000000'),
    JoinEUI = ubinascii.unhexlify('0000000000000000'),
    AppKey  = ubinascii.unhexlify('00000000000000000000000000000000')
    )

# OTAA Version 1.1.x Example
lora.commission(
    type    = lora._commission.OTAA,
    version = lora._version.VERSION_1_1_X,
    DevEUI  = ubinascii.unhexlify('0000000000000000'),
    JoinEUI = ubinascii.unhexlify('0000000000000000'),
    AppKey  = ubinascii.unhexlify('00000000000000000000000000000000'),
    NwkKey  = ubinascii.unhexlify('00000000000000000000000000000000')
    )

# ABP Version 1.0.x Example
lora.commission(
    type    = lora._commission.ABP,
    version = lora._version.VERSION_1_0_X,
    DevAddr = 0x00000000,
    DevEUI  = ubinascii.unhexlify('0000000000000000'),
    AppSKey = ubinascii.unhexlify('00000000000000000000000000000000'),
    NwkSKey = ubinascii.unhexlify('00000000000000000000000000000000')
    )

# ABP Version 1.1.x Example
lora.commission(
    type    = lora._commission.ABP,
    version = lora._version.VERSION_1_1_X,
    DevAddr = 0x00000000,
    DevEUI  = ubinascii.unhexlify('0000000000000000'),
    AppSKey = ubinascii.unhexlify('00000000000000000000000000000000'),
    NwkSKey = ubinascii.unhexlify('00000000000000000000000000000000')
    )
```

Join

Mandatory operation to let the device join the network and be able to TX/RX with the lora-WAN server. In case of ABP activation, the end-device is considered joined after commissioning and join here will not have any effect.

Example:

```
import lora
import time

# start join procedure
lora.join()

# wait until join
while lora.is_joined() == False:
    time.sleep(2)
    pass
```

Sending data

The successfully joined device is capable to tx/rx with the LoRaWAN server. To start tx/rx operation, the user shall open a port first using the `lora.port_open()` first, otherwise, no tx/rx operation will be performed.

To plan an UL message. It takes the following parameters:

- `message` the message buffer to be sent, can be a normal string or byte array
- optional arguments:

parameter-name	value-type	default-value	desc
`confirm`	bool	False	To receive an ack from network server upon its reception
`port`	int	1	on which lora-wan port to send this message
`retries`	int	0	number of retried until the UL tx succeeded
`timeout`	int	`no-timeout`	time-out in ms to perform the full UL operation
`sync`	int	False	block until timeout or operation success/failure
`id`	int	0	user defined message id to be returned in the callback

Example:

```
# send an asynchronous UL message, with id=0, and without confirmation, no
# retries upon tx failure, and no specified timeout which means the message␣
↪will
# be scheduled for UL in its turn within the pending UL messages until the
# duty-cycle tx operation fetches it and send it.
lora.send('ul tx message')

# send a message like before message, but if timeout of 3 seconds passed,
# drop the message and don't send it
```

(continues on next page)

(continued from previous page)

```
lora.send('ul tx message', timeout=3000)

# repeat the transmission for upto 2 times, the full operation timeout␣
↪including
# the retries attempts is 20 seconds
lora.send('ul tx message', timeout=20000, retries=2)

# same as before message but wait for confirmation as well from the network
# server
lora.send('ul tx message', timeout=20000, retries=2, confirm=True)

# same as the previous message, but the caller will be blocked until␣
↪timeout,
# or message is successfully sent and acked
lora.send('ul tx message', timeout=20000, retries=2, confirm=True,␣
↪sync=True)
```

Receiving Data

The received data will come in the callback only

LoRa Ports

LoRa WAN sends/receives data over what is called `ports`, valid application ports are from `1` to `223`.

A port must be opened first before sending and receiving data.

Example:

```
# opening port 1
lora.port_open(1)   # data can be tx/rx over port 1

lora.send('data', port=5)    # ignored because port 5 is not opened
lora.send('data', port=1)    # will be planned successfully for UL

# opening port 1
lora.port_open(5)   # data can be tx/rx over port 5
lora.send('data', port=5)    # now it will be planned successfully

lora.port_close(1)  # no tx/rx more over this port
lora.port_close(5)  # no tx/rx more over this port
```

Callbacks `lora.callback()`

It can set a user lever callback and it takes the following parameters:

- 'handler' a callbeack function to be called.
- 'trigger' an OR combination of the required events that can trigger to this callback.
- 'port' a special port of the incoming messages events (default `any`)

Example:

```
def lora_callback(context):
    def get_class_const_name(__class, __const):
        for k,v in __class.__dict__.items():
            if v == __const:
                return k
        return 'unknown'
    print('lora event: {} with-context: {}'.format(
        get_class_const_name(lora._event, context['event']), context))
    pass

lora.callback( handler = lora_callback )
```

NOTE: Refer to the comprehensive documentation on the LoRa-Callback system for more details here.

Duty cycle operations

The device is normally working in class-A and the device shall follow a duty-cycle to perform an UL/DL operation. This duty cycle should be regulated to respect the time-on-air for this device.

The available operation are `duty_set()`, `duty_get()`, `duty_start()`, `duty_stop()`

Example

```
lora.duty_set(15000)     # sets the duty cycle timer to 15 seconds
                         # which means that every 15 seconds the device will␣
↪check
                         # if TX pending and send it, and listen in the RX␣
↪window
                         # to any scheduled DL message for this device

lora.duty_get()          # retrieve the current duty cycle time

lora.duty_start()        # start duty cycle operation

lora.duty_stop()         # stop duty cycle operation
```

RX listening

RX listening means that the device will send a dummy UL message in case no pending TX message is pending, so that the server will plan an RX window for this device and hence the device can receive any pending DL message.

the default behaviour is that the RX listening is **disabled**

Example

```
lora.enable_rx_listening()       # enable listening

lora.disable_rx_listening()      # disable listening
                                 # the device will listen only when there is␣
↪a
                                 # real planned UL TX message.
```

Adaptive Data Rate (ADR)

Adaptive Data Rate allows the LoRaWAN network server to optimise each device's data rate and transmission power based on the observed link quality. ADR is **enabled by default** when switching to WAN mode.

Disable ADR when the device is mobile or the RF environment is expected to vary frequently, so the network server does not lock the device to a data rate that may become unsuitable.

The `adr` keyword argument is accepted by `lora.mode()` when switching to WAN mode, and can also be changed at any time while the stack is running.

Example:

```
import lora

# switch to WAN mode with ADR disabled from the start
lora.mode(lora._mode.WAN, adr=False)

# re-enable ADR later (e.g. once the device is stationary)
lora.mode(lora._mode.WAN, adr=True)
```

TX Airtime — `lora.tx_airtime()`

Returns the time-on-air (in milliseconds) of the most recently transmitted LoRaWAN packet. The value is updated immediately after each successful transmission, before the RX windows open.

The returned value is `0` until the first packet has been sent in the current session.

This is useful for duty-cycle management: by knowing the exact airtime of the last frame you can compute the minimum off-time required by regional regulations before the next transmission.

Example:

```
import lora
import time

lora.send('hello')

# after the send callback fires:
airtime_ms = lora.tx_airtime()
print('last TX airtime: {} ms'.format(airtime_ms))

# simple 1%-duty-cycle guard (EU868 default sub-band)
min_off_time_ms = airtime_ms * 99
time.sleep_ms(min_off_time_ms)
```

Last Network RX Timestamp — `lora.last_rx_at()`

Returns the value of the monotonic millisecond timer (`utime.ticks_ms()` compatible) at the moment the most recent downlink frame was received from the network.

The timestamp is updated on:

- Any application-layer downlink (port 1–223)

- Any MAC-only downlink (port 0 / network commands)
- An uplink ACK (`AckReceived`) returned by the network server in response to a confirmed uplink

The returned value is `0` until the first downlink (or ACK) has been received in the current session.

This is useful for implementing a network-connectivity watchdog: if `utime.ticks_diff(utime.ticks_ms(), lora.last_rx_at())` exceeds a threshold, the device can decide to re-join or perform a reset.

Example:

```
import lora
import utime

WATCHDOG_TIMEOUT_MS = 10 * 60 * 1000   # 10 minutes without any network␣
↪contact

def check_network_health():
    last_rx = lora.last_rx_at()
    if last_rx == 0:
        print('no downlink received yet')
        return False
    elapsed = utime.ticks_diff(utime.ticks_ms(), last_rx)
    if elapsed > WATCHDOG_TIMEOUT_MS:
        print('no network contact for {} ms — triggering rejoin'.
↪format(elapsed))
        return False
    return True
```

9.2.3 LoRa RAW API Documentation

Available LoRa RAW APIs Summary

API Call	Brief description
`lora.stats()`	displays the current stats of lora RAW
`lora.radio_params()`	set one or more radio parameter
`lora.callback()`	set a user level callback
`lora.send()`	transmit a given data over LoRa
`lora.recv()`	open a timed-out rx window to listen to any incoming data
`lora.recv_cont_start()`	switch to continuous rx mode
`lora.recv_cont_stop()`	close the continuous rx mode
`lora.tx_continuous_wave_start()`	start tx continuous wave operation
`lora.tx_continuous_wave_stop()`	stops tx continuous wave operation

LoRa Raw Settings

To display the current settings of the lora RAW, use `lora.stats()`, Then you will experience something like this:

```
>>> lora.stats()
    regional params
```

(continues on next page)

(continued from previous page)

```
        region          : EU868
        frequency       : 868000000 Hz
        freq_khz        : 868000.000 KHz
        freq_mhz        : 868.000 MHz
    modulation params
        sf              : 12
        bandwidth       : 125 KHz
        coding_rate     : 4_7
    packet params
        preamble        : 8
        payload         : 51
        crc_on          : False
    lora tranceiver
        chip            : SX1262
        max tx_power    : +22 dBm
    tx params
        tx_power        : +10 dBm
        antenna_gain    : +1.00 dBi
        tx_power_eff    : +9 dBm
        tx_timeout      : 6000 msec
        tx_iq           : False
    rx params
        rx_timeout      : 6000 msec
        rx_iq           : False
```

Here is the meaning of each displayed parameter:

- ``**regional params**``: The parameters corresponding to the current region
  - `region`: The region in which the device will operate.
  - `frequency`, `freq_khz` or `freq_mhz`: The required frequency in **Hz**, **KHz** or **MHz** respectively.
- ``**modulation params**``: The current modulation parameters which are; spreading-factor `sf`, `bandwidth` and `coding_rate`
- ``**packet params**``: parameters related to the packet data constraints such as `preamble` length, current maximum `payload` size, and if the `crc_on` is applied to the payload or not
- ``**lora tranceiver**``: shows the current info about the current used tranceiver such as the `chip` used and the maximum `tx_power` it can produce.
- ``**tx params**``: the current tx settings;
  - the current desired `tx_power` including the antenna gain
  - the `antenna_gain`; should be set according to the current HW prescribed antenna gain to be taken into consideration while determining the chip output tx power
  - the `tx_power_eff` which is the actual effective chip output power after subtracting the antenna gain from the desired `tx_power`
  - `tx_timeout` the time-out of sending a message; it should be sufficient enough according to the time on air required for the current modulation parameters.

  - `tx_iq` indicates whether inverted IQ polarity feature is enabled or not

- ``**rx params**``: the current rx settings;
  - `rx_timeout` the rx window time in non continuous reception
  - `rx_iq` indicates whether inverted IQ polarity feature is enabled or not

To reset all parameters to the region defaults, provide `reset_all` flag like:

```
>>> lora.radio_params(reset_all=True)
```

Modifying Radio Parameters

To change any radio parameter, use `lora.radio_params()` which takes its parameters as in the following BNF formatted description:

```
<radio-param-change-call> ::=
        "lora.radio_params(" <param-value-pair>  <params-list> ")"

<params-list> ::=
        "," <param-value-pair>
        | <params-list>
        | ""

<param-value-pair> ::=
    reset_all    "=" <bool-value>            ; reset to factory settings
    | region      "=" <region-value>           ; change the region
    | frequency   "=" <freq-value-in-hz>       ; desired freq in Hz
    | freq_khz    "=" <freq-value-in-khz>      ; desired freq in KHz
    | freq_mhz    "=" <freq-value-in-mhz>      ; desired freq in MHz
    | tx_power    "=" <signed-int-value>       ; desired tx power
    | sf          "=" <sf-value>               ; spreading factor
    | coding_rate "=" <cr-value>               ; coding rate
    | preamble    "=" <integer-value>          ; preamble length
    | bandwidth   "=" <bw-value>               ; band-width
    | tx_iq       "=" <bool-value>             ; inverted IQ feature

<bool-value> ::= "True" | "False"

<region-value> ::= "lora._region." <region>
<region> ::=
    "REGION_AS923" | "REGION_AU915" | "REGION_CN470" | "REGION_CN779"
    | "REGION_EU433" | "REGION_EU868" | "REGION_IN865" | "REGION_KR920"
    | "REGION_RU864" | "REGION_US915"

<freq-value-in-hz> ::= <positive-integer-value>
<freq-value-in-khz> ::= <floating-point-value>
<freq-value-in-mhz> ::= <floating-point-value>

<sf-value> ::= "7" | "8" | "9" | "10" | "11" | "12"

<bw-value> ::= "lora._bw." <bw>
```

(continues on next page)

(continued from previous page)

```
<bw> ::= "BW_125KHZ" | "BW_250KHZ" | "BW_500KHZ"

<cr-value> ::= "lora._cr." <cr>
<cr> ::= "CODING_4_5" | "CODING_4_6" | "CODING_4_7" | "CODING_4_8"
```

Examples:

```
# setting a new lora region to set the lora constraints to this region
lora.radio_params(region=lora._region.REGION_EU433)

# set frequency to 433.3 MHz
lora.radio_params(freq_mhz=433.3)    # accepted because it is valid region␣
↪freq

# set frequency to 435 MHz
lora.radio_params(freq_mhz=433.3)    # accepted because it is valid region␣
↪freq

lora.radio_params(freq_mhz=435)      # regected because it is non-valid␣
↪region freq
    # error: incompatible frequency 435000000 with the region EU-433

# set the tx power to 5 dBm
lora.radio_params(tx_power=5)
lora.stats()
    #    tx_power       : +5 dBm        --> desired tx-power
    #    antenna_gain   : +2.15 dBi     --> current set antenna-gain
    #    tx_power_eff   : +2 dBm        --> actual effective lora chip␣
↪output power
lora.radio_params(antenna_gain=1)       # the effective power will change␣
↪accordingly
lora.stats()
    #    tx_power       : +5 dBm        --> desired tx-power
    #    antenna_gain   : +1.00 dBi     --> current set antenna-gain
    #    tx_power_eff   : +4 dBm        --> actual effective lora chip␣
↪output power

# setting out of region valid tx power will be regicted for saftey
lora.radio_params(tx_power=45)
    # error: invalid chip power +45 dBm -- chip SX1262 tx power range ( -8 ~
↪ +23 ) dBm considering antenna gain 1.00 dBi
    # error: invalid tx-power 45

# setting spreading factor to 8 and BW to 250 and coding rate to 4/6
lora.radio_params(sf = 8, bandwidth = lora._bw.BW_250KHZ, coding_rate =␣
↪lora._cr.CODING_4_6)
    # modulation params
    #     sf             : 8
    #     bandwidth      : 250
    #     coding_rate    : 4_6
```

(continues on next page)

(continued from previous page)

```
# setting wrong values will be regected and the whole parameters will be␣
↪ignored
lora.radio_params(bandwidth=9, tx_power=44, sf=90) # gived the following␣
↪reported errors
error: invalid chip power +44 dBm -- chip SX1262 tx power range ( -7 ~ +24␣
↪) dBm considering antenna gain 2.15 dBi
error: invalid argument value 'tx_power'
error: invalid argument value 'sf'
error: invalid argument value 'bandwidth'
```

Note: Changing the region, will reset the entire radio parameters to the defaults of this new region

Note: The lora interface provides some class constants for some radio parameters:

- `lora._bw`: contains all supported band width values
- `lora._cr`: contains all supported coding rate values
- `lora._region`: contains all supported regions values

```
# Example
# you can see the allowed values constans, by pressing the class␣
↪names
# followed by double <tab>

>>> lora._bw.       # press <tab> <tab> to see the following list
BW_125KHZ       BW_250KHZ       BW_500KHZ

>>> lora._cr.       # press <tab> <tab> to see the following list
CODING_4_5      CODING_4_6      CODING_4_7      CODING_4_8

>>> lora._region.   # press <tab> <tab> to see the following list
REGION_AS923    REGION_AU915    REGION_CN470    REGION_CN779
REGION_EU433    REGION_EU868    REGION_IN865    REGION_KR920
REGION_RU864    REGION_US915
```

Note: To change the frequency value, it can be done through one of these parameters (`frequency`, `freq_khz` or `freq_mhz`), however it is possible to specify one or more of those parameters. Hence in that case, the specified parameters will be considered in a priority fashion. `frequency` parameter has highest consideration priority and `freq_mhz` has lowest consideration priority.

```
# consider the current radio frequency parameter is 868.000 MHz

>>> lora.radio_params(frequency=868000000, freq_mhz=868.3)
# the specified `frequency` parameter will be considered first,␣
↪but because
# it has the same value of the current frequency, it will be␣
↪bypassed,
# then the next specified `freq_mhz` parameter will be considered,␣
```

(continues on next page)

(continued from previous page)

```
↪and
# the radio frequency will be changed accordingly.
# --> hence the current radio frequency parameter becomes 868.300␣
↪MHz

>>> lora.radio_params(freq_mhz=868.3, frequency=868000000)
# the highest priority parameter `frequency` will be considered␣
↪first.
# and because it holds newer value than the current radio␣
↪frequency, the
# radio frequency will be modified accordingly.
# --> hence the current radio frequency parameter becomes 868.000␣
↪MHz
# --> and the next specified `freq_mhz` parameter is neglicted
```

Setting LoRa RAW user Callback

To set a user level callback to listen the RX events, see the following example to know the available events that can come in the callback

Example

```
def lora_callback(context):
    def get_class_const_name(__class, __const):
        for k,v in __class.__dict__.items():
            if v == __const:
                return k
        return 'unknown'
    print('lora event: {} with-context: {}'.format(
        get_class_const_name(lora._event, context['event']), context))
    pass

lora.callback( handler = lora_callback )
```

Send (TX) Data

To send a specific data message, it takes the following parameters:

- `message`: it is the normal data buffer, it could be a normal string or byte array
- `timeout`: it is an optional argument to specify the tx operation deadline the default timeout will be the radio `tx_timeout` parameter
- `sync`: it is an optional argument to perform this operation synchronously or asynchronously (default: `sync=False`)

Examples

```
lora.send("test message")    # sends a message asynchronously and with tx_
↪timeout
                             # as a full tx operation timeout
```

(continues on next page)

(continued from previous page)

```
# send a message asynchronously and the full tx operation shall canceled␣
↪after 1 second
lora.send("test message", timeout=1000)

# send a message synchronously and the full tx operation shall canceled␣
↪after 1 second
# in this case, the caller will be blocked until tx succeeded or timeout is␣
↪over
lora.send("test message", timeout=1000, sync=True)
```

Receive (RX) Data

To receive a data and it takes the following parameters:

- `timeout`: it is an optional argument to specify the tx operation deadline the default timeout will be the radio `rx_timeout` parameter
- `sync`: it is an optional argument to perform this operation synchronously or asynchronously (default: `sync=True`)
  - `sync` the function will return the received message
  - `async` the received message will be returned in the RX event in the callback

Example:

```
lora.recv()   # waits for `rx_timeout` radio parameter or until a data is␣
↪received

# wait for maximum 2 second or until a data is received
lora.recv(timeout=2000)

# place a receive request and return immediately
#   - if a data is received before 3 second timeout, it will be returned in␣
↪the callback
#   - if timeout happened, the rx operation will be canceled and a timeout␣
↪event will be
#     fed back in the callback
lora.recv(timeout=3000, sync=True)
```

RX Continuous Mode

LoRa RAW can operate in continuous reception mode and any received data will be thrown in the registered user callback

Example:

```
lora.recv_cont_start()     # starting the RX continuous mode

lora.recv_cont_stop()      # exiting the RX continuous mode
```

Continuous TX Wave mode

Sets the radio tranciver to continuous transmission mode for testing.

The tx continuous wave mode does not use the normal parameters set by the `lora.radio_params()` method, but instead it uses the following parameters

- `tx_power` the required tx power during the test
- `frequency` the required test frequency in Hz (default: 868 MHz)
- `timeout` an optional timeout in milliseconds (default: 10 seconds)

  Remark: if the system is in sending or receiving operation, the operation will be cancelled and the system will start serving the tx continuous wave test command. After timeout is over, the system will go to its IDLE state.

Example:

```
lora.tx_continuous_wave_start(      # starting the TX continuous wave mode
    tx_power = 20,                  # use tx_power = 20dBm
    frequency = 433000000,          # test frequency 433 MHz
    timeout = 20000)                # timeout for the tx continuous wave 20␣
↪sec

lora.tx_continuous_wave_stop()    # exiting the TX continuous wave mode
```

9.2.4 LoRa Callback System

Contents

- Introduction
- LoRa Events
- LoRa Callback Generic Interface
- Example - LoRa-RAW
- Example - LoRa-WAN
- Remarks

Introduction

LoRa APIs provides very flexible interface to enable the user to connect a special callback routine to listen to emmitted LoRa events.

When LoRa is in its operation whether sending or receiving, it generates an event to let the user knows and listen to its occurred events, so that the user can take a proper action based on his application logic.

LoRa Events

LoRa stack can emmit the events described in the following table:

LoRa Event	Valid Mode	Brief description
`lora._event.EVENT_RX_DONE`	RAW, WAN	occurs when the LoRa stack receives something
`lora._event.EVENT_RX_FAIL`	RAW	occurs if the receiving operation failed
`lora._event.EVENT_RX_TIMEOUT`	RAW	occurs if the receiving operation timed-out
`lora._event.EVENT_TX_DONE`	RAW, WAN	when the requested transmission operation is full-filled
`lora._event.EVENT_TX_CONFIRM`	WAN	when a LoRa-WAN confirmation is received
`lora._event.EVENT_TX_FAILED`	RAW, WAN	requested transmission operation failed
`lora._event.EVENT_TX_TIMEOUT`	RAW, WAN	requested tx operation timedout (deadline)

`lora._event.EVENT_RX_DONE`

This event occurs when the lora stack received something from the air.

- **In LoRa-RAW mode**

  It is generated if the device is set to receive something from the air and successfully received new data. or the device is set in continuous receiving mode and new data received and present to be delivered to the user.

  The event data attached to this event in the callback is a tuble that carry the following key information:

  - `data` a byte array object containing the actual received data.
  - `RSSI` an integer value represnting the RSSI of the received signal.
  - `SNR` an integer value represnting the SNR of the received signal.

- **In LoRa-WAN mode**

  It is generated automatically if a Class-A cycle occurs and a scheduled data from the network side is successfully received by the device.

  It is also generated when the device is working in Class-C and the device receives a message from the network.

  In LoRa-WAN, only data dedicated for this device identity (DevEUI, AppEUI) will be received and the user will be notified by the received data by this event.

  The event data attached to this event in the callback is a tuble that carry the following key information:

  - `data` a byte array object containing the actual received data.
  - `RSSI` an integer value represnting the RSSI of the received signal.
  - `SNR` an integer value represnting the SNR of the received signal.
  - `port` the port number on which this data is received.
  - `DR` the data rate of the received data.
  - `dl_frame_counter` The LoRa-WAN parameter (Downlink Frame Counter).

lora._event.EVENT_RX_FAIL

This event is only generated in case of the **LoRa-RAW** mode when the user request to receive something and the receiving operation failed to be fulfilled.

lora._event.EVENT_RX_TIMEOUT

This event is only generated in case of the **LoRa-RAW** mode when the user request to receive something within a certain timeout and the receiving operation instantiated successfully, but nothing is received within the user given timeout.

lora._event.EVENT_TX_DONE

This event occurs when the user request to sent data on the air and the device managed successfully to fullfil the sending operation.

In case of the **LoRa-WAN**, it depends on whether the user wants a confirmation from the network upon receiving this message or not.

- If a confirmation is requested, this event will not occur at all, and the user should be waiting for either `lora._event.EVENT_TX_CONFIRM` or `lora._event.EVENT_TX_FAILED`
- If a confirmation is not requested, this event will be generated upon an an operation fulfillment from the device perspective without waiting a network confirmation on the requested transmission data.

lora._event.EVENT_TX_CONFIRM

This event is only generated in case of the **LoRa-WAN** mode when the user request to send data with a network confirmation. If the device manages to send the data and received a network confirmation upon this data, this event will be generated.

lora._event.EVENT_TX_FAILED

This event occurs in the following cases:

- In **LoRa-RAW**: If the user requested to send a data and the device failed to fullfil the transmission operation.
- In **LoRa-WAN**: It can occur in two distinct situations:
  - If the user requested to send a data to the network and the device failed to fullfil the transmission operation from the device side.
  - If the user wants a reception confirmation from the network side and the confirmation is not received.

lora._event.EVENT_TX_TIMEOUT

This event is generated if the device failed to fullfill the transmission operation within the user provided timeout(deadline) time.

LoRa Callback Generic Interface

To set a callback routine to the LoRa Stack, the following generic interface shall be used at the micropython level:

```
lora.callback(
    handler = <callback-routine>
    [, trigger = <dedicated-event>]
    [, port = <dedicated-port>]
    )
```

The parameters description is as follows:

- ``**handler**`` This is the real micropython function to be called when a LoRa event occurs.

  The handler takes an argument called `context` which is a tuple carrying all needed information attached to event that triggerred the callback. The `context` elemnts are:

key	Valid Mode	event
`event`	WAN, RAW	All Events
`msg_id`	WAN	Only any TX event
`data`	WAN, RAW	`lora._event.EVENT_RX_DONE`
`RSSI`	WAN, RAW	`lora._event.EVENT_RX_DONE`
`SNR`	WAN, RAW	`lora._event.EVENT_RX_DONE`
`port`	WAN	`lora._event.EVENT_RX_DONE`
`DR`	WAN	`lora._event.EVENT_RX_DONE`
`dl_frame_counter`	WAN	`lora._event.EVENT_RX_DONE`

  The following is a typical example of the LoRa callback function

```
def lora_generic_callback(context):

  event = context.get('event')

  # --- lora raw case
  if lora.mode() == lora._mode.RAW:

      if event == lora._event.EVENT_RX_DONE:
          print(f'received data: {context.get('data')}')
          pass
      elif event == lora._event.EVENT_RX_FAIL:
          pass
      elif event == lora._event.EVENT_RX_TIMEOUT:
          pass
      elif event == lora._event.EVENT_TX_DONE:
          pass
      elif event == lora._event.EVENT_TX_CONFIRM:
          # unexpected event in lora-raw
          pass
      elif event == lora._event.EVENT_TX_FAILED:
          pass
```

(continues on next page)

(continued from previous page)

```
    elif event == lora._event.EVENT_TX_TIMEOUT:
        pass
    else:
        print('error: unknown error')

# --- lora wan case
elif lora.mode() == lora._mode.WAN:

    msg_id = context.get('msg_id')

    if event == lora._event.EVENT_RX_DONE:
        print(f'received data: {context.get('data')}')
        pass
    elif event == lora._event.EVENT_RX_FAIL:
        # unexpected event in lora-wan
        pass
    elif event == lora._event.EVENT_RX_TIMEOUT:
        # unexpected event in lora-wan
        pass
    elif event == lora._event.EVENT_TX_DONE:
        print(f"Message ID {msg_id} has transmitted successfully")
        pass
    elif event == lora._event.EVENT_TX_CONFIRM:
        print(f"Message ID {msg_id} has been confirmed")
        pass
    elif event == lora._event.EVENT_TX_FAILED:
        print(f"Message ID {msg_id} is not transmitted")
        pass
    elif event == lora._event.EVENT_TX_TIMEOUT:
        print(f"Message ID {msg_id} tx timeout")
        pass
    else:
        print('error: unknown error')

else:
    print('error: unknown lora mode')
    pass

pass
```

- ``**trigger**`` It is an optional argument to set a callback routine dedicated for a certain lora stack event. It shall be equal to one or more of the expected mode lora events. If more than one event shall be used they shall be combined using an `OR` operation (ex: `lora._event.EVENT_TX_TIMEOUT | lora._event.EVENT_RX_TIMEOUT`)
- ``**port**`` It is an optional argument applicable only for LoRa-WAN mode. It gives the user more flexibility in callbacks to be able to receive lora-stack events for dedicated LoRa-WAN port in a specialized callback routine.

Example LoRa-RAW

```
# initialization
import lora
lora.mode(lora._mode.RAW)

# generic method for conatant value retrieval
def get_class_const_name(__class, __const):
    for k,v in __class.__dict__.items():
        if v == __const:
            return k
    return 'unknown'

# any event callback
def lora_raw_callback(context):
    print('lora-raw event: {} with data: {}'.format(
        get_class_const_name(lora._event, context['event']), context))
    pass

def lora_raw_callback_rx_done(context):
    event = context.get('event')
    if event != lora._event.EVENT_RX_DONE:
        print("unexpected event in this callback")
        return
    print('lora-raw received data: {}'.format(context))
    pass

def lora_raw_callback_timeout(context):
    event = context.get('event')
    if event != lora._event.EVENT_RX_TIMEOUT and \
            event != lora._event.EVENT_TX_TIMEOUT:
        print("unexpected event in this callback")
        return
    print('lora-raw timeout event: {}'.format(
        get_class_const_name(lora._event, event)))
    pass

# registering the callbacks
lora.callback( handler = lora_raw_callback )    # for all event

lora.callback(      # specialized callback for EVENT_RX_DONE event
    handler = lora_raw_callback_rx_done,
    trigger = lora._event.EVENT_RX_DONE )

# to specialize a callback for two different events, the events shall be
# combined using OR operation
lora.callback(      # specialized callback for EVENT_TX_TIMEOUT event
    handler = lora_raw_callback_timeout,
    trigger = lora._event.EVENT_TX_TIMEOUT |
              lora._event.EVENT_RX_TIMEOUT
    )
```

(continues on next page)

(continued from previous page)

```
# at this point the incoming lora events will be like this:
#   [ event ]                       [ triggered callback ]
#   ---------                       ----------------------
#   lora._event.EVENT_RX_DONE       lora_raw_callback_rx_done()
#   lora._event.EVENT_RX_FAIL       lora_raw_callback()
#   lora._event.EVENT_RX_TIMEOUT    lora_raw_callback_timeout()
#   lora._event.EVENT_TX_DONE       lora_raw_callback()
#   lora._event.EVENT_TX_CONFIRM    -- unexpected --
#   lora._event.EVENT_TX_FAILED     lora_raw_callback()
#   lora._event.EVENT_TX_TIMEOUT    lora_raw_callback_timeout
```

Example LoRa-WAN

```
# basic initialization
import lora                  # init lora stack
lora.mode(lora._mode.WAN)    # switch to lora-wan
# make sure that the device is commissioned before completing the following
lora.stats()

# defining some example callbacks

def get_class_const_name(__class, __const):
    for k,v in __class.__dict__.items():
        if v == __const:
            return k
    return 'unknown'

def lora_wan_callback(context):
    event = context.get('event')
    print('lora-wan event: {} with data: {}'.format(
        get_class_const_name(lora._event, event), context))
    pass

def lora_wan_callback_port_1_any(context):
    event = context.get('event')
    print('lora-wan port 1 event: {} with data: {}'.format(
        get_class_const_name(lora._event, event), context))
    pass

def lora_wan_callback_port_1_tx_confirm(context):
    event = context.get('event')
    if event != lora._event.EVENT_TX_CONFIRM:
        print("unexpected event in this callback")
        return
    print(f'lora-wan port 1 tx confirm on message id : {context}')
    pass

def lora_wan_callback_port_2_timeout(context):
```

(continues on next page)

(continued from previous page)

```
    event = context.get('event')
    if event != lora._event.EVENT_RX_TIMEOUT and \
            event != lora._event.EVENT_TX_TIMEOUT:
        print("unexpected event in this callback")
        return
    print('lora-wan port 2 timeout event: {}'.format(
        get_class_const_name(lora._event, event)))
    pass

# respective ports shall be opened before registering their specialized␣
↪callbacks
lora.port_open(1)
lora.port_open(2)

# registering callbacks
lora.callback(  # for all event on all port
    handler = lora_wan_callback
    )

lora.callback(  # for port 1 events only
    handler = lora_wan_callback_port_1_any,
    port = 1
    )

lora.callback(  # for port 1 event lora._event.EVENT_TX_CONFIRM only
    handler = lora_wan_callback_port_1_any,
    port = 1,
    trigger = lora._event.EVENT_TX_CONFIRM
    )

lora.callback(  # for port 2 events lora._event.EVENT_RX_TIMEOUT and
                # event != lora._event.EVENT_TX_TIMEOUT only
    handler = lora_wan_callback_port_2_timeout,
    port = 1,
    trigger = lora._event.EVENT_RX_TIMEOUT |
              lora._event.EVENT_TX_TIMEOUT
    )
```

Remarks

If only a specialized callback is defined and registered, the user will be able to listen to this specialized event only and not the other at all.

In LoRa-WAN mode; Registering a callback for a didicated port which is opened yet, will be ignored. The call back registeration shall be done again after opening the port.

9.2.5 lte Module - C Implementation

The `lte` module provides MicroPython bindings for LTE modem functionality using a robust C implementation built on top of the ESP modem library. This is the new implementation that replaces the Python-based `LTE.py` module.

Usage

```
import lte

# Initialize modem
lte.init(carrier='standard')

# Attach to network
lte.attach(apn='iot.1nce.net')

# Start data session
lte.connect()

# Check connection
if lte.isconnected():
    print("Connected!")
    print(lte.ifconfig())

# Clean up
lte.disconnect()
lte.deinit()
```

Methods

lte.init([carrier='standard'])

Initialize the LTE modem subsystem. This method is idempotent - it's safe to call multiple times.

The modem intelligently handles three possible power states:

1. **Powered off** - Powers on and initializes from scratch
2. **Normal AT mode** (crash recovery) - Resumes from existing state
3. **CMUX mode** (soft reset) - Cleans up and reinitializes

Parameters:

- `carrier` (str, optional): Carrier conformance mode for band selection. Options:
  - `'standard'` (default)
  - `'verizon'`
  - `'att'`
  - `'docomo'`
  - `'kddi'`
  - `'telstra'`

  - `'tmo'`
  - `'verizon-no-roaming'`
  - `'3gpp-conformance'`

Carrier Conformance Mode Behavior:

The `carrier` parameter controls the modem's carrier conformance mode (AT+SQNCTM), which affects band selection and carrier-specific optimizations:

- **``lte.init()`` or ``lte.init(carrier=None)``**: Uses whatever conformance mode is currently configured on the modem. Does not change or check the mode.
- **``lte.init(carrier='standard')``** or any explicit carrier: Checks the modem's current conformance mode and changes it if different. If the mode needs to change, the modem will automatically reset during initialization.
- **Changing conformance mode after initialization**: If you call `lte.init(carrier='verizon')` after already initializing with a different carrier, the modem will automatically change the conformance mode and reset. No need to call `lte.deinit()` first - the change is handled seamlessly.

Example:

```
# Use default - keeps whatever mode is configured
lte.init()

# Explicitly set to standard mode
lte.init(carrier='standard')

# Switch to Verizon mode - seamlessly changes and resets modem
lte.init(carrier='verizon')

# Switch back to standard - automatic mode change
lte.init(carrier='standard')

# Attach after mode change
lte.attach(apn='iot.1nce.net')
```

Note: Changing the conformance mode triggers a modem reset. The initialization process handles this automatically (including the baudrate switch from 115200 to 921600), but it adds a few seconds to the init time.

lte.attach([apn=None, type='IP', cid=None, band=None, bands=None])

Enable radio and attach to the cellular network. This is a **non-blocking** operation - use `lte.isattached()` to poll for registration status.

Note: NB-IoT initial attachment may take several minutes.

Parameters:

- `apn` (str, optional): Access Point Name (e.g., `'iot.1nce.net'`). Defaults to empty string (carrier default)
- `type` (str, optional): PDP context type. Options: `'IP'` (default), `'IPV6'`, `'IPV4V6'`

- `cid` (int, optional): Context Identifier. Default: 1 (Verizon uses 3)
- `band` (int, optional): Single frequency band to use. 0 for auto-select. Valid bands: 1-28, 66, 71
- `bands` (list, optional): List of frequency bands (e.g., `[2, 4, 12]`). Cannot use with `band`

Example:

```
# Simple attach with APN
lte.attach(apn='iot.1nce.net')

# Attach with specific band
lte.attach(apn='iot.1nce.net', band=12)

# Attach with multiple bands
lte.attach(apn='iot.1nce.net', bands=[2, 4, 12])

# Wait for attachment
import time
for i in range(180):  # 3 minute timeout
    if lte.isattached():
        print("Attached!")
        break
    time.sleep(1)
```

lte.connect([cid=None])

Start a data session using CMUX multiplexing and PPP protocol. This is a **non-blocking** operation - use `lte.isconnected()` to poll for PPP status.

Prerequisites: Network must be attached (`lte.isattached()` returns `True`)

Parameters:

- `cid` (int, optional): Context Identifier (typically uses CID from `attach()`)

Example:

```
# Wait for connection
lte.connect()
import time
for i in range(60):
    if lte.isconnected():
        print("Connected!")
        print(lte.ifconfig())
        break
    time.sleep(1)
```

lte.disconnect()

Disconnect from the data session but keep network attachment.

Example:

```
lte.disconnect()
```

lte.detach()

Gracefully detach from the cellular network and disable radio functionality.

Example:

```
lte.detach()
```

lte.deinit([detach=True, power_off=True])

Deinitialize the modem with optional control over network detachment and power off.

This method allows flexible power management strategies:

- **Full shutdown** (default): Detaches from network and powers off modem
- **Low power mode**: Keeps modem powered and attached, using PSM/eDRX for power saving
- **Quick restart**: Keeps modem powered for faster re-initialization

Parameters:

- `detach` (bool, optional): If `True`, detach from cellular network. Default: `True`
- `power_off` (bool, optional): If `True`, power off the modem completely. Default: `True`

Power Saving Strategy:

When using `detach=False, power_off=False`, the modem stays attached to the network in a low-power state:

- Flow control pins are de-asserted, allowing the modem to enter power saving mode
- Use PSM (Power Saving Mode) or eDRX (extended DRX) for ultra-low power consumption
- Modem can wake the ESP32 via RING signal for incoming messages or mobile-terminated events
- ESP32 can enter deep sleep while modem maintains network connection

Example:

```
# Full shutdown (default)
lte.deinit()

# Keep modem powered for quick restart
lte.deinit(power_off=False)

# Low power mode with network attachment (PSM/eDRX)
# Configure PSM/eDRX first with AT commands
lte.send_at_cmd('AT+CPSMS=1,"","","00000001","00000001"')  # Enable PSM
lte.deinit(detach=False, power_off=False)
# ESP32 can now deep sleep, modem wakes it via RING on incoming data

# Detach but keep powered (for reconfiguration)
lte.deinit(detach=True, power_off=False)
```

Use Cases:

1. **Normal shutdown**: `lte.deinit()` - Clean disconnect and power off
2. **Quick restart**: `lte.deinit(power_off=False)` - Faster next `lte.init()`
3. **Ultra-low power with connectivity**: `lte.deinit(detach=False, power_off=False)` + PSM/eDRX
4. **Reconfigure network**: `lte.deinit(detach=True, power_off=False)` - Change APN/bands

lte.reset()

Perform a hardware reset on the modem using `AT^RESET`. Waits for the modem to shutdown and reboot, then re-detects baudrate.

This function can take up to 10 seconds to complete.

Example:

```
lte.reset()
```

lte.isattached()

Check if the modem is attached to the cellular network.

Returns: `True` if registered (home or roaming), `False` otherwise

Example:

```
if lte.isattached():
    print("Attached to network")
```

lte.is_attached()

Alias for `lte.isattached()`. Provided for compatibility with legacy scripts.

Returns: `True` if attached, `False` otherwise

lte.isconnected()

Check if PPP data connection is active and IP address has been obtained.

Returns: `True` if connected, `False` otherwise

Example:

```
if lte.isconnected():
    ip_info = lte.ifconfig()
    print(f"IP: {ip_info[0]}")
```

lte.is_connected()

Alias for `lte.isconnected()`. Provided for compatibility with legacy scripts.

Returns: `True` if connected, `False` otherwise

lte.send_at_cmd(cmd='AT', [timeout=-1, wait_ok_error=False, check_error=False, buffer_size=4096])

Send a raw AT command to the modem and return the response.

> **Important:** CMUX allows simultaneous AT commands and data sessions.

Parameters:

- `cmd` (str): AT command to send (without `\r\n`). Default: `'AT'`
- `timeout` (int, optional): Timeout in milliseconds. -1 for default (5000ms)
- `wait_ok_error` (bool, optional): Wait for OK/ERROR response. Default: False
- `check_error` (bool, optional): Raise exception on ERROR. Default: False
- `buffer_size` (int, optional): Response buffer size (1024-32768). Default: 4096

Returns: Response string with OK/ERROR lines removed

Example:

```
# Query network registration
response = lte.send_at_cmd('AT+CEREG?')
print(response)

# Check signal strength
response = lte.send_at_cmd('AT+CSQ')
print(response)

# Long-running command
response = lte.send_at_cmd('AT+SQNINS', wait_ok_error=True, timeout=30000)
```

lte.mode([new_mode=None])

Get or set the modem operating mode (CAT-M1 or NB-IoT).

Parameters:

- `new_mode` (int, optional): New mode to set. Use `lte.CATM1` (0) or `lte.NBIOT` (1)

Returns:

- If no parameter: Current mode (0 for CAT-M1, 1 for NB-IoT)
- If parameter provided: None (mode is set)

  > **Note:** Setting a new mode causes the modem to reset.

Example:

```
# Get current mode
current = lte.mode()
print(f"Mode: {'CAT-M1' if current == 0 else 'NB-IoT'}")

# Switch to NB-IoT
lte.mode(lte.NBIOT)
```

lte.imei()

Get the modem's IMEI (International Mobile Equipment Identity) number.

Returns: String containing the 15-digit IMEI

Example:

```
imei = lte.imei()
print(f"IMEI: {imei}")
```

lte.iccid()

Get the SIM card's ICCID (Integrated Circuit Card Identification) number.

This function checks if the SIM is present and ready before reading the ICCID.

Returns: String containing the 19-20 digit ICCID

Raises: OSError if SIM card is not present or not ready

Example:

```
try:
    iccid = lte.iccid()
    print(f"ICCID: {iccid}")
except OSError:
    print("SIM card not present or not ready")
```

lte.get_signal_strength()

Get signal strength information from the modem.

Returns: Tuple of `(rssi, rssi_dbm, ber)` where:

- `rssi`: Raw RSSI value (0-31, 99=unknown)
- `rssi_dbm`: RSSI in dBm (-113 to -51, -999=unknown)
- `ber`: Bit Error Rate (0-7, 99=unknown)

Example:

```
rssi, rssi_dbm, ber = lte.get_signal_strength()
print(f"Signal: {rssi_dbm} dBm (BER: {ber})")
```

lte.get_status()

Get comprehensive modem status information.

Returns: Dictionary with the following keys:

- `powered` (bool): Modem power state
- `sim_ready` (bool): SIM card present and ready
- `network_attached` (bool): Attached to network
- `ppp_connected` (bool): PPP session active
- `cmux_active` (bool): CMUX multiplexing enabled
- `baudrate` (int): Current UART baudrate
- `rssi` (int): Signal strength (raw RSSI value)
- `ber` (int): Bit error rate

Example:

```
status = lte.get_status()
print(f"Powered: {status['powered']}")
print(f"Network: {status['network_attached']}")
print(f"PPP: {status['ppp_connected']}")
print(f"Signal: {status['rssi']}")
```

lte.ifconfig()

Get network interface configuration (IP address information).

Returns: Tuple of `(ip, netmask, gateway, dns)` or `None` if not connected

Example:

```
if lte.isconnected():
    ip, netmask, gateway, dns = lte.ifconfig()
    print(f"IP: {ip}")
    print(f"Gateway: {gateway}")
    print(f"DNS: {dns}")
```

lte.power_on([wait_ok=True])

Power on the LTE modem hardware via IO expander.

Parameters:

- `wait_ok` (bool, optional): Wait for modem to respond with OK. Default: True

Example:

```
lte.power_on()
# or
lte.power_on(wait_ok=False)  # Don't wait for response
```

lte.power_off([force=False])

Power off the LTE modem hardware via IO expander.

Parameters:

- `force` (bool, optional): Force immediate power off without graceful shutdown. Default: False

Example:

```
lte.power_off()  # Graceful shutdown
# or
lte.power_off(force=True)  # Immediate power off
```

lte.is_powered()

Check if the modem is currently powered on.

Returns: `True` if powered, `False` otherwise

Example:

```
if lte.is_powered():
    print("Modem is powered on")
```

lte.check_sim_present()

Check if a SIM card is present and ready.

This function retries up to 5 times with delays, and ensures the radio is enabled to read the SIM.

Returns: `True` if SIM is ready, `False` otherwise

Example:

```
if lte.check_sim_present():
    iccid = lte.iccid()
    print(f"SIM ready: {iccid}")
else:
    print("No SIM card detected")
```

lte.set_event_handler(handler, event_mask=EVENT_ALL, lock=False)

Register a unified event handler for LTE modem events, providing structured event data with automatic parsing.

Parameters:

- `handler` (function): Event handler function that receives a dictionary, or `None` to unregister
- `event_mask` (int, optional): Bitmask of events to subscribe to. Default: `lte.EVENT_ALL`
- `lock` (bool, optional): If `True`, lock the event handler so it cannot be overridden until `lte.deinit()` is called. Attempting to set or unregister the handler while locked raises `OSError`. Default: `False`

Event Handler Signature:

```
def handler(event: dict) -> None:
    """
    Args:
        event: Dictionary containing:
            - 'type': Event type constant (lte.EVENT_xxx)
            - Additional keys depending on event type
    """
    pass
```

Event Types:

1. **``lte.EVENT_REGISTRATION_STATUS`` (0x0001)** - Network registration changes
   - `stat` (int): Registration status (0-10, 80)
     - 0: Not registered (not searching)
     - 1: Registered (home network)
     - 2: Not registered (searching)
     - 3: Registration denied
     - 5: Registered (roaming)
   - `tac` (str, optional): Tracking Area Code
   - `ci` (str, optional): Cell ID
   - `act` (int, optional): Access Technology (-1=unknown, 7=LTE-M, 9=NB-IoT)
   - `cause_type` (int, optional): Reject cause type
   - `reject_cause` (int, optional): Reject cause code
   - `active_time` (str, optional): PSM active time
   - `periodic_tau` (str, optional): PSM periodic TAU
2. **``lte.EVENT_PPP_CONNECTED`` (0x0002)** - PPP connection established
   - `ip` (str): Assigned IP address
   - `netmask` (str): Network mask
   - `gateway` (str): Gateway address
   - `dns1` (str): Primary DNS server
   - `dns2` (str, optional): Secondary DNS server
3. **``lte.EVENT_PPP_DISCONNECTED`` (0x0004)** - PPP connection lost
4. **``lte.EVENT_MODEM_CRASH`` (0x0008)** - Modem crash detected
   - `break_count` (int): Number of break signals received
5. **``lte.EVENT_MODEM_RESET`` (0x0010)** - Modem was reset
   - `user_initiated` (bool): Whether reset was user-initiated
   - `reason` (str, optional): Reset reason
6. **``lte.EVENT_SIGNAL_QUALITY`` (0x0020)** - Signal strength update

- `rssi` (int): Raw RSSI value (0-31, 99=unknown)
- `rssi_dbm` (int): RSSI in dBm (-113 to -51, -999=unknown)
- `ber` (int): Bit Error Rate (0-7, 99=unknown)

7. **``lte.EVENT_URC`` (0x0040)** - Unsolicited response code (raw AT response)
   - `data` (str): The raw URC string
8. **``lte.EVENT_ERROR`` (0x0080)** - Error occurred
   - `error_code` (int): Error code
   - `message` (str, optional): Error message
   - `operation` (str, optional): Operation that failed

Event Mask Combinations:

You can combine event types using bitwise OR to subscribe to specific events:

```
# Subscribe to connection events only
lte.set_event_handler(handler, lte.EVENT_PPP_CONNECTED | lte.EVENT_PPP_
↪DISCONNECTED)

# Subscribe to registration and signal quality
lte.set_event_handler(handler, lte.EVENT_REGISTRATION_STATUS | lte.EVENT_
↪SIGNAL_QUALITY)

# Subscribe to all events (default)
lte.set_event_handler(handler, lte.EVENT_ALL)

# Lock the handler so it cannot be overridden (released on lte.deinit())
lte.set_event_handler(handler, lte.EVENT_ALL, lock=True)
```

Locking:

When `lock=True` is passed, the handler is protected from being overridden or unregistered. Any subsequent call to `lte.set_event_handler()` (including `lte.set_event_handler(None)`) will raise `OSError` until `lte.deinit()` is called, which releases the lock.

This is useful for system-level code (e.g., the CTRL client) that needs to guarantee its event handler remains active while managing the modem connection:

```
# System code locks the handler
lte.init()
lte.set_event_handler(system_handler, lte.EVENT_ALL, lock=True)

# User code cannot override it
try:
    lte.set_event_handler(my_handler)  # Raises OSError
except OSError as e:
    print(e)  # "event handler is locked, call lte.deinit() first"

# Lock is released on deinit
lte.deinit()
```

(continues on next page)

(continued from previous page)

```
lte.init()
lte.set_event_handler(my_handler)  # Works now
```

Basic Example:

```
def lte_event_handler(event):
    event_type = event['type']

    if event_type == lte.EVENT_REGISTRATION_STATUS:
        stat = event['stat']
        if stat == 1:
            print("✓ Registered (home network)")
        elif stat == 5:
            print("✓ Registered (roaming)")
        elif stat == 2:
            print("🔍 Searching...")

    elif event_type == lte.EVENT_PPP_CONNECTED:
        print(f"✓ Connected! IP: {event['ip']}")

    elif event_type == lte.EVENT_PPP_DISCONNECTED:
        print("📡 Disconnected")

# Register handler for all events
lte.init()
lte.set_event_handler(lte_event_handler, lte.EVENT_ALL)
lte.attach(apn='iot.1nce.net')
lte.connect()

# Unregister when done
lte.set_event_handler(None)
```

Complete Example: See `/home/ehlers/sg-sdk/examples/lte/lte_event_handler.py` for a comprehensive example showing all event types and how to handle them.

Constants

Mode Constants

- `lte.CATM1` (0): CAT-M1 mode constant
- `lte.NBIOT` (1): NB-IoT mode constant

Example:

```
# Set mode to CAT-M1
lte.mode(lte.CATM1)

# Set mode to NB-IoT
lte.mode(lte.NBIOT)
```

Event Type Constants

Event type constants for use with `lte.set_event_handler()`:

- `lte.EVENT_REGISTRATION_STATUS` (0x0001): Network registration status changes
- `lte.EVENT_PPP_CONNECTED` (0x0002): PPP connection established
- `lte.EVENT_PPP_DISCONNECTED` (0x0004): PPP connection lost
- `lte.EVENT_MODEM_CRASH` (0x0008): Modem crash detected
- `lte.EVENT_MODEM_RESET` (0x0010): Modem reset occurred
- `lte.EVENT_SIGNAL_QUALITY` (0x0020): Signal strength update
- `lte.EVENT_URC` (0x0040): Raw unsolicited response code
- `lte.EVENT_ERROR` (0x0080): Error occurred
- `lte.EVENT_ALL` (0xFFFF): Subscribe to all events

Example:

```
# Subscribe to specific events
lte.set_event_handler(my_handler, lte.EVENT_PPP_CONNECTED | lte.EVENT_PPP_
↪DISCONNECTED)

# Subscribe to all events
lte.set_event_handler(my_handler, lte.EVENT_ALL)
```

Error Handling

The module raises MicroPython exceptions for errors:

- `OSError` - General modem errors (timeout, not responding, etc.)
- `ValueError` - Invalid parameters
- `MemoryError` - Memory allocation failure
- `TypeError` - Wrong type for callback

Example:

```
try:
    lte.init()
    lte.attach(apn='iot.1nce.net')
except OSError as e:
    print(f"LTE error: {e}")
except ValueError as e:
    print(f"Invalid parameter: {e}")
```

Implementation Notes

Differences from LTE.py

This C implementation provides several improvements over the Python `LTE.py` module:

1. **Better Resource Management:**

- Proper UART reservation/release
- Memory-efficient buffers
- WiFi/LTE coexistence handled automatically

2. **CMUX Support:**
   - Simultaneous AT commands and data sessions
   - No need for `pause_ppp()`/`resume_ppp()`
   - More reliable operation
3. **Event-Driven Architecture:**
   - Unified event handler with structured data
   - No need to manually poll for status changes
   - Real-time event notification with automatic parsing
4. **Robust Initialization:**
   - Handles modem power states intelligently
   - Automatic crash recovery
   - Idempotent initialization
5. **Performance:**
   - Native C implementation
   - Faster command execution
   - Lower memory footprint

Compatibility Notes

Most methods are compatible with `LTE.py`, but note:

- `pause_ppp()` and `resume_ppp()` are **not needed** (CMUX handles this)
- `read_rsp()` is replaced by `set_event_handler()`
- Initialization is simpler (no explicit baudrate management)
- `check_power()` renamed to `is_powered()`
- Automatic WiFi conflict detection/resolution

Advanced Usage

Monitoring Network Registration with Event Handler

```
import lte
import time

def lte_event_handler(event):
    """Monitor network registration and connection events"""
    event_type = event['type']
```

(continues on next page)

(continued from previous page)

```
    if event_type == lte.EVENT_REGISTRATION_STATUS:
        stat = event['stat']
        states = {
            0: "Not registered (not searching)",
            1: "Registered (home network)",
            2: "Not registered (searching)",
            3: "Registration denied",
            5: "Registered (roaming)"
        }
        print(f"Network: {states.get(stat, f'Unknown ({stat})')}")

        # Show location info if available
        if 'tac' in event and event['tac']:
            print(f"  Tracking Area: {event['tac']}")
        if 'ci' in event and event['ci']:
            print(f"  Cell ID: {event['ci']}")
        if 'act' in event and event['act'] != -1:
            act_names = {7: "LTE-M", 9: "NB-IoT"}
            print(f"  Technology: {act_names.get(event['act'], 'Unknown')}")

    elif event_type == lte.EVENT_SIGNAL_QUALITY:
        rssi_dbm = event['rssi_dbm']
        print(f"Signal: {rssi_dbm} dBm")

    elif event_type == lte.EVENT_PPP_CONNECTED:
        print(f"Connected! IP: {event['ip']}")

    elif event_type == lte.EVENT_PPP_DISCONNECTED:
        print("Disconnected - implementing reconnection...")
        # Application reconnection logic here

lte.init()
lte.set_event_handler(lte_event_handler, lte.EVENT_ALL)
lte.attach(apn='iot.1nce.net')

# Wait and monitor via events
for i in range(180):
    if lte.isattached():
        print("Attached! Connecting...")
        lte.connect()
        break
    time.sleep(1)

# Events continue to be monitored automatically
time.sleep(60)

lte.set_event_handler(None)
lte.disconnect()
```

(continues on next page)

(continued from previous page)

```
lte.deinit()
```

Connection Events Only

Subscribe only to connection-related events for simple connection monitoring:

```
import lte

def connection_handler(event):
    """Handle only connection events"""
    if event['type'] == lte.EVENT_PPP_CONNECTED:
        print(f"✓ Connected: {event['ip']}")
    elif event['type'] == lte.EVENT_PPP_DISCONNECTED:
        print("⍰ Disconnected - reconnecting...")
        lte.connect()  # Auto-reconnect

lte.init()
# Subscribe only to PPP events
lte.set_event_handler(
    connection_handler,
    lte.EVENT_PPP_CONNECTED | lte.EVENT_PPP_DISCONNECTED
)
lte.attach(apn='iot.1nce.net')
lte.connect()
```

Complete Connection Example

```
import lte
import time

def connect_lte(apn, timeout=180):
    """Connect to LTE with timeout"""
    print("Initializing...")
    lte.init()

    # Check SIM
    if not lte.check_sim_present():
        raise RuntimeError("No SIM card")

    print(f"ICCID: {lte.iccid()}")
    print(f"IMEI: {lte.imei()}")

    # Attach
    print("Attaching...")
    lte.attach(apn=apn)

    start = time.time()
    while time.time() - start < timeout:
```

(continues on next page)

(continued from previous page)

```
        if lte.isattached():
            rssi, rssi_dbm, ber = lte.get_signal_strength()
            print(f"Attached! Signal: {rssi_dbm} dBm")
            break
        time.sleep(1)
    else:
        raise TimeoutError("Attachment timeout")

    # Connect
    print("Connecting...")
    lte.connect()

    start = time.time()
    while time.time() - start < 60:
        if lte.isconnected():
            ip_info = lte.ifconfig()
            print(f"Connected! IP: {ip_info[0]}")
            return
        time.sleep(1)
    else:
        raise TimeoutError("Connection timeout")

# Use it
try:
    connect_lte('iot.1nce.net')

    # Now you can use sockets!
    import socket
    s = socket.socket()
    s.connect(('example.com', 80))
    # ...

finally:
    lte.disconnect()
    lte.deinit()
```

Logging and Debugging

The `lte` module uses the SG-SDK structured logging system with two components:

- ``**espmodem**`` - Low-level ESP modem library operations (UART, AT commands, internal state)
- ``**modlte**`` - High-level MicroPython bindings (function calls, operations, results)

Enabling Logging

Enable logging at the start of your script before importing `lte`:

```
import logs
```

(continues on next page)

(continued from previous page)

```
# Enable the entire 'lte' subsystem
logs.filter_subsystem('lte', True)

# Enable specific components for granular control
logs.filter_component('lte', 'espmodem', True)  # Low-level modem operations
logs.filter_component('lte', 'modlte', True)    # High-level Python bindings

import lte
```

Logging Levels

Each component logs at different levels:

- **INFO** - High-level operations (function calls, status changes)
- **DEBUG** - Detailed operation flow (AT commands, responses, state transitions)
- **WARN** - Recoverable issues (retries, dropped events)
- **ERROR** - Failures requiring attention

What Gets Logged

``modlte`` component logs:

- Function calls with parameters: `lte.init(carrier='standard')`
- AT command execution: `AT: AT+CEREG=2`
- AT command responses: `AT response: +CEREG: 2,0`
- Operation results: `lte.isattached() -> True`
- Callback registration and URC handling
- High-level operation flow (attach, connect, disconnect)

``espmodem`` component logs:

- Modem initialization and power state detection
- UART communication details
- ESP modem library events
- CMUX multiplexing operations
- PPP session management
- Low-level error handling and retries

Example Output

```
import logs
logs.filter_component('lte', 'modlte', True)
import lte
```

(continues on next page)

(continued from previous page)

```
lte.init()
lte.attach(apn='iot.1nce.net')
response = lte.send_at_cmd('AT+CEREG?')
```

Output:

```
|000:00:01-234|  info  |1:mp_task |lte.init       |   lte   |   modlte    |␣
↪lte.init(carrier='standard')
|000:00:01-345|  info  |1:mp_task |lte.init       |   lte   |   modlte    |␣
↪LTE initialization complete
|000:00:02-456|  info  |1:mp_task |lte.attach     |   lte   |   modlte    |␣
↪lte.attach(apn='iot.1nce.net', type='IP', cid=1)
|000:00:02-567|  info  |1:mp_task |send_at_cmd    |   lte   |   modlte    |␣
↪AT: AT+CEREG?
|000:00:02-678|  info  |1:mp_task |send_at_cmd    |   lte   |   modlte    |␣
↪AT response: +CEREG: 2,5,"001E","059B2A79",7
```

Debugging Tips

For general troubleshooting:

```
# Enable only high-level operations
logs.filter_component('lte', 'modlte', True)
```

For deep debugging:

```
# Enable both components for full visibility
logs.filter_component('lte', 'espmodem', True)
logs.filter_component('lte', 'modlte', True)
```

For production:

```
# Disable logging or enable only errors
logs.filter_subsystem('lte', False)
```

Troubleshooting Event Handler Issues

When debugging event handlers, enable `modlte` logging to see:

- Handler registration: `"Event handler registered successfully"`
- Event reception: `"Event handler called from UART task context"`
- Event scheduling: `"Event successfully scheduled for processing"`
- Handler execution: `"Handler is valid and callable, invoking Python function"`

```
import logs
logs.filter_component('lte', 'modlte', True)
import lte

def event_handler(event):
```

(continues on next page)

(continued from previous page)

```
    print(f"[Event] {event['type']}: {event}")

lte.init()
lte.set_event_handler(event_handler)  # Watch logs for registration␣
↪confirmation
lte.attach(apn='iot.1nce.net')
```

9.2.6 Constructors

class LTE.LTE(...)

Create and configure a LTE object. See __init__ for params of configuration.

```
from LTE import LTE
lte = LTE()
```

9.2.7 Methods

lte.__init__([carrier='standard', cid=1, mode=None, baudrate=115200, debug=None])

This method is used to set up the LTE subsystem. Optionally specify

- `carrier name`. The currently available options are:
  - `'att'`
  - `'verizon'`
  - `'standard'`
  - `'docomo'`
  - `'kddi'`
  - `'telstra'`
  - `'tmo'`
  - `'verizon-no-roaming'`
  - `'3gpp-conformance'`
- `cid` is the connection id. Most operators use cid=1 except Verizon which uses cid=3 when using a Verizon issued SIM card
- `mode` is LTE.CATM1 or LTE.NBIOT. If not specified, modem will use current setting
- `baudrate` is the speed with which the modem is operating. Default is 115200bps
- `debug`. True or False, display additional debugging output

lte.deinit([reset=False])

Disables LTE modem completely. This reduces the power consumption to the minimum. Call this before entering deepsleep. Optional parameter reset was added for compatibility with legacy code and is not used

lte.attach([apn=None, type='IP', cid=None, band=None, bands=None])

Enable radio functionality and attach to the LTE network authorised by the inserted SIM card. Optionally specify:

- `band` : to scan for networks. If no band (or `None`) is specified, the currently configured bands will be scanned (this is persistent through resetting the modem). The possible values for the band are: `1,2,3,4,5,8,12,13,17,18,19,20,25,26,28,66 or 71`.
- `bands` : a tuple of mutiple band entries (see band above). **Note**: When using the new C-based LTE driver, `bands` must be a list of **integers** (e.g., `[1, 3, 8]`), not strings. The legacy Python-based LTE.py driver accepted either format, but the new driver requires strict typing.
- `apn` : Specify the APN (Access point Name).
- `cid` : connection ID, see `LTE.\_\_init()\_\_` and `LTE.connect()`. when the ID is set here it will be remembered when doing connect so no need to specify again
- `type` : PDP context type either `IP` or `IPV4V6`. These are options to specify PDP type 'Packet Data protocol' either IP [Internet Protocol] or IPV4V6 [Internet Protocol version 4 and version 6] , that depend on what the Network supports.

Migration Note: If migrating from the legacy LTE.py driver, ensure band values are converted to integers:

```
# Legacy (accepted both):
bands = ['1', '3', '8']  # worked with LTE.py
bands = [1, 3, 8]        # also worked

# New driver (requires integers):
bands = [1, 3, 8]                              # correct
bands = [int(b) for b in '1,3,8'.split(',')] # convert from string
```

lte.isattached()

Returns `True` if the cellular mode is attached to the network. `False` otherwise.

lte.is_attached()

Returns `True` if the cellular mode is attached to the network. `False` otherwise. Same as `lte.isattached()` and provided for compatibility with legacy scripts

lte.detach()

Gracefully detach the modem from the LTE-M network and disable the radio functionality.

lte.connect([cid=None])

Start a data session and obtain and IP address. Optionally specify a CID (Connection ID) for the data session. The arguments are:

- `cid`: connection ID, see `LTE.\_\_init()\_\_` and `LTE.attach()`.

lte.isconnected()

Returns `True` if there is an active LTE data session and IP address has been obtained. `False` otherwise.

lte.is_connected()

Returns `True` if there is an active LTE data session and IP address has been obtained. `False` otherwise.

lte.disconnect()

End the data session with the network.

lte.send_at_cmd(cmd, [timeout=-1, wait_ok_error=False, check_error=False])

Send an AT command directly to the modem. Returns the raw response from the modem as a string object. You can find the possible AT commands here.

> If a data session is active (i.e. the modem is *connected*), you will need to `lte.pause_ppp()` and `lte.resume_ppp` around the AT command.

Example:

```
lte.send_at_cmd('AT+CEREG?')     # check for network registration manually␣
↪(sames as lte.isattached())
```

Optionally the response can be parsed for pretty printing:

- `timeout` : specify the timeout milliseconds the esp32 chip will wait after the AT command to receive the response. -1 means wait forever
- `wait_ok_error` : wait for the modem to respond with OK or ERROR after sending the command. Not all commands return OK or ERROR which means the command might be waiting forever. Some commands such as `AT+SQNINS` run longer than the maximum timeout, and setting `wait_ok_error=True` is required to get the results
- `check_error` : Will check if an error occured and raise an exception. Helpful in scripts that should abort if an error occurs.

lte.reset()

Perform a hardware reset on the cellular modem. This function can take up to 5 seconds to return as it waits for the modem to shutdown and reboot.

lte.pause_ppp()

Suspend PPP session with LTE modem. this function can be used when needing to send AT commands which is not supported in PPP mode.

lte.resume_ppp()

Resumes PPP session with LTE modem.

lte.mode([new_mode=None])

If no parameter is specified, return the current operating mode (0 for LTE.CATM1 and 1 for LTE.NBIOT) If `new_mode` is specified, switches the modem into the specified operating mode. Use LTE.CATM1 or LTE.NBIOT Example: `lte.mode(new_mode=LTE.CATM1)`

The modem will reset and switch to the new operating mode

lte.power_on([wait_ok=True])

Turn the LTE modem power on. Will optionally wait until the modem answers with OK.

lte.power_off([force=False])

Turn off power to the LTE modem. Will gracefully shut down the LTE connection unless force=True is used.

lte.check_sim_present()

Check if a SIM card is present and readable

lte.check_power()

Returns True if the lte module is powered on, otherwise false

lte.print_pretty_response(rsp, [flush=False, prefix=None])

Removes unnecessary line feed and OK/ERROR output from a modem response before printing the response on the REPL

lte.return_pretty_response(resp)

Removes unnecessary line feed and OK/ERROR output from a modem response before returning the resp

lte.read_rsp([size=None, timeout=-1, wait_ok_error=False, check_error=False])

This function allows reading unsolicited responses from the modem. These are responses sent by the modem without being requested using `lte.send_at_cmd()`. The response can be formatted with `lte.return_pretty_response` and `lte.print_pretty_response` if desired.

lte.check_ppp()

Function will raise an exception if the modem is in active ppp mode.

lte.ifconfig()

Function will return a tuple of IP address information from the PPP stack

lte.imei()

Function will return the IMEI (International Mobile Equipment Identity) number from the LTE module

lte.iccid()

Function will return the SIM card ICCID (Integrated Circuit Card Identification number). This function will check if a SIM card is present.

9.2.8 Constants

- `LTE.CATM1` : For use in CATM1 mode
- `LTE.NBIOT` : For use in NBIOT mode

9.3 Peripherals & System

9.3.1 RGB-LED (WS2812B)

Contents

- Introduction
- RGB LED Functions

Introduction

The RGB-LED feature is supported for the board shield that have the RGB-LED module on it. If so, the RGB-LED firmware interface component will be activated and build with the final firmware image.

This interface component utilizes the open-source driver for this LED which can be found at this git repository https://github.com/JSchaenzle/ESP32-NeoPixel-WS2812-RMT

The interface component offers some extra services such as heartbeat and decorated light sequencing.

RGB LED Functions

- `initialization`: the module will be initialized once it is imported. after its initialization, it could be deinitialized by calling `rgbled.deinit()` and to initialize it again use `rgbled.initialize()`
- `rgbled.color()` to set the LED color continuously. The color follows this hex formatting `xxRRGGBB`, in which the `RR`, `GG` and `BB` are representint the `red`, `green` and `blue` components of the color respectively and `xx` is a don't care value.

```
# rgbled.color() example:
import rgbled
rgbled.color(0x00FF0000)    # sets the LED color to red
rgbled.color(0x0000FF00)    # sets the LED color to green
rgbled.color(0x000000FF)    # sets the LED color to blue
rgbled.color(0x00FFFF00)    # sets the LED color to yellow
```

- `rgbled.heartbeat()` to start the heartbeat blinking service. it has three signature as follows:

```
(1) rgbled.heartbeat()
(2) rgbled.heartbeat( <enable> )
(3) rgbled.heartbeat( <color>, <cycle-time>, <blink-percentage> )
```

the description of each signature is as follows:

1. check the current status of the heartbeat service and returns `True` or `False`.
2. to enable/disable the service
3. to set new configuration for the service and start or restart it

the description of the available argument are:

- `<enable>` the enable or disable flag and it is a boolean value.
- `<color>` the color value similar to the `rgbled.color()` function.
- `<cycle-time>` the total period of the duty-cycle (the light on + light off periods).
- `<blink-percentage>` the percentage(p) value (0 < p < 100) where the light is on

```
# rgbled.heartbeat() example:
import rgbled

rgbled.heartbeat()          # check the status of the service
    # returns False

rgbled.heartbeat( True )    # start the heartbeat service
    # start with the default configs
rgbled.heartbeat()          # check the status of the service
    # returns True

# to set the blue color blinking for about 200 msec each one second.
rgbled.heartbeat(0x000000FF, 1000, 20)
    # new config is set and service restarted
rgbled.heartbeat()          # check the status of the service
    # returns True

rgbled.heartbeat( False )   # stop the heartbeat service

# to set the red color blinking for about 10 msec each 50 ms. (very␣
↪fast)
rgbled.heartbeat(0x00FF0000, 50, 20)
    # new config is set and service started
rgbled.heartbeat()          # check the status of the service
    # returns True

rgbled.heartbeat( True )    # start the heartbeat service
    # it will start with latest config (0x00FF0000, 50, 20)
rgbled.heartbeat()          # check the status of the service
    # returns True
```

- `rgbled.decoration()` It provide a fancy way of doing a decorative light blinking by specifying a sequence of blinking descriptors. it follows the following syntax:

```
rgbled.decoration( <blink-desc-list>, <repeat> )

<blink-desc-list>  ::= [ <blink-desc-tuple>, ... ]
```

(continues on next page)

(continued from previous page)

```
<blink-desc-tuple> ::= ( <color-value>,
                         <duty-period>,
                         <light-on-percent>,
                         <loop-count> )
```

where:

- `<blink-desc-list>` is a list of four elements tuples to describe a time window of blinking.
- `<blink-desc-tuple>` a tuple which specify a time window of blinking.
- `<color-value>` the color value as described in `rgbled.color()`
- `<duty-period>` the total light blinking duty cycle
- `<light-on-percent>` the ligh on time percentage of the duty cycle period
- `<loop-count>` number of repetition of this duty cycle period

Example:

```
# assume we want the following time light sequence
#
#  ___ 50 ___            ___ 50 ___           _____________ _____________
# | G |__| G |__________| B |__| B |__________|      R      |      Y      |
# |<--------2 Sec----->|<--------2 Sec----->|<- 0.5 sec->|<- 0.5 sec->|
#
# where G and R period are 50 msec
# The sequence above shall be repeated and a one second color should be
# off between each repetition
#
import rgbled
rgbled.decoration([
  (0x00001100, 100, 50, 2),    # the first two G's pulses
  (0, 2000 - 200, 0, 1),       # the light off between G's and B's␣
↪pulses
  (0x00000011, 100, 50, 2),    # the second two B's pulses
  (0, 2000 - 200, 0, 1),       # the light off after b's pulses
  (0x00110000, 500, 100,1),    # the R period
  (0x00111100, 500, 100,1),    # the Y period
  (0, 1000, 0, 1)              # the light off time before repeating
],
True)                          # repeat the whole sequence again
```

- `rgbled._color` it is a class carrying the basic color definitions, it can be used directly in place of the color value.

```
import rgbled
rgbled.color( rgbled._color.RED )
rgbled.color( rgbled._color.GREEN )
rgbled.color( rgbled._color.BLUE )
rgbled.color( rgbled._color.YELLOW )
rgbled.color( rgbled._color.MAGENTA )
```

(continues on next page)

(continued from previous page)

```
rgbled.color( rgbled._color.CYAN )
rgbled.color( rgbled._color.WHITE )
```

9.3.2 Non-Volatile Storage (NVS) Interface

Contents

Introduction

The NVS (Non-Volatile Storage) interface provides access to the ESP32's NVS flash memory system. NVS is used to persistently store key-value pairs, configuration data, and other application state that must survive system resets and power cycles.

The NVS interface component exposes the underlying ESP NVS library functionality through a micropython module, allowing applications to read, write, and manage data stored in dedicated NVS partitions.

NVS Statistics

`nvs_if.stat()` displays comprehensive statistics about the NVS partitions and their contents. It can be called with optional filters to view specific partitions or namespaces.

Function Signature

```
(1) nvs_if.stat()
(2) nvs_if.stat( dump_blobs=<bool> )
(3) nvs_if.stat( partition=<name>, namespace=<name>, dump_blobs=<bool> )
```

Parameters

- `dump_blobs` (bool, default: False) - If True, displays hexadecimal content of BLOB entries in addition to their size.
- `partition` (str, optional) - Filter output to a specific NVS partition (e.g., "nvs"). If omitted, all partitions are displayed.
- `namespace` (str, optional) - Filter output to a specific namespace within the partition(s).

Output Format

The output displays all key-value pairs organized by partition and namespace, with the following information for each entry:

- **namespace** - The namespace the key belongs to
- **key** - The key name
- **type** - The data type (U8, I32, STR, BLOB, etc.)

- **val** - The current value or size (for BLOBs)

At the end of each partition, a summary is displayed showing:

- **partition size** - Total capacity of the NVS partition in bytes
- **used entries** - Number of entries currently used / total available entries
- **free** - Percentage of available free space

Examples

```
import nvs_if

# Display all NVS statistics for all partitions
nvs_if.stat()

# Display statistics for a specific partition
nvs_if.stat(partition="nvs")

# Display statistics for a specific namespace
nvs_if.stat(namespace="app")

# Display all entries with BLOB hex data
nvs_if.stat(dump_blobs=True)
```

Example Output

```
================================================================================
namespace   key              type    val
================================================================================

---( nvs )-----------------------------------------------------------------
 ↪-----------------------
ctrl        ztp              U8      0
lora-stack  lora-mgr         BLOB    size: 12
lora-stack  lw-radio-proc    BLOB    size: 20
app         frame            STR     Frame #3
app         iframe           I32     3
phy         cal_data         BLOB    size: 1904
phy         cal_version      U32     540

    partition size : 24576 bytes
    used entries   : 32 / 126
    free           : 74%
================================================================================
```

NVS Query

`nvs_if.exists()` checks whether a key exists in NVS storage, either in a specific partition and namespace or in one of the default partitions.

Function Signature

```
nvs_if.exists( key, namespace, partition=<name> )
```

Parameters

- `key` (str) - The key name to check for existence. If None, checks for namespace existence only.
- `namespace` (str) - The namespace in which to search.
- `partition` (str, optional) - The NVS partition name. If omitted, searches the default "nvs" partition.

Return Value

- **True** if the key (or namespace) exists
- **False** if the key (or namespace) does not exist

Examples

```
import nvs_if

# Check if a key exists in the default partition
if nvs_if.exists(key="ztp", namespace="ctrl"):
    print("ZTP key found")
else:
    print("ZTP key not found")

# Check namespace existence
if nvs_if.exists(key=None, namespace="app"):
    print("app namespace exists")

# Check in a specific partition
if nvs_if.exists(key="data", namespace="myapp", partition="nvs"):
  print("data key found in nvs partition")
```

NVS Key-Value Operations

`nvs_if.set()` stores or updates a key-value pair in NVS storage.

Function Signature

```
nvs_if.set( key, value, namespace, partition=<name> )
```

Parameters

- `key` (str) - The key name
- `value` - The value to store (bytes, int, or str, depending on intended type)
- `namespace` (str) - The namespace in which to store the key
- `partition` (str, optional) - The NVS partition name. If omitted, uses the default "nvs" partition

Return Value

- **True** if the operation was successful
- **False** if the operation failed

Examples

```
import nvs_if

# Store an integer value
nvs_if.set(key="counter", value=42, namespace="app")

# Store a string value
nvs_if.set(key="ssid", value="WiFiNetwork", namespace="config")

# Store binary data (BLOB)
nvs_if.set(key="data", value=b"\x01\x02\x03\x04", namespace="app")

# Store in a specific partition
nvs_if.set(key="setting", value=100, namespace="app", partition="nvs")
```

Reading Values

`nvs_if.get()` retrieves a stored value from NVS storage.

Function Signature

```
nvs_if.get( key, namespace, partition=<name>, default=None )
```

Parameters

- `key` (str) - The key name to retrieve
- `namespace` (str) - The namespace containing the key
- `partition` (str, optional) - The NVS partition name. If omitted, searches the default "nvs" partition
- `default` (optional) - Value to return if the key is not found. If not specified and key doesn't exist, raises an exception.

Return Value

The stored value, or the default value if the key doesn't exist.

Examples

```
import nvs_if

# Retrieve a value
counter = nvs_if.get(key="counter", namespace="app")
print(f"Counter: {counter}")

# Retrieve with a default value
ssid = nvs_if.get(key="ssid", namespace="config", default="DefaultSSID")

# Retrieve from a specific partition
setting = nvs_if.get(key="setting", namespace="app", partition="nvs")
```

9.3.3 System Information

Contents

- Introduction
- Board Information
- version Information
- Other Information

Introduction

System information collects different information from different system components and displays it to the user.

Board Information

`sysinfo.board()` This method used to get the board related information. It returns a dict object with the following keyword fields:

- `full_name` a complete name for the board comrising the OEM module name, part number and the shield name if available.
- `platform` a platform name of the used OEM module.
- `module_name` the used OEM module name such as `F1`, `F1-C`, `F1-L`, ...
- `module_number` the corresponding part number of the module such as: `SGW3201`, `SGW3501`, ...
- `shield` the used shield for the OEM module. such as `StarterKit`. If the shield is not available, it will be `OEM`.

Example

```
import sysinfo
board_info = sysinfo.board()
print( board_info )
print( f'full name      : {board_info.full_name}' )
print( f'platform       : {board_info.platform}' )
print( f'module name    : {board_info.module_name}' )
print( f'module number : {board_info.module_number}' )
print( f'shield         : {board_info.shield}' )
```

the output will be as similar to the following:

```
(full_name='SGW3201-F1-L-OEM', platform='F1', module_name='F1-L', module_
↪number='SGW3201', shield='OEM')
full name      : SGW3201-F1-L-OEM
platform       : F1
module name    : F1-L
module number : SGW3201
shield         : OEM
```

`sysinfo.show_board()` displays directly the board information on screen as in the following example:

```
>>> sysinfo.show_board()
================================== board info␣
↪==================================
 - board full name            SGW3201-F1-L-OEM
 - platform                   F1
 - board name                 F1-L
 - board number               SGW3201
 - board shield               OEM

 - micropython board name     SGWireless SGW3201-F1-L-OEM
 - micropython MCU name       ESP32S3
 - micropython system name    F1-L
```

Version Information

`sysinfo.version()` This method used to get the version related information. It returns a dict object with the following keyword fields:

(major=0, minor=1, patch=0, git_delta=1, git_tag='0b532043', build_date='2024.04.18', build_time='07:30', custom='dirty', release='v0.1.0', build='v0.1.0-1-0b532043-20240418-dirty')

- `major` an integer containing the version major number.
- `minor` an integer containing the version minor number.
- `patch` an integer containing the version patch number.
- `git_delta` an integer containing the number of commits between the base release git commit and the current build git commit.
- `git_tag` a string representing the build git commit short hash(8 characters).
- `build_date` a string representing the date of this build in format (yyyy.mm.dd).

- `build_time` a string representing the time of this build in format (hh:mm).
- `custom` a string containing a user given custom string in the build command or it contains `dirty` word if no provided custom version string in the build command and the build source has uncommitted changes. Otherwise, it is empty.
- `release` contains the base release version of this build.
- `build` contains the complete build version of this build.

Example

```
import sysinfo
version_info = sysinfo.version()
print( version_info )
print( f'base release  : {version_info.release}' )
print( f'build version : {version_info.build}' )
print( f'date and time : {version_info.build_date} - {version_info.build_
↪time}')
```

the output will be as similar to the following:

```
(major=0, minor=1, patch=0, git_delta=1, git_tag='0b532043', build_date=
↪'2024.04.18', build_time='07:30', custom='dirty', release='v0.1.0', build=
↪'v0.1.0-1-0b532043-20240418-dirty')
base release  : v0.1.0
build version : v0.1.0-1-0b532043-20240418-dirty
date and time : 2024.04.18 - 07:30
```

`sysinfo.show_version()` displays directly the version information on screen as in the following example:

```
>>> sysinfo.show_version()
================================ firmware version␣
↪================================
 - firmware version           v0.1.0-1-0b532043-20240418-bug#123-test-fix
 - firmware base release      v0.1.0
 - custom version string      bug#123-test-fix

 - build date and time        2024.04.18 - 08:44

 - git hash long              0b532043c19433554dbb65b33878fb8ba6643516
 - git hash short             0b532043
 - git delta                  1

 - micropython build          v1.19.1-796-gf4811b0b4
 - micropython build date     2024-04-18
================================================================================
```

Other Information

`sysinfo.show_efuses()` displays the eFuses information of the board. as in the following example

```
>>> sysinfo.show_efuses()
============================== efuses for user data␣
↪==============================
 - Layout Version              00
 - LoRa MAC                    70 b3 d5 49 92 a6 92 0f
 - Serial Number               00 00 00 00 00 00
 - HW ID                       01 00 00
 - Project ID                  00 00 00
 - WiFi MAC                    7c 51 89 02 03 40
================================================================================
```

`sysinfo.show_flash()` displays the flash storage information and the current deployed partition table as in the following example:

```
>>> sysinfo.show_flash()
================================== flash stats␣
↪==================================
 - flash frequency             40 MHz
 - flash size                  16777216  Bytes ~= 16 MB
 - partition table:
    label     type  subtype     enc   start      end      size_b  size_kb␣
↪size_mb
    nvs       data  nvs         no  00011000  0001bfff     45056     44.0  ␣
↪0.0
    otadata   data  factory     no  0001c000  0001dfff      8192      8.0  ␣
↪0.0
    config    data  data-undef  no  0001e000  0001efff      4096      4.0  ␣
↪0.0
    rfu1      data  data-undef  no  0001f000  0001ffff      4096      4.0  ␣
↪0.0
    factory   app   factory     no  00020000  0029ffff   2621440   2560.0  ␣
↪2.5
    ota_0     app   ota         no  002a0000  0051ffff   2621440   2560.0  ␣
↪2.5
    ota_1     app   ota         no  00520000  0079ffff   2621440   2560.0  ␣
↪2.5
    rfu2      data  data-undef  no  007a0000  007a017f       384      0.4  ␣
↪0.0
    vfs       data  data-undef  no  00800000  00ffffff   8388608   8192.0  ␣
↪8.0
================================================================================
```

`sysinfo.show_spiram()` displays the spiram related information as in the following example:

```
>>> sysinfo.show_spiram()
================================== spiram stats␣
↪==================================
 - ram size                    8388608 Bytes ~= 8 MB
```

(continues on next page)

(continued from previous page)

```
================================================================================
```

`sysinfo.show_all()` displays all above system information together.

9.3.4 IO Expander Interface (PCAL6408A)

Contents

Introduction

The IO Expander interface provides access to the PCAL6408A 8-bit I2C GPIO expander available on SGW3501-F1-StarterKit boards. This component extends the available GPIO pins and provides interrupt-capable I/O expansion.

I2C Bridge Architecture

The IO expander implementation uses MicroPython I2C Bridge architecture for ESP-IDF v5.4+ compatibility:

- **Driver**: PCAL6408A 8-bit I2C GPIO expander
- **Bridge**: `mp_i2c_bridge` component provides C interface to MicroPython I2C
- **Benefits**: Unified I2C management, modern ESP-IDF compatibility, future-proof

Implementation Details

- **Device Address**: 0x20 (7-bit I2C address)
- **I2C Pins**: SCL=20, SDA=21 (configurable)
- **Frequency**: 100kHz I2C bus frequency
- **Integration**: Uses `ioexp_mp.c` with I2C bridge for MicroPython builds

PCAL6408A Features

- 8-bit I2C-bus GPIO with interrupt and weak pull-up
- 5V tolerant inputs
- Polarity inversion register

- Low current consumption
- Interrupt output for pin change detection
- Compatible with standard GPIO operations

MicroPython Interface

Initialization

```
import ioexp

# Initialize the IO expander
ioexp.init()
```

Pin Configuration

```
# Set pin direction (1 = input, 0 = output)
ioexp.set_direction(pin_number, direction)

# Example: Set pin 0 as output, pin 1 as input
ioexp.set_direction(0, 0)  # Output
ioexp.set_direction(1, 1)  # Input
```

Digital I/O Operations

```
# Write to output pin
ioexp.write_pin(pin_number, value)

# Example: Set pin 0 high
ioexp.write_pin(0, 1)

# Read from input pin
value = ioexp.read_pin(pin_number)

# Example: Read pin 1 state
state = ioexp.read_pin(1)
print(f"Pin 1 state: {state}")
```

Port Operations

```
# Write entire 8-bit port at once
ioexp.write_port(0b10101010)  # Set pins 1,3,5,7 high

# Read entire port
port_state = ioexp.read_port()
print(f"Port state: 0x{port_state:02X}")
```

Pull-up Configuration

```
# Enable/disable weak pull-up resistors
ioexp.set_pullup(pin_number, enable)

# Example: Enable pull-up on pin 2
ioexp.set_pullup(2, 1)
```

Polarity Inversion

```
# Configure polarity inversion
ioexp.set_polarity(pin_number, inverted)

# Example: Invert polarity on pin 3
ioexp.set_polarity(3, 1)
```

Complete Example

```
import ioexp
import time

# Initialize the IO expander
ioexp.init()

# Configure pins: 0-3 as outputs, 4-7 as inputs with pull-ups
for pin in range(4):
    ioexp.set_direction(pin, 0)  # Output

for pin in range(4, 8):
    ioexp.set_direction(pin, 1)  # Input
    ioexp.set_pullup(pin, 1)     # Enable pull-up

# Blink LEDs on output pins
while True:
    # Set outputs high
    for pin in range(4):
        ioexp.write_pin(pin, 1)

    # Read and display input states
    for pin in range(4, 8):
        state = ioexp.read_pin(pin)
        print(f"Input pin {pin}: {state}")

    time.sleep(0.5)

    # Set outputs low
    for pin in range(4):
        ioexp.write_pin(pin, 0)

    time.sleep(0.5)
```

C/C++ Interface

For native C/C++ applications, the same functionality is available through the header interface:

```
#include "ioexp.h"

// Initialize
esp_err_t err = ioexp_init();

// Configure pin direction
ioexp_set_direction(0, IOEXP_OUTPUT);
ioexp_set_direction(4, IOEXP_INPUT);

// Digital operations
ioexp_write_pin(0, 1);  // Set pin 0 high
int state = ioexp_read_pin(4);  // Read pin 4
```

Hardware Connection

The PCAL6408A is typically connected as follows on SGW3501-F1-StarterKit:

- **VCC**: 3.3V power supply
- **GND**: Ground
- **SCL**: I2C clock line (GPIO 20)
- **SDA**: I2C data line (GPIO 21)
- **INT**: Interrupt output (optional, board-dependent)
- **ADDR**: Address select pin (determines I2C address)

Troubleshooting

Common Issues

1. **I2C Communication Errors**
   - Verify I2C connections (SCL, SDA, power, ground)
   - Check I2C address (0x20 default)
   - Ensure proper pull-up resistors on I2C lines
2. **Pin State Issues**
   - Verify pin direction configuration
   - Check for polarity inversion settings
   - Ensure proper voltage levels (5V tolerance)
3. **Initialization Failures**
   - Confirm device presence with I2C scan
   - Verify power supply stability
   - Check for I2C bus conflicts

Debug Commands

```
# Test I2C communication
import machine
i2c = machine.I2C(0, scl=20, sda=21)
devices = i2c.scan()
print(f"I2C devices found: {[hex(d) for d in devices]}")

# Expected output should include 0x20 for PCAL6408A
```

Register Map

For advanced users, the PCAL6408A register map:

Register	Address	Function
Input Port	0x00	Read input levels
Output Port	0x01	Write output levels
Polarity Inversion	0x02	Configure input polarity
Configuration	0x03	Set pin direction
Output Drive Strength	0x40	Configure drive strength
Input Latch	0x42	Input latch control
Pull-up/Pull-down Enable	0x43	Pull resistor enable
Pull-up/Pull-down Select	0x44	Pull resistor direction
Interrupt Mask	0x45	Interrupt enable
Interrupt Status	0x46	Interrupt status
Output Port Configuration	0x4F	Output port configuration

9.3.5 Fuel-Gauge (BQ27421)

Contents

- Introduction
- Fuel-Gauge Functions
- Battery Info Fields
- Example

Introduction

The Fuel-Gauge feature is supported for board shields that have the BQ27421 fuel gauge IC on board. If present, the fuel-gauge firmware interface component will be activated and built with the final firmware image.

This interface component utilizes the open-source driver for this fuel gauge which can be found at this git repository https://github.com/svcguy/lib-BQ27421/tree/master

I2C Bridge Architecture

The fuel gauge implementation uses the MicroPython I2C Bridge architecture for ESP-IDF v5.4+ compatibility:

- **Driver**: BQ27421 fuel gauge IC communicates via I2C
- **Device Address**: 0x55 (7-bit I2C address)
- **Bridge**: `mp_i2c_bridge` component provides a C interface to MicroPython I2C
- **I2C Pins**: SCL and SDA are configurable (defaults in `fuel_gauge.config`)
- **Frequency**: 100 kHz I2C bus frequency

Fuel-Gauge Functions

- `fuel_gauge.init()` initializes the module with given battery parameters. It accepts the following optional keyword arguments:

```
fuel_gauge.init( designCapacity_mAh=<int>,
                 minSysVoltage_mV=<int>,
                 taperCurrent_mA=<int> )
```

  - `designCapacity_mAh` (int, default: 1200) - the design capacity of the battery in mAh.
  - `minSysVoltage_mV` (int, default: 0) - an optional system minimum operating voltage in mV. Pass 0 if not specified.
  - `taperCurrent_mA` (int, default: 0) - an optional taper current detection threshold of the charger (including charger tolerances). The charger will stop charging below this value. Pass 0 if not specified.

  Raises `OSError` if the fuel gauge cannot be initialized (e.g. no battery connected).

- `fuel_gauge.deinit()` deinitializes the module. Before using the module again it has to be re-initialized using `fuel_gauge.init()`.
- `fuel_gauge.info()` reads current information from the fuel gauge IC and returns a named tuple with all battery parameters. See Battery Info Fields for the complete list of returned fields. Raises `OSError` if the module is not initialized or the battery is not present.
- `fuel_gauge.print()` reads and prints all fuel gauge information in a human-readable format to the console output.

Battery Info Fields

The named tuple returned by `fuel_gauge.info()` contains the following fields:

Field	Type	Unit	Description
`voltage_mV`	int	mV	Current battery voltage
`current_mA`	int	mA	Current draw (positive = charging)
`temp_degC`	float	°C	Battery temperature
`charge_percent`	int	%	State of Charge (SOC)
`health_percent`	int	%	State of Health (SOH)
`designCapacity_mAh`	int	mAh	Battery design capacity
`remainingCapacity_mAh`	int	mAh	Current usable capacity remaining
`fullChargeCapacity_mAh`	int	mAh	Current full charge capacity
`isCritical`	bool	—	Battery voltage critically low
`isLow`	bool	—	Battery voltage low
`isFull`	bool	—	Battery fully charged
`isCharging`	bool	—	Battery is currently charging
`isDischarging`	bool	—	Battery is currently discharging

Example

```
import fuel_gauge

# initialize with a 1500 mAh battery
fuel_gauge.init(designCapacity_mAh=1500)

# print formatted battery status to the console
fuel_gauge.print()
```

Example output of `fuel_gauge.print()`:

```
Voltage                4200 mV
Current                62 mA
Temprature             30.10 degC
Charge State           100 %
Health State           91 %
Design Capacity        1200 mAh
Remaining Capacity     497 mAh
Full Charge Capacity   497 mAh
is critical            no
is low                 no
is full                no
is charging            yes
is discharging         no
```

Using `fuel_gauge.info()` for scripting:

```
import fuel_gauge

fuel_gauge.init()

info = fuel_gauge.info()
```

(continues on next page)

(continued from previous page)

```
print(f"Battery voltage : {info.voltage_mV} mV")
print(f"Charge          : {info.charge_percent} %")
print(f"Health          : {info.health_percent} %")
print(f"Temperature     : {info.temp_degC} degC")
print(f"Remaining       : {info.remainingCapacity_mAh} mAh")

if info.isCharging:
    print("Battery is charging")
elif info.isDischarging:
    print("Battery is discharging")

if info.isCritical:
    print("WARNING: Battery critically low!")

# cleanup
fuel_gauge.deinit()
```

9.3.6 CAN API Documentation

Contents

- Functions
- Example

This document describes the CAN API functions

Functions

`can.init`

Initializes the CAN interface with the specified parameters.

- **Parameters**:
  - `RxPin` (default: 1): Receive pin number.
  - `TxPin` (default: 2): Transmit pin number.
  - `Baud` (default: 250000): Baud rate.(25000,50000,100000,125000,250000,500000,800000,1000000).
  - `Mode` (default: 0): CAN mode (e.g., 0-NORMAL, 1-NO ACK□2-LISTEN_ONLY).
- **Usage**: python can.init(RxPin, TxPin, Baud, Mode)

`can.deinit`

Deinitializes the CAN interface.

- **Usage**: python can.deinit()

can.send

Sends a CAN message.

- **Parameters**:
  - `flags`: Message flags.
  - `id`: Message identifier.
  - `dat`: Data to send (string or bytes).
- **Usage**: python can.send(flags, id, dat)

can.filter

Sets a CAN filter.

- **Parameters**:
  - `dat`: Filter data (string or bytes).
  - `single_filter`: Boolean indicating if a single filter is used.
- **Usage**: python can.filter(dat, single_filter)

can.any

Checks if any CAN messages are available.

- **Returns**: Integer indicating the presence of messages.
- **Usage**: python can.any()

can.recv

Receives a CAN message.

- **Returns**: Bytes representing the received message.
- **Usage**: python can.recv()

Example

```
from machine import Pin,UART,Timer,I2C,WDT
from micropython import const
import machine
import time
import can
import binascii
import struct

RxPin=39
TxPin=38
Baud=100000
Mode=0
```

(continues on next page)

(continued from previous page)

```
SendDelay=0
CanSendFlags=0#Extended frame
CanSendId=0
CanSendBuf=''
CanRecBytes=bytes()
CanSendBytes=bytes()
T1S=0

def SetCanFlags(extd,rtr,ss,self_rev,dlc_non_comp):
    global CanSendFlags
    CanSendFlags=0
    if extd:#0-standard frame; 1- Extended frame
        CanSendFlags |= 0x01
    if rtr:#0- data frame,1-Remote Frame
        CanSendFlags |= 0x02
    if ss:#0-Error resend; 1-Single send
        CanSendFlags |= 0x04
    if self_rev:#0- Do not receive messages sent by oneself, 1- Receive␣
↪messages sent by oneself
        CanSendFlags |= 0x08
    if dlc_non_comp:#0-Data length not exceeding 8 (ISO 11898-1); 1- Data␣
↪length greater than 8 (non-standard)
        CanSendFlags |= 0x10

def time0_irq(time0):
    global CanSendFlags,CanSendId,SendDelay,T1S,CanRecBuf
    if T1S<1000:
      T1S+=1
    else:
      T1S=0
    if SendDelay:
        SendDelay-=1
        if SendDelay==0:
            CanSendFlags = int.from_bytes(CanRecBytes[0:4], 'little')
            CanSendId= int.from_bytes(CanRecBytes[4:8], 'little')
            len = CanRecBytes[8]
            #CanSendBuf = CanRecBytes[9:9+len].decode('utf-8')
            CanSendBytes = CanRecBytes[9:9+len]
            #print(CanSendBuf)
            #can.send(CanSendId,CanSendBuf)
            can.send(CanSendFlags,CanSendId,CanSendBytes)
            #can.send(0x123,'12345678')
def CanFilter(acceptance_code,acceptance_mask,single_filter):
    dat = acceptance_code.to_bytes(4,'little')
    dat += acceptance_mask.to_bytes(4,'little')
    can.filter(dat,single_filter)

def setup():
    CanFilter(0,0xffffffff,True)
```

(continues on next page)

(continued from previous page)

```
    #CanFilter(0xe00001,0x1ffffe,True)
    #CanFilter(0xe00000,0x1fffff,False)
    #CanFilter(0xe0,0x1f,False)
    can.init(RxPin,TxPin,Baud,Mode)
    time0=Timer(0)
    time0.init(period=1,mode=Timer.PERIODIC,callback=time0_irq)

def loop():
    global SendDelay,CanRecBuf,CanRecBytes
    print('loop')
    while(1):
        if can.any():
            CanRecBytes = can.recv();
            print(CanRecBytes.hex(' '))
            SendDelay=200

if __name__=="__main__":
    setup()
    machine.lightsleep(10)
    loop()
```

9.3.7 eFuse Interface

Contents

- Introduction
- eFuse Read Functions
- eFuse Layout
- Example

Introduction

The eFuse interface provides read access to the ESP32-S3 one-time programmable (OTP) fuse memory. eFuses store factory-provisioned board identity data such as the LoRa-WAN DevEUI, WiFi MAC address, serial number, and hardware identifiers.

The values stored in eFuses are written once during manufacturing and cannot be changed afterwards. This module allows micropython scripts to retrieve these values at runtime.

An optional internal RAM cache can be enabled (`SDK_BOARD_EFUSE_INTERNAL_CACHE_ENABLE`) so that the eFuse block is read from hardware only once at startup, with subsequent reads served from memory.

eFuse Read Functions

All read functions take no arguments and return a `bytes` object containing the raw value read from the eFuse block. The module is imported as `efuse_if`.

- `efuse_if.layout_version()` returns the eFuse user-data block layout version (1 byte).
- `efuse_if.lora_mac()` returns the LoRa-WAN DevEUI (8 bytes).
- `efuse_if.serial_number()` returns the board serial number (6 bytes).

- `efuse_if.hw_id()` returns the manufacturer hardware ID (3 bytes).
- `efuse_if.project_id()` returns the project-specific ID (3 bytes).
- `efuse_if.wifi_mac()` returns the WiFi MAC address (6 bytes).

Optional LoRa Key Functions

The following functions are only available when LoRa key storage on eFuses is enabled (`SDK_BOARD_LORA_WAN_KEYS_ON_EFUSES`):

- `efuse_if.lora_app_key()` returns the LoRa-WAN OTAA AppKey (16 bytes).
- `efuse_if.lora_nwk_key()` returns the LoRa-WAN OTAA NwkKey (16 bytes).

Test Function (Virtual eFuse Mode Only)

When building with `CONFIG_EFUSE_VIRTUAL` enabled for development or testing:

- `efuse_if.write_test_lora_keys(AppKey=<bytes>, NwkKey=<bytes>)` writes test LoRa keys to the virtual eFuse block. Both keyword arguments must be provided as 16-byte buffers.

eFuse Layout

The board identity data is stored in **EFUSE_BLK3** (User Data) on the ESP32-S3. The current layout (version 0) is:

Field	Size	Description
LPWAN_MAC	8 bytes	LoRa-WAN DevEUI
SERIAL_NUMBER	6 bytes	Board serial number
HW_ID	3 bytes	Hardware ID
PROJECT_ID	3 bytes	Project-specific ID
Reserved	4 bytes	Reserved for future use
LAYOUT_VERSION	1 byte	Layout version number
WIFI_MAC	6 bytes	WiFi MAC address
CRC8	1 byte	CRC8 checksum

When LoRa key storage is enabled, the AppKey and NwkKey (16 bytes each) are stored in a separate eFuse KEY block (configurable, default KEY5).

Example

```
import efuse_if

# read the LoRa-WAN DevEUI
deveui = efuse_if.lora_mac()
print("DevEUI:", deveui.hex())

# read the WiFi MAC address
wifi = efuse_if.wifi_mac()
print("WiFi MAC:", wifi.hex())
```

(continues on next page)

(continued from previous page)

```
# read the board serial number
sn = efuse_if.serial_number()
print("Serial:", sn.hex())

# read hardware and project identifiers
hw = efuse_if.hw_id()
proj = efuse_if.project_id()
print("HW ID:", hw.hex())
print("Project ID:", proj.hex())

# read the layout version
ver = efuse_if.layout_version()
print("Layout version:", int.from_bytes(ver, 'little'))
```

Example output:

```
DevEUI: 70b3d54992a6920f
WiFi MAC: 7c5189020340
Serial: 000000000000
HW ID: 010000
Project ID: 000000
Layout version: 0
```

9.3.8 FUOTA Module - Firmware Update Over The Air

The `fuota` module provides Python APIs for managing OTA (Over-The-Air) firmware updates on ESP32 devices.

Features

- ✓ **Simple one-line upgrade**: `fuota.upgrade(url)` handles everything
- ✓ **Non-blocking mode**: Background OTA with status monitoring
- ✓ **Blocking mode**: Traditional synchronous operation
- ✓ **HTTPS support**: Built-in certificate bundle for secure downloads
- ✓ **Automatic partition management**: Selects next available OTA partition
- ✓ **Validation**: Verifies firmware image before flashing
- ✓ **Low memory usage**: Runs in C, no 64KB Python thread stack needed
- ✓ **Progress tracking**: Real-time status and progress monitoring
- ✓ **Percentage logging**: 1% increment progress updates
- ✓ **Network detection**: Automatic WiFi/LTE connectivity check
- ✓ **Automatic validation**: Validates new firmware on first boot
- ✓ **Soft reset safe**: Cleanly cancels OTA on Ctrl+D
- ✓ **Resume on network drops**: HTTP `Range` resume across TCP/TLS disconnects (ideal for NB-IoT / LTE-M / flaky Wi-Fi)

- ✓ **Exponential backoff retries**: Bounded retries with configurable backoff so the radio / NAT gets time to recover
- ✓ **Artifact identity check**: `ETag` / `Last-Modified` / total-size validated on every resume; server-side rebuilds force a clean restart
- ✓ **Optional chunk cap**: Reconnect every N bytes to sidestep carrier NAT / PSM timeouts even when data is flowing

Resume Behaviour (NB-IoT / LTE-M)

The download loop keeps the ESP-IDF OTA handle (and its erased target partition) alive across transport errors. When a read fails the HTTP client is closed cleanly, the task sleeps with exponential backoff, then reconnects with a `Range: bytes=<offset>-` request and keeps appending to the same partition. If the server ignores `Range` (returns HTTP 200) or the artifact identity changes mid-download (new `ETag` / `Last-Modified` / size) the upgrade automatically restarts from byte 0.

Transient conditions that now auto-recover instead of failing the upgrade: TLS read errors (`MBEDTLS_ERR_NET_CONN_RESET`, `ENOTCONN` on the HTTP socket), `esp_http_client_open` failures, premature EOF, HTTP 5xx responses, transient DNS failures.

Non-retryable conditions that still fail fast: HTTP 4xx, image validation (`ESP_ERR_OTA_VALIDATE_FAILED`), no OTA partition available, memory allocation failure for the RX buffer or OTA begin, `esp_ota_write` failure.

API Reference

`fuota.upgrade(url [, opts])`

BLOCKING MODE - Downloads and installs firmware synchronously.

Parameters:

- `url` (str, required): HTTPS/HTTP URL to firmware binary
- `opts` (dict, optional): Tunables. See *Options* below.

Returns:

- `True` on success, `False` on failure

Blocks until complete - REPL will be frozen during download and installation.

Example:

```
import fuota
import machine

# Blocking upgrade - waits for completion
if fuota.upgrade('https://example.com/firmware.bin'):
    print("Success!")
    machine.reset()
else:
    print("Failed!")
```

fuota.start_upgrade(url [, opts])

NON-BLOCKING MODE (RECOMMENDED) - Downloads and installs firmware in background.

Parameters:

- `url` (str, required): HTTPS/HTTP URL to firmware binary
- `opts` (dict, optional): Tunables. See *Options* below.

Returns:

- `None` - Returns immediately, use `fuota.status()` to monitor progress

Raises:

- `OSError`: If OTA already in progress, network error, or failed to create task

Example:

```
import fuota
import time
import machine

# Start OTA in background - returns immediately, tuned for NB-IoT
fuota.start_upgrade('https://example.com/firmware.bin', {
    'max_retries': 30,
    'timeout_ms': 90000,
    'max_chunk_bytes': 131072,   # reconnect every 128 KB
})

# Monitor progress
while True:
    status = fuota.status()
    print("state={} attempt={} offset={}/{} ({}%)".format(
        status['state'],
        status.get('attempts', 0),
        status['bytes_written'],
        status.get('total_size', '?'),
        status.get('percent', '?'),
    ))

    if status['state'] == 'success':
        print("Upgrade successful!")
        machine.reset()
        break
    elif status['state'] == 'failed':
        print("Upgrade failed: {}".format(status.get('err_msg', 'Unknown␣
↪error')))
        break

    time.sleep(1)
```

Options

All keys are optional; defaults are shown in parentheses.

Key	Default	Description
`max_`	20	Total HTTP attempts, including the first.
`back` `off_`	2000	**Delay before the first retry. Doubled on each subsequent failure,** capped by `backoff_max_ms`.
`back` `off_`	6000	Upper bound on the exponential backoff delay.
`time` `out_`	6000	Per-HTTP-request timeout (read / connect).
`rx_b`	4096	Receive buffer + OTA write block size (clamped to 512–16384).
`max_`	0	**0 = unlimited. Non-zero caps how much data is pulled per HTTP** connection; the next `Range` request resumes from where it left off. Useful to defeat carrier NAT / PSM idle timeouts on long downloads.

`fuota.status()`

Returns current OTA operation status (for non-blocking mode).

Returns: Dictionary with keys:

- `state` (str): `'idle'`, `'running'`, `'success'`, `'failed'`, or `'aborted'`
- `bytes_written` (int): Bytes downloaded so far
- `attempts` (int): Number of HTTP attempts used so far (1 on the first try)
- `resume_offset` (int): Offset of the most recent HTTP attempt
- `total_size` (int, optional): Total firmware size (if known)
- `percent` (int, optional): 0-100 progress (if `total_size` is known)
- `error_code` (int, optional): ESP-IDF error code (on failure)
- `err_msg` (str, optional): Error description (on failure)

Example:

```
>>> fuota.status()
{'state': 'running', 'bytes_written': 524288, 'total_size': 2621440,
 'percent': 20, 'attempts': 3, 'resume_offset': 458752}

>>> fuota.status()
{'state': 'failed', 'bytes_written': 100000, 'attempts': 20,
 'resume_offset': 98304, 'error_code': -1,
 'err_msg': 'max retries exceeded [ESP_FAIL]'}
```

---

fuota.abort_upgrade()

Cancels an ongoing OTA upgrade (non-blocking mode). The cancellation is cooperative: the download task notices the request at the next read / retry boundary, cleanly closes the HTTP client and releases the OTA partition handle. `fuota.status()` will transition to `'aborted'` shortly after the call returns.

Raises:

- `OSError`: If no OTA upgrade is in progress

Example:

```
fuota.start_upgrade('https://example.com/firmware.bin')
time.sleep(5)
fuota.abort_upgrade()  # Cancel after 5 seconds
```

---

fuota.info()

Prints current partition information to console.

Example:

```
>>> fuota.info()
Next update partition: ota_1
Running partition: ota_0
Boot partition: ota_0
```

---

fuota.partition_info()

Returns partition information as a named tuple (use this instead of `info()` for programmatic access).

Returns: Named tuple with fields:

- `next_update_partition` (str): Partition for next OTA
- `running_partition` (str): Currently running partition
- `boot_partition` (str): Partition system booted from

Example:

```
>>> info = fuota.partition_info()
>>> print(f"Running: {info.running_partition}, Next: {info.next_update_
↪partition}")
Running: ota_0, Next: ota_1

>>> info.running_partition
'ota_0'
```

---

fuota.valid()

Marks current firmware as valid, preventing automatic rollback.

Important: Call this after validating new firmware works correctly!

---

fuota.rollback()

Marks current firmware as invalid and reboots to previous version.

Low-Level API (Advanced)

fuota.start()

Begins manual OTA process.

fuota.write(data)

Writes firmware data chunk.

Parameters:

- `data` (bytes): Firmware data chunk

fuota.finish()

Completes manual OTA and sets boot partition.

Usage Patterns

Pattern 1: Non-Blocking Upgrade with Progress (Recommended)

```
import fuota
import time
import machine

url = 'https://example.com/firmware.bin'

# Start OTA in background
fuota.start_upgrade(url)
print("OTA started in background...")

# Monitor progress
while True:
    status = fuota.status()
    state = status['state']

    if state == 'running':
        bytes_written = status['bytes_written']
        total = status.get('total_size')
```

(continues on next page)

(continued from previous page)

```
        if total:
            pct = (bytes_written * 100) // total
            print(f"Downloading: {pct}% ({bytes_written}/{total} bytes)")
        else:
            print(f"Downloading: {bytes_written} bytes")

    elif state == 'success':
        print("✓ Upgrade successful! Rebooting...")
        time.sleep(1)
        machine.reset()
        break

    elif state == 'failed':
        print(f"✗ Upgrade failed: {status.get('err_msg', 'Unknown error')}")
        print(f"  Error code: {status.get('error_code')}")
        break

    elif state == 'aborted':
        print("OTA was aborted")
        break

    time.sleep(1)
```

Pattern 2: Simple Blocking Upgrade

```
import fuota
import machine

try:
    # Blocks until complete
    if fuota.upgrade('https://example.com/firmware.bin'):
        print("Upgrade successful, rebooting...")
        machine.reset()
    else:
        print("Upgrade failed")
except OSError as e:
    print(f"Error: {e}")
```

Pattern 3: User-Cancelable Upgrade

```
import fuota
import time
import sys

url = 'https://example.com/firmware.bin'
fuota.start_upgrade(url)

print("Upgrading... Press Ctrl+C to cancel")
```

(continues on next page)

(continued from previous page)

```
try:
    while True:
        status = fuota.status()
        if status['state'] in ('success', 'failed', 'aborted'):
            break
        time.sleep(0.5)

except KeyboardInterrupt:
    print("\nCanceling upgrade...")
    fuota.abort_upgrade()
    print("Upgrade canceled")
```

Pattern 4: Safe Upgrade with Automatic Validation

The firmware automatically validates on first boot after OTA! Just reset after successful upgrade:

```
import fuota
import machine

# No need to manually call fuota.valid() anymore!
if fuota.upgrade('https://example.com/firmware.bin'):
    print("Rebooting to new firmware...")
    machine.reset()  # New firmware auto-validates on boot
```

Pattern 5: Safe Upgrade with Manual Validation

```
import fuota
import machine

# Check current state
info = fuota.partition_info()
print(f"Current partition: {info.running_partition}")

# Perform upgrade
if fuota.upgrade('https://example.com/firmware.bin'):
    print("Upgrade complete, rebooting...")
    machine.reset()

# After reboot, in boot.py or main.py:
def validate_firmware():
    """Test critical functionality"""
    try:
        # Test your critical features
        import important_module
        important_module.test()
        return True
    except Exception as e:
```

(continues on next page)

(continued from previous page)

```
        print(f"Validation failed: {e}")
        return False

if validate_firmware():
    print("✓ Firmware validated")
    fuota.valid()  # Mark as valid
else:
    print("⍰ Validation failed, rolling back...")
    fuota.rollback()  # Reboot to previous version
```

Memory Usage

The new `fuota.upgrade()` function is implemented in C and uses the ESP-IDF's optimized HTTPS OTA stack:

- **No Python thread needed**: Runs directly in main thread
- **No 64KB stack allocation**: Uses default C stack
- **Efficient streaming**: Downloads and flashes in chunks
- **PSRAM available**: All 8MB PSRAM remains free for other uses

Old vs New Approach

Old (Python thread with urequests):

```
_thread.start_new_thread(download_thread, ())  # 64KB internal RAM!
```

- Required 64KB thread stack in internal RAM
- Used Python HTTP libraries (slower, more memory)
- Complex error handling

New (C implementation):

```
fuota.upgrade(url)  # No extra stack needed
```

- Uses default C stack
- ESP-IDF optimized HTTP/TLS stack
- Automatic error handling

Network Requirements

- Active WiFi or LTE connection
- DNS resolution for HTTPS URLs
- Sufficient bandwidth for firmware download
- Stable connection (30s - several minutes depending on size)

Security

- **HTTPS recommended**: Use certificate bundle or custom cert
- **Firmware validation**: Automatic integrity checking
- **Rollback protection**: Mark firmware valid after testing
- **Partition isolation**: Failed upgrades don't affect running firmware

Troubleshooting

"OTA upgrade already in progress"

- An OTA operation is already running
- Wait for it to complete or call `fuota.abort_upgrade()`

"Network not connected - please connect to WiFi or LTE first"

- No active network connection detected
- Connect to WiFi or LTE before starting OTA
- Verify IP address is assigned

"ESP HTTPS OTA Begin failed"

- Invalid URL or unreachable server
- Check URL is correct and server is running
- Test URL in browser first

"Image validation failed, image is corrupted"

- Downloaded firmware is corrupted or incomplete
- Check network stability during download
- Verify firmware binary is valid for your hardware

"Unable to read remote firmware descriptor"

- Server returned invalid firmware image
- Ensure URL points to actual firmware binary (not HTML page)
- Check firmware is built for ESP32-S3

Slow Downloads

- Increase HTTP buffer size (requires code modification)
- Current: 128 bytes for fast flash writes
- Trade-off: Larger buffers = faster download but slower flash writes

REPL Hangs During Blocking Mode

- Fixed in current version with `is_background_task` flag
- Ensure you're using latest firmware
- Use non-blocking mode if issues persist

Partition Shows Wrong Subtype After Downgrade

- Flashing new firmware updates partition table
- Subtype changes from `6` (IDF v4) to `131/0x83` (IDF v5)
- This is normal - indicates which firmware version is active
- **Important**: Downgrading from IDF v5 to IDF v4 requires `--erase-flash` due to LittleFS version incompatibility

Implementation Details

Architecture

- **C-based implementation**: Uses ESP-IDF `esp_https_ota` API
- **Two distinct APIs**:
  - `fuota.upgrade(url)`: **Blocking** - runs in calling thread, returns after completion
  - `fuota.start_upgrade(url)`: **Non-blocking** - runs in FreeRTOS task (priority 5, 8KB stack)
- **Automatic partition selection**: Toggles between ota_0/ota_1
- **Built-in certificate bundle**: Support for common CA certificates
- **Configurable HTTP timeout**: Default 30 seconds
- **Efficient streaming**: 128-byte HTTP buffers for fast flash writes

Progress Logging

- **Percentage-based**: Logs every 1% when total size known
- **Byte-based**: Logs every write when size unknown
- **Event-driven**: Uses ESP-IDF OTA event system
- **Debug output**: Comprehensive firmware metadata logging

Network Detection

Automatically checks connectivity before starting:

1. Checks WiFi (WIFI_STA_DEF interface)
2. Falls back to LTE (PPP_DEF interface)
3. Fails fast if no IP address available

Automatic Validation

New firmware is automatically validated on first boot:

- Checks if running from OTA partition
- Detects pending validation state
- Calls `esp_ota_mark_app_valid_cancel_rollback()` automatically
- No manual `fuota.valid()` call needed (unless you want extra validation)

Soft Reset Safety

When you press Ctrl+D (soft reset):

- Automatically cancels any running OTA operation
- Cleans up background task
- Prevents orphaned OTA tasks
- Safe to restart Python environment during OTA

Task Priority

Background OTA task runs at priority 5 to:

- Avoid preemption during flash writes
- Ensure stable flash operations
- Allow REPL to remain responsive
- Priority higher than MicroPython main task (priority 1)

Future Enhancements

- ☒ Non-blocking mode with status monitoring
- ☒ Progress tracking with percentage updates
- ☒ Automatic firmware validation on boot
- ☒ Soft reset safety (clean OTA cancellation)
- ☒ Network connectivity detection
- ☐ Resume interrupted downloads
- ☐ Delta/differential updates
- ☐ Custom progress callbacks
- ☐ Configurable HTTP buffer size via API
- ☐ Multi-stage bootloader support

See Also

- Example: `/examples/fuota/ota_example.py`
- ESP-IDF OTA docs: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/ota.html

9.3.9 SafeBoot Feature

Contents

- Introduction
- Entering Safeboot
- Safeboot Configurations
- Safeboot APIs

Introduction

SafeBoot feature introduces two modes of system boot:

- `Normal-mode` in which the system performs the normal start-up booting sequence and initialization scripts `boot.py` and `main.py` in case of the micropython build variant.
- `Safeboot-mode` in which the system performs only the basic system booting sequence such as hardware initialization and skips the normal application initialization and intialization scripts in case of micropython build variant.

Entering Safeboot

The system normally boots in `Normal-mode`, and the system offers two ways to trigger the `Safeboot-mode`:

- `software-reset`: the system can be triggered to reset in `safeboot-mode` by a software call to this function `bootif_safeboot_soft_reset()`. In `micropython` variant, there is a shortcut key `Ctrl + F` which triggers the same function call.
- `Hardware-Safeboot-Button`: the system also, offers another way using an external switch push button connected to one of the microcontroller boot strapping GPIOs. The system will detect if this button is pressed during reset and enters the `Safeboot-mode`.

  The boards which are have this button mounted on it, will activate this feature by default, but for the OEM modules which does not have any shield, it is disabled by default.

  For OEM modules, the customer will eventually integrate it on a special PCB for his own application project. In case the new project design needs to utilize this feature, then it could be activated from the SDK `menu-config` options and can be configgered to use another GPIO pin than the default one.

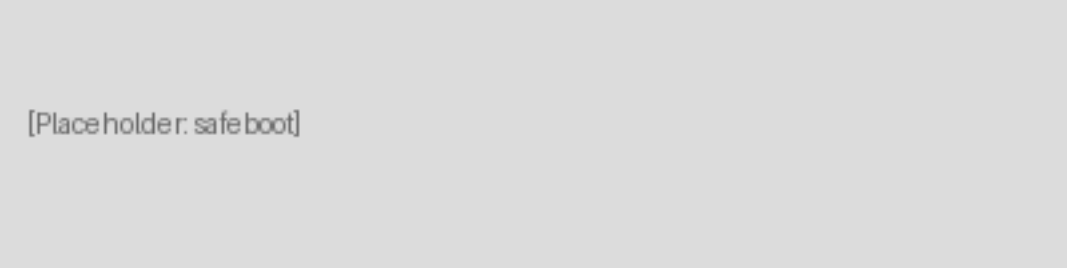
  In this triggerring mode, the system offers as well, some extra options depending on the switch press holding time. There are three different holding times:

  - `Safeboot in Latest Firmware Hold Time` (default 0 <= T < 3 sec): The system boots from the latest updated firmware image.
  - `Safeboot in previous OTA Firmware Hold Time` (default 3 <= T < 7 sec): The system boots from the previous OTA image if exist. Otherwise, boots from the latest.

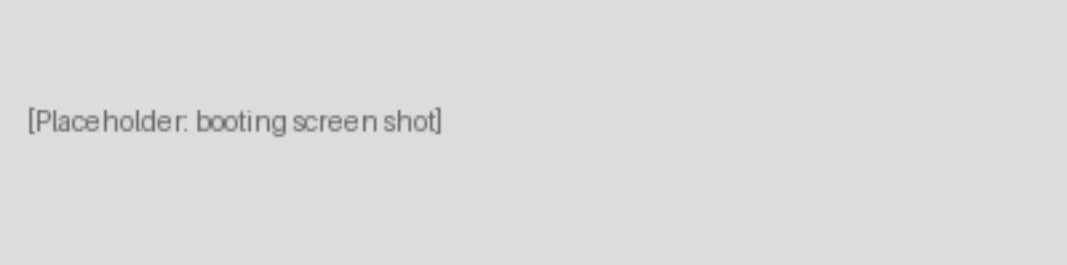
  - `Safeboot in Factory Firmware Hold Time` (default 7 <= T sec): The system boots from the factory firmware in safeboot mode.

All of those times are configurables.

The following sequence chart, shows the interaction between the bootloader and the running application in the context of the Safeboot actions;

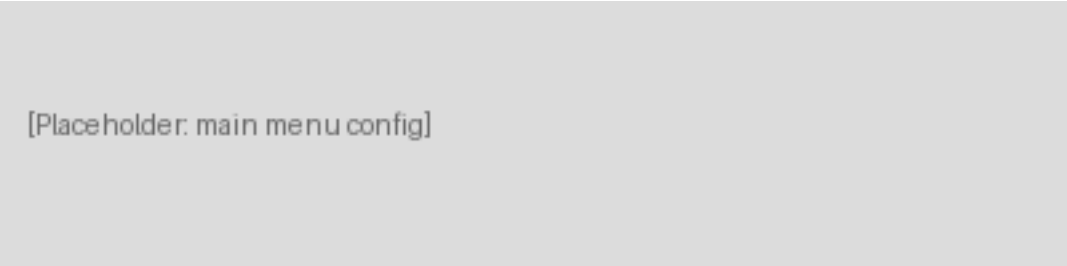


The following picture, shows the system `Normal-boot` after system reset, followed by another system reset in `Safeboot-mode` by pressing `Ctrl + F`.

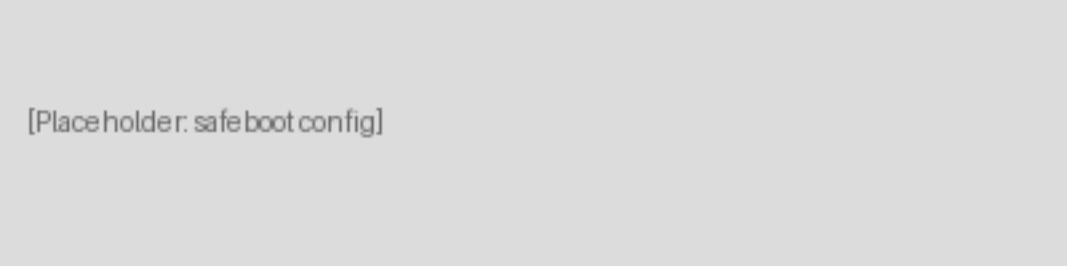


Safeboot Configurations

First trigger the build-system command `config` to enter the `menu-config` screen, for example: `./fw_builder.sh --board SGW3501-F1-EVB config`, then the build system will open a screen like this:



Then select `*** SDK Platform` and in the next screen, select `F1 Platform`, then `Safeboot Feature`, then you get this screen:



- Safeboot APIs

Safeboot APIs

The main interface of the safeboot functionalities are in `<SDK-ROOT>/src/platforms/F1/bootloader_components/boot-if/boot_if.h` which declare the following APIs:

- `bootif_state_set()` this function sets the current boot state of the system it is used as a ping-pong value between the bootloader and the firmware in the following way:

  If it set by the firmware, the boot loader will start the safeboot mode automatically.

  If it is set by the firmare, it means the safeboot button is pressed and according to the hold-time the intended image will be loaded.

  the responsibility to reset the state to normal, is for the firmware after reading it, every time for both triggering modes whether by software reset of by the hardware safeboot button.

- `bootif_state_set()` this function gets the current boot state of the system, it is usually called from the application to know whether the system boots in normal or safeboot to take the proper action.

- `bootif_safeboot_soft_reset()` this function hard resets the system in safeboot state. it is called from the application side to make a software safeboot.

- `bootif_safeboot_soft_reset_init()` init the software reset mechanism. it must be called in the system startup initialization sequence, so that the rest of safeboot API is accessible.

9.3.10 System Inspection

Contents

- Introduction
- Memory Dump
- Peripherals Power

Introduction

This component introduces some miscellaneous functions to help in system inspection and debugging.

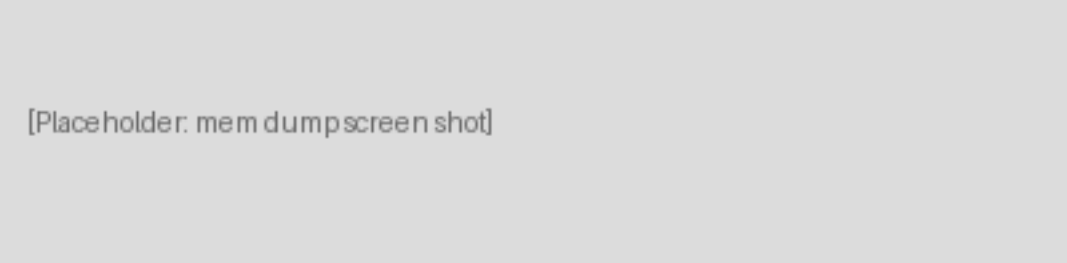
Memory Dump `sys_inspect.dump_memory()`

It is a utility function to dump a certain memory space.

It takes the following keyword argument:

- `address` the start address of the dump memory space
- `size` size of data to dumpped
- `wordsize` display word size of dumped data (32, 16, or 8)
- `linesize` number of display bytes per line
- `disp_text` show the printable ascii character of the data.

Example: the following picture shows an example of the memory dump

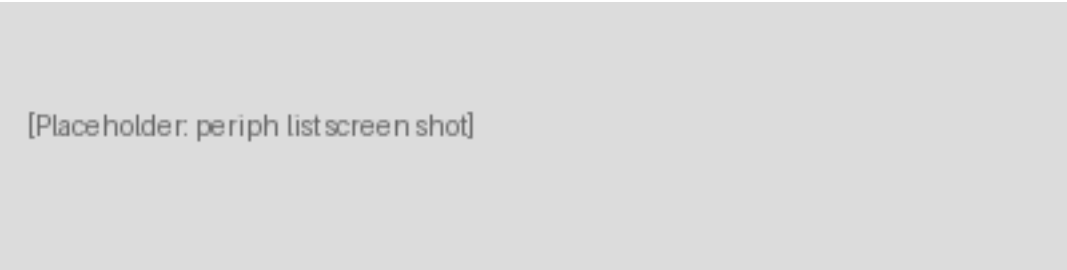


Peripherals Power

It is a set of utilities that help in inspection of the enabled system peripherals. That could help in power management inspection. The current available methods are:

- `sys_inspect.periph_module_list()` It lists the whole list of system peripherals and indicate the peripheral along with how many other system actors are using it. The periphiral in use indicate that its power is on.
- `sys_inspect.periph_power(<periph>, True|False)` It tries to disabled the module compulsory, in this case, the system may perform in undesired behaviour

The following is a screen shot to the `periph_module_list()`



CHAPTER

TEN

MICROPYTHON LIBRARIES

The SG Wireless firmware is built on **MicroPython** and includes its full standard library. The upstream documentation applies directly—use the cross-references below to find detailed API descriptions.

10.1 Standard Libraries

These modules ship with every MicroPython port and behave identically on SG Wireless hardware:

- builtins
- array
- binascii
- collections
- errno
- gc
- hashlib
- io
- json
- math
- os
- random
- re
- select
- socket
- ssl
- struct
- sys
- time
- uasyncio

10.2 Hardware & System Libraries

- machine
- machine.Pin
- machine.I2C
- machine.SPI
- machine.UART
- machine.ADC
- machine.Timer
- machine.RTC
- machine.WDT
- micropython
- network
- bluetooth
- esp32

10.3 SG Wireless Extensions

The following modules are **SG Wireless specific** and are documented in the *API Reference* section:

- *LoRa API Documentation* — LoRa WAN and RAW radio
- *lte Module - C Implementation* — LTE Cat-M1 / NB-IoT
- *CTRL API Documentation* — Ctrl Cloud client
- *Non-Volatile Storage (NVS) Interface* — Non-volatile storage
- *System Information* — Board and firmware info
- *IO Expander Interface (PCAL6408A)* — GPIO expander
- *Fuel-Gauge (BQ27421)* — Battery fuel gauge
- *CAN API Documentation* — CAN bus

- *eFuse Interface* — eFuse identity data
- *FUOTA Module - Firmware Update Over The Air* — Firmware OTA updates
- *SafeBoot Feature* — SafeBoot feature
- *System Inspection* — Memory and peripheral inspection
- *RGB-LED (WS2812B)* — RGB LED

10.4 Further Reading

- MicroPython documentation
- MicroPython quick reference for ESP32

CHAPTER

ELEVEN

TUTORIALS & EXAMPLES

This section contains tutorials for using MicroPython APIs: connecting to cellular networks, WiFi, Bluetooth, controlling I/O pins and setting up LoRa nodes.

11.1 Basic Tutorials

11.1.1 REPL

F1 smart module has pre-installed MicroPython as its operating system (OS), which includes a REPL. REPL stands for Read-Eval-Print Loop, an interactive interpreter mode that allows you to input code, execute it, and immediately see the results.

Using the CtrlR Plugin, open and connect a device or use a serial terminal (PuTTY, screen, picocom, etc). Upon connecting, there should be a blank screen with a flashing cursor. Press Enter and a MicroPython prompt should appear, i.e. >>>. Let's make sure it is working with the obligatory test:

```
>>> print("Hello F1!")
Hello F1!
```

Note

The >>> characters should not be typed. They indicate the prompt. Once the text `print("Hello F1!")` has been entered and `Enter` pressed, the output should appear on screen. Basic Python commands can be tested out in a similar fashion.

If this is not working, try either a hard reset or a soft reset; see below.

Resetting the Device

If something goes wrong, the device can be reset with two methods: hard reset and soft reboot.

Hard reset

By pressing the RESET button on the F1 Starter Kit (or applying a high signal to the F1 module reset signal), a reset signal is triggered to the RESET pin of the F1 module.

Please notice that any serial/COM port connection will reset and may need to be manually reconnected if the auto-connect option is not enabled in the CtrlR plug-in.

After reset, the normal MicroPython boot message will appear in the terminal:

```
>>> ESP-ROM:esp32s3-20210327
Build:Mar 27 2021
rst:0x1 (POWERON),boot:0x8 (SPI_FAST_FLASH_BOOT)
SPIWP:0xee
mode:DIO, clock div:2
load:0x3fce3830,len:0xfac
load:0x403c9700,len:0xd3c
load:0x403cc700,len:0x2dd8
entry 0x403c9964
== micropython normal mode
-- start ctrl client
MicroPython v1.19.1-796-gf4811b0b4 on 2024-05-02; SGWireless SGW3501-F1-StarterKi
t with ESP32S3
Type "help()" for more information.
>>>
```

Soft reboot

By pressing `Ctrl+D` at the MicroPython prompt, a soft reset is performed. The soft reboot message will appear in the terminal:

```
MPY: soft reboot
== micropython normal mode
-- start ctrl client
MicroPython v1.19.1-796-gf4811b0b4 on 2024-05-02; SGWireless SGW3501-F1-StarterKi
t with ESP32S3
Type "help()" for more information.
>>>
```

REPL control commands

In MicroPython, there are several control commands available in the REPL:

- `Ctrl+A` — on a blank line, enter raw REPL mode
- `Ctrl+B` — on a blank line, enter normal REPL mode
- `Ctrl+C` — interrupt a running program
- `Ctrl+D` — on a blank line, do a soft reset of the board
- `Ctrl+E` — on a blank line, enter paste mode
- `Ctrl+F` — do hard reset in safeboot mode

Boot modes

There are two boot modes in MicroPython: normal boot mode and safe boot mode.

Normal boot mode

MicroPython searches for and runs `boot.py` during startup, then runs `main.py` after `boot.py` if those files exist in the file system under the `/` directory.

If those files are absent, MicroPython will skip and continue booting.

Safe boot mode

In safe boot mode, MicroPython will intentionally bypass searching and running `boot.py` and `main.py`. This is a safeguard to boot up MicroPython in case of a lock-up caused by `boot.py` or `main.py`.

11.1.2 Sleep

There are several methods to make your device sleep. First we cover the basic sleep. Similar to `delay()` used in Arduino, sleep will yield your program until the time is over. Important is that all microcontroller functions keep running. Also the LoRa and LTE modems can be used directly (without re-attaching) after regular sleep.

Basic sleep

```
import time

time.sleep(1)           # sleep 1 second
time.sleep_ms(10)       # sleep 10 milliseconds
time.sleep_us(10)       # sleep 10 microseconds
```

Similar to `yield()` in other languages, in MicroPython we use:

```
import machine
machine.idle()
```

Power saving

To save power, we can also put the controller into sleep modes using the following examples.

Light sleep

The `machine.sleep()` command will put the controller into a light sleep mode. WiFi and BLE are switched off, but the main CPU and RAM are still running. The LoRa and LTE modems are stopped as well and have to be re-initialized after wakeup. The controller will continue running the code after waking up. GPIO states are also conserved. Setting the second argument to `True` will restore the WiFi and BLE after wakeup.

```
import machine
import time
print("this will be printed before: " + str(time.ticks_ms()))
machine.sleep(1000 * 10)
print("this will be printed after 10 seconds: " + str(time.ticks_ms()))
```

Deep sleep

Deepsleep disables, next to the lightsleep, the main CPU and RAM. This leaves only a low power coprocessor and RTC timer running. After waking up, the board will start again at `boot.py`, just like with pressing the reset button. The CPU counter (`time.ticks()`) will continue to count however!

You can also leave the brackets empty to sleep indefinitely, until the reset button is pressed, the power is removed, or an external wake up signal (interrupt) is provided. Be aware that the LTE modem will remain switched on unless you actively switch off its power, or use its own power saving modes.

```
import machine
print("Wake up")
machine.deepsleep(1000)  # deepsleep 1 second
print("this will never get printed!")
```

Wake up reason

Sometimes, we want to know the reason the board woke up, to differentiate between pressing the reset button and other ways of waking up from sleep:

```
import machine
import time

wake_reason = machine.wake_reason()
print("Device running for: " + str(time.ticks_ms()) + "ms")

if wake_reason == machine.PWRON_RESET:
    print("Woke up by reset button")
elif wake_reason == machine.PIN_WAKE:
    print("Woke up by external pin (external interrupt)")
elif wake_reason == machine.ULP_WAKE:
    print("Woke up by ULP (capacitive touch)")
elif wake_reason == machine.TOUCHPAD_WAKE:
    print("Woke up by touchpad")

machine.deepsleep(1000 * 60)  # sleep for 1 minute
print("This will never be printed")
```

Note

Using `deepsleep()` will also stop the USB connection. Be wary of that when trying to upload new code to the device!

11.1.3 Print

Using `print()` statements in your Python script is quite easy. But did you know you can also concatenate strings and variables inline? If you are familiar with C, its functionality is similar to the `printf()` function, but with `\n` always included.

```
import machine

print("hello world")
print("hello world.", end='')  # do not end line

# you can also specify different endings, like additional text, tabs
# or any other escape characters
for i in range(0, 9):
    print(".", end='')
print("\n")  # feed a new line
```

(continues on next page)

(continued from previous page)

```
print("\t tabbed in")

# you can specify a variable into the string as well!
print("hello world: " + str(machine.rng()) + " random number")

# or use format
print("hello world: {} {}".format(machine.rng(), " random number"))

# you can also ask for user input
result = input("what's up?\n")
print(result)

# and lastly, you can also print like this, which is very useful
# when printing large amounts of data
i = 10
print(1, 2, 3, 'e', i)
```

11.1.4 RGB LED

By default the heartbeat LED flashes in blue colour once every 4s to signal that the system is alive. This can be overridden through the F1 Starter Kit command.

`initialization`: the module will be initialized once it is imported. After its initialization, it can be deinitialized by calling `rgbled.deinit()` and to initialize it again use `rgbled.initialize()`.

rgbled.color()

`rgbled.color()` sets the LED color continuously. The color follows the hex formatting `xxRRGGBB`, in which `RR`, `GG` and `BB` represent the red, green and blue components of the color respectively and `xx` is a don't care value.

```
import rgbled

rgbled.heartbeat(False)       # stop the heartbeat service
rgbled.color(0x00FF0000)      # sets the LED color to red
rgbled.color(0x0000FF00)      # sets the LED color to green
rgbled.color(0x000000FF)      # sets the LED color to blue
rgbled.color(0x00FFFF00)      # sets the LED color to yellow
```

rgbled.heartbeat()

`rgbled.heartbeat()` starts the heartbeat blinking service. It has three signatures:

- `rgbled.heartbeat()` — check the current status of the heartbeat service, returns `True` or `False`.
- `rgbled.heartbeat(<enable>)` — enable or disable the service.
- `rgbled.heartbeat(<color>, <cycle-time>, <blink-percentage>)` — set new configuration for the service and start or restart it.

Arguments:

- `<enable>` — boolean value to enable or disable the service.
- `<color>` — color value, same as `rgbled.color()`.
- `<cycle-time>` — total period of the duty-cycle (light on + light off).
- `<blink-percentage>` — percentage (0 < p < 100) where the light is on.

```
import rgbled

rgbled.heartbeat()                          # check status → False
rgbled.heartbeat(True)                      # start with default configs
rgbled.heartbeat()                          # check status → True

# blue color blinking for ~200 ms each second
rgbled.heartbeat(0x000000FF, 1000, 20)

rgbled.heartbeat(False)                     # stop the heartbeat service

# red color blinking for ~10 ms each 50 ms (very fast)
rgbled.heartbeat(0x00FF0000, 50, 20)

rgbled.heartbeat(True)                      # restart with latest config
```

rgbled.decoration()

`rgbled.decoration()` provides a fancy way of doing decorative light blinking by specifying a sequence of blinking descriptors.

Syntax:

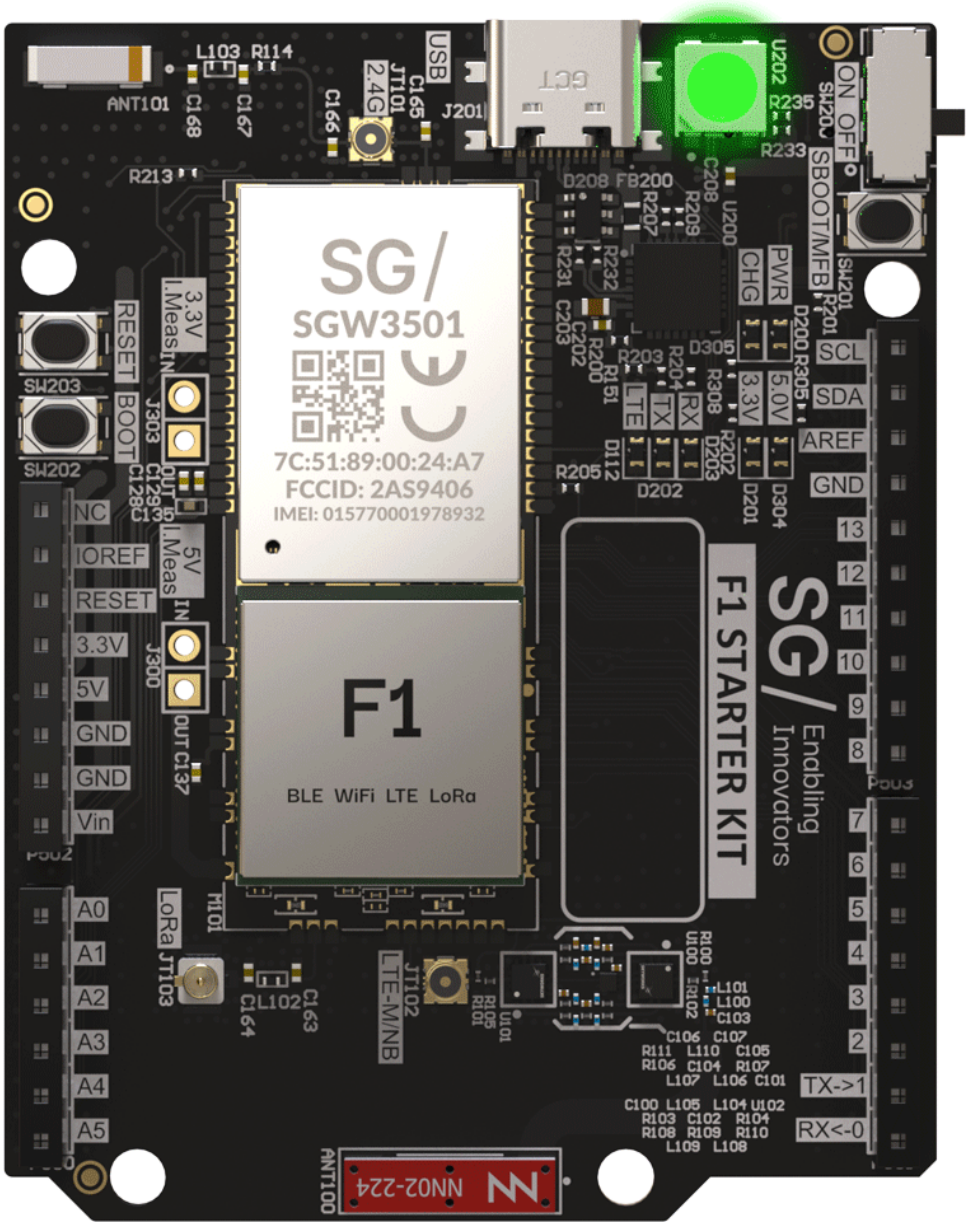
`rgbled.decoration(<blink-desc-list>, <repeat>)`

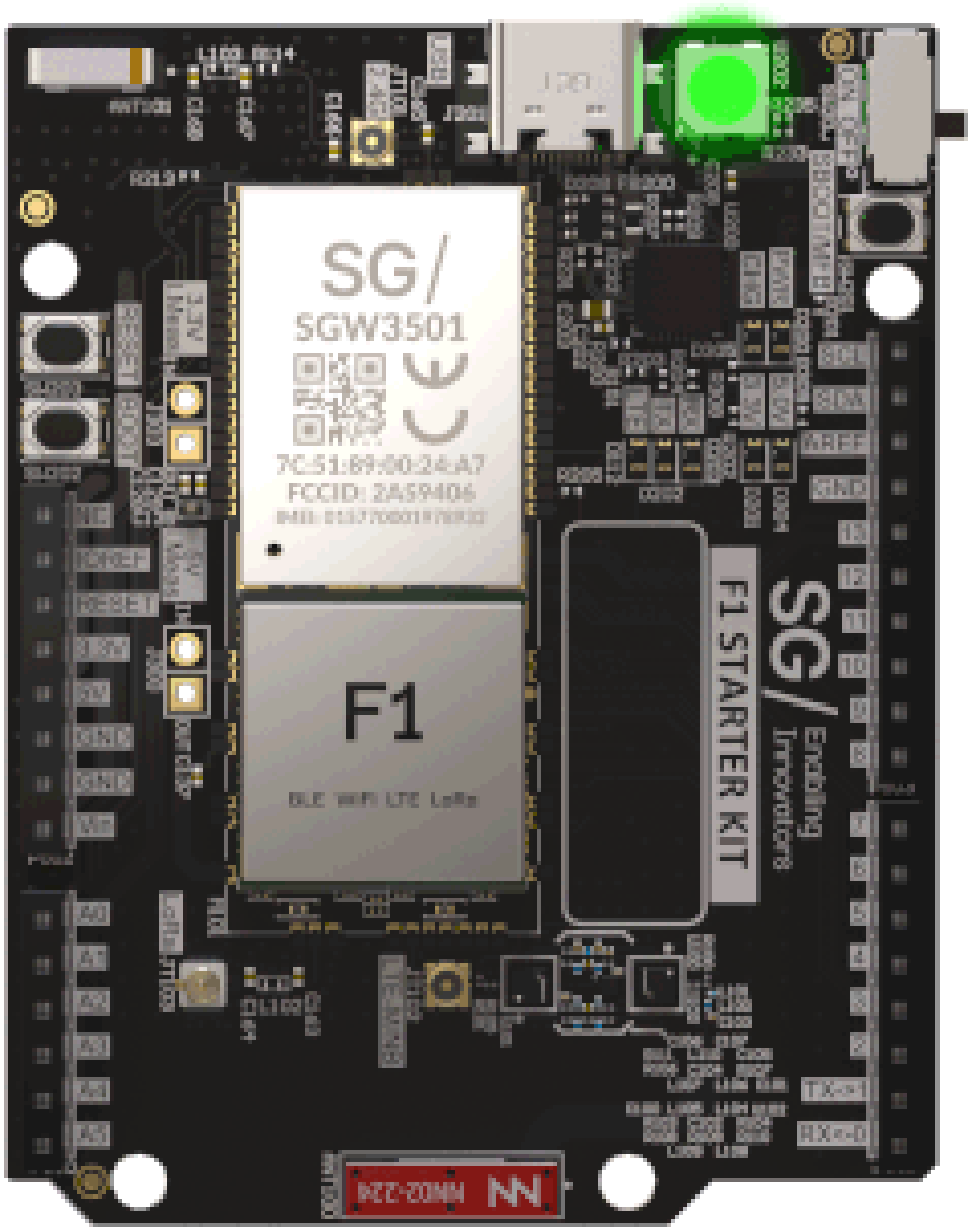
Where:

- `<blink-desc-list>` — list of four-element tuples describing a time window of blinking.
- Each tuple: `(<color-value>, <duty-period>, <light-on-percent>, <loop-count>)`

```
#  ___ 50 ___          ___ 50 ___           __________ __________
# | G |__| G |________| B |__| B |__________|    R     |     Y    |
# |--------2 Sec-----|--------2 Sec-------|- 0.5 sec-|- 0.5 sec-|

import rgbled

rgbled.decoration([
    (0x00001100, 100, 50, 2),     # two green pulses
    (0, 2000 - 200, 0, 1),        # light off between G and B
    (0x00000011, 100, 50, 2),     # two blue pulses
    (0, 2000 - 200, 0, 1),        # light off after B pulses
    (0x00110000, 500, 100, 1),    # red period
    (0x00111100, 500, 100, 1),    # yellow period
    (0, 1000, 0, 1)               # light off before repeating
], True)                          # repeat the whole sequence
```

Here is the expected result:





rgbled._color

`rgbled._color` is a class carrying the basic color definitions. It can be used directly in place of the color value:

```
import rgbled

rgbled.color(rgbled._color.RED)
rgbled.color(rgbled._color.GREEN)
rgbled.color(rgbled._color.BLUE)
rgbled.color(rgbled._color.YELLOW)
rgbled.color(rgbled._color.MAGENTA)
rgbled.color(rgbled._color.CYAN)
rgbled.color(rgbled._color.WHITE)
```

11.1.5 GPIO

The F1 smart module has more than twenty spare General Purpose Input-Output (GPIO) pins available for you to use with your own sensors and actuators. Below outlines the basics for configuring and controlling the GPIO as output and input pins.

Output

Controlling the GPIO pins of the module is straightforward. The example below shows how to generate a 1 Hz clock signal at Pin 9.

```
from machine import Pin
import time

Pin9 = Pin(Pin.P9, mode=Pin.OUT)

while True:
    print("high")
    Pin9.value(1)
    time.sleep(1)
    print("low")
    Pin9.value(0)
    time.sleep(1)
```

Input

Sometimes, it is useful to know the state of a pin. For example, you could use the SBOOT button on the Starter Kit to toggle the LED.

```
from machine import Pin
import rgbled

button = Pin(Pin.P12, mode=Pin.IN)  # SBOOT button on Starter Kit

while True:
    if button() == 1:
        rgbled.color(0x00003300)  # turn LED to Green
    else:
        rgbled.color(0x00000000)  # turn off LED
```

11.2 Hardware Tutorials

These tutorials cover interfacing with hardware peripherals on the F1 module.

11.2.1 ADC

This example is a simple ADC sample. In this example F1 Starter Kit pin A1 (i.e. F1 smart module pin P17) is set to be used for ADC.

```
from machine import ADC
from machine import Pin
```

(continues on next page)

(continued from previous page)

```
adc = ADC(Pin(Pin.P17))
adc.read_u16()  # read the ADC analog raw value mapped to 0-65535
adc.read_uv()   # read the ADC analog value in microvolt
```

11.2.2 I2C

The following example receives data from a light sensor using I2C. The sensor used is the BH1750FVI Digital Light Sensor.

> **Note**
>
> For F1 module and F1 Starter Kit, `I2C(0)` is reserved for internal use. Please start from `I2C(1)` for any new I2C bus.

```
import time
from machine import I2C
import bh1750fvi

i2c = I2C(1, freq=100000)
# Remark: I2C(0) is reserved in F1 as internal I2C bus
light_sensor = bh1750fvi.BH1750FVI(i2c, addr=i2c.scan()[0])

while True:
    data = light_sensor.read()
    print(data)
    time.sleep(1)
```

Drivers for the BH1750FVI

Place this code into a file named `bh1750fvi.py`. This can then be imported as a library.

```
# Simple driver for the BH1750FVI digital light sensor

class BH1750FVI:
    MEASUREMENT_TIME = const(120)

    def __init__(self, i2c, addr=0x23, period=150):
        self.i2c = i2c
        self.period = period
        self.addr = addr
        self.time = 0
        self.value = 0
        self.i2c.writeto(addr, bytes([0x10]))  # start continuous 1 Lux
↪readings

    def read(self):
        self.time += self.period
```

(continues on next page)

(continued from previous page)

```
        if self.time >= MEASUREMENT_TIME:
            self.time = 0
            data = self.i2c.readfrom(self.addr, 2)
            self.value = (((data[0] << 8) + data[1]) * 1200) // 1000
        return self.value
```

11.2.3 Timers

Detailed information about this class can be found in the `Timer` API reference.

Timer

The following example showcases a simple stopwatch application.

```
import time

initial_time = time.time()   # time.time() gets the system time tick
time.sleep(3)                # simulate 3 seconds time off
end_time = time.time()
difference = end_time - initial_time

print("\nthe time difference [in second] is:", difference)
```

11.3 Network Tutorials

These tutorials cover wireless network connectivity on the F1 module.

11.3.1 WiFi

The WLAN (WiFi) is a system feature of all SG devices, therefore it is enabled by default. The development boards include an on-board antenna by default, so no external antenna is needed to get started. When deploying your solution, you might want to consider using the external antenna to increase the wireless range.

On this page, we cover:

- Connected by devices (act as an AP)
- Connecting to a router (act as a Device)
- Using an external antenna

> **Note**
>
> Generally, code in either section is applicable to both WLAN modes.

Connected by Devices (Act as an AP)

Using the WLAN class from `network`, you can change the name (SSID) and security settings (auth) of the access point.

```
import network

ap = network.WLAN(network.AP_IF)          # create access-point interface
ap.config(essid='ESP-AP')                 # set the ESSID of the access point
ap.config(password='micropython')
ap.config(authmode=network.AUTH_WPA2_PSK)
ap.config(max_clients=10)                 # set how many clients can connect
ap.active(True)                           # activate the interface
```

The device will not be able to access the internet, but you will be able to run a simple webserver. By default, the IP address will be configured to `192.168.4.1`.

Connecting to a Router (Act as a Device)

To connect to an existing network, the WiFi class must be configured as a station:

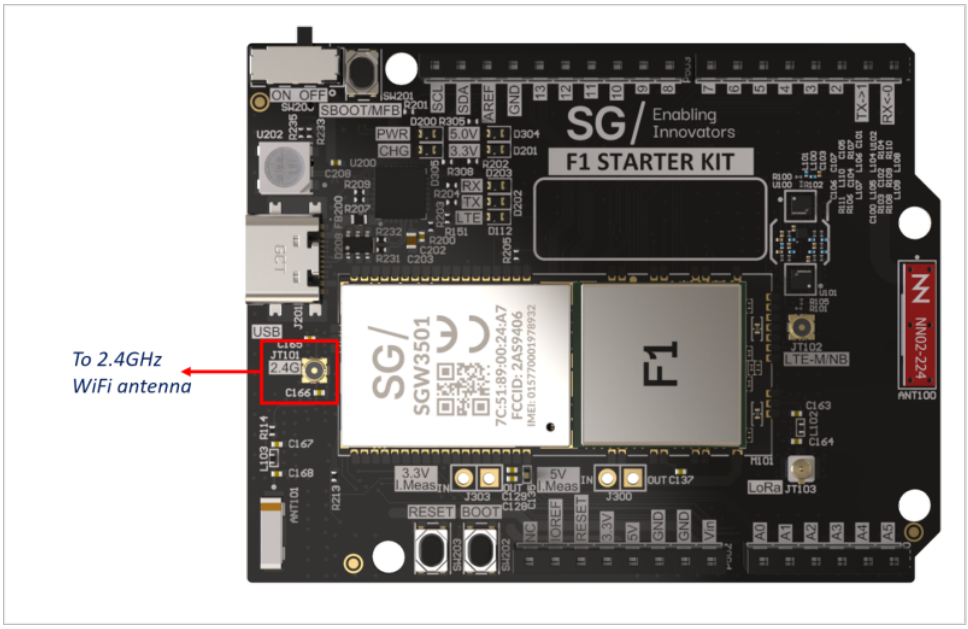
```
import network

wlan = network.WLAN(network.STA_IF)  # create station interface
wlan.active(True)                    # activate the interface
wlan.scan()                          # scan for access points
wlan.isconnected()                   # check if connected to an AP
wlan.connect('essid', 'password')    # connect to an AP
wlan.config('mac')                   # get the interface's MAC address
wlan.ifconfig()                      # get IP/netmask/gw/DNS addresses
```

Note

For more details on the WLAN class, please refer to the MicroPython WLAN documentation.

Using an external antenna

Connect a WiFi antenna to the microwave SWF type switch connector on your development board in order to use an external antenna for WiFi.



The microwave SWF type switch connector is a hardware connector with switch. By default, it is connected to the on-board Wi-Fi chip antenna. If an external antenna is connected to the SWF type switch connector, it will switch to the external antenna and is isolated from the default on-board Wi-Fi chip antenna.

Note

Since BLE and Wi-Fi share the same RF path, if an external antenna is connected both Wi-Fi and BLE signals will go through the external antenna.

11.3.2 BLE

The BLE (Bluetooth Low Energy) is a system feature of the SG module. There are many applications including iBeacon, sensors, smartwatch, etc.

For more information, please refer to the MicroPython bluetooth class documentation.

You may also want to look at the aioble library which provides a higher-level API for BLE operations.

11.3.3 LoRa Examples

The following tutorials demonstrate the use of the LoRa functionality. LoRa can work in 2 different modes: LoRa-MAC (which we also call Raw-LoRa) and LoRaWAN mode.

When using LoRa, always connect the appropriate LoRa antenna to your device. See the figure below for the correct antenna placement.

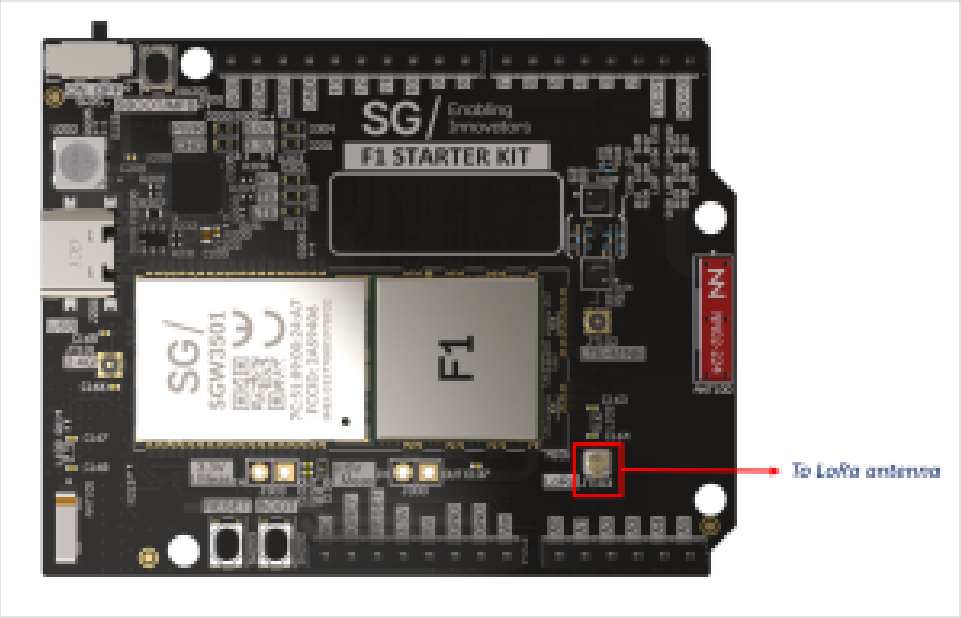


Fig. 1: F1 Starter Kit

Note

A standalone pluggable LoRa antenna in u.FL connector type is included in the F1 Starter Kit.

- **LoRaWAN mode** implements the full LoRaWAN stack for a class A device. It supports both OTAA and ABP connection methods, as well as advanced features like adding and removing custom channels to support "special" frequency plans like those used in New Zealand. There are two basic ways of accessing the LoRaWAN network:
  - *LoRaWAN with OTAA*
  - *LoRaWAN with ABP*

Note

When using LoRaWAN, first register with one of the networks.

- **LoRa-MAC mode** basically accesses the radio directly and packets are sent using the LoRa modulation on the selected frequency without any headers, addressing information or encryption. Only a CRC is added at the tail of the packet and this is removed before the received frame is passed on to the application. This mode can be used to build any higher level protocol that can benefit from the long range features of the LoRa modulation.
  - *LoRa-MAC (Raw LoRa)*

LoRaWAN with OTAA

OTAA stands for Over The Air Authentication. With this method the device sends a Join request to the LoRaWAN Gateway using the `app_eui` and `app_key` provided in your LoRaWAN application (like TheThingsNetwork, Chirpstack etc.). If the keys are correct the Gateway will reply with a join accept message and from that point on the device is able to send and receive packets to/from the Gateway. If the keys are incorrect no response will be received and the `has_joined()` method will always return `False`.

The example below attempts to get any data received after sending the frame. Keep in mind that the Gateway might not be sending any data back, therefore we make the socket non-blocking before attempting to receive, in order to prevent getting stuck waiting for a packet that will never arrive.

If everything works correctly, the device will print `Joined` to the terminal, and you should see a data packet arrive in your LoRaWAN application containing `0x01 0x02 0x03`.

Note

If the `dev_eui` is not provided, the LoRa MAC of the device will be used in its place. You will need to change the `dev_eui` in your LoRaWAN application to the LoRa MAC address of the device. You can get the LoRa MAC using:

```
from network import LoRa
import binascii
print(binascii.hexlify(LoRa().mac()).upper())
```

Note

US915 / AU915 regions: Most LoRaWAN gateways are configured to listen to 8 channels only, while the region supports up to 64 uplink channels. In order to receive packets, please confirm the frequency plan of your gateway with the channels configured on your device. By default, our devices will transmit on all 64 channels, meaning you might receive packets intermittently. The most common configuration is `FSB2`, or channels 8-15. Uncomment the respective section in the example below to select these uplink channels.

For LoRaWAN v1.0.x:

```
# ------------------------------------------------------------------------
↪-- #
# Copyright (c) 2023-2024 SG Wireless - All Rights Reserved
#
# Permission is hereby granted, free of charge, to any person obtaining a␣
↪copy
# of this software and associated documentation files(the "Software"), to␣
↪deal
# in the Software without restriction, including without limitation the␣
↪rights
# to use,  copy,  modify,  merge, publish, distribute, sublicense, and/or␣
↪sell
# copies  of  the  Software,  and  to  permit  persons to whom the Software␣
↪is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS",  WITHOUT WARRANTY OF ANY KIND,  EXPRESS␣
↪OR
```

(continues on next page)

(continued from previous page)

```
# IMPLIED,  INCLUDING BUT NOT LIMITED TO  THE  WARRANTIES  OF
↪MERCHANTABILITY
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL
↪THE
# AUTHORS  OR  COPYRIGHT  HOLDERS  BE  LIABLE FOR ANY CLAIM,  DAMAGES OR
↪OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
↪FROM,
# OUT OF OR IN  CONNECTION WITH  THE SOFTWARE OR  THE USE OR OTHER DEALINGS
↪IN
# THE SOFTWARE.
#
# Desc      Implements simple lora OTAA activation.
# ------------------------------------------------------------------------
↪-- #

# required imports
import logs          # for system logging management
import lora          # for lora-stack
import ubinascii     # for hex/string conversions
import time          # for time manipulation

# disable fw logs
logs.filter_subsystem('lora', False)

# switch to lora-wan if not
lora.mode(lora._mode.WAN)

# configure the stack on region EU-868
lora.wan_params(region=lora._region.REGION_EU868, lwclass=lora._class.CLASS_
↪A)

# start end-device commissioning
lora.commission(
    type    = lora._commission.OTAA,
    version = lora._version.VERSION_1_0_X,
    DevEUI  = ubinascii.unhexlify('0000000000000000'),
    JoinEUI = ubinascii.unhexlify('0000000000000000'),
    AppKey  = ubinascii.unhexlify('00000000000000000000000000000000')
    )

# start join procedure
lora.join()

# wait until join
while lora.is_joined() == False:
    print("wait joining ...")
    time.sleep(2)
    pass
```

(continues on next page)

(continued from previous page)

```
print("-- JOINED --")
lora.stats()

# open a working port
lora.port_open(1)

# attach required callback
def get_event_str(event):
    if event == lora._event.EVENT_TX_CONFIRM:
        return 'EVENT_TX_CONFIRM'
    elif event == lora._event.EVENT_TX_DONE:
        return 'EVENT_TX_DONE'
    elif event == lora._event.EVENT_TX_TIMEOUT:
        return 'EVENT_TX_TIMEOUT'
    elif event == lora._event.EVENT_TX_FAILED:
        return 'EVENT_TX_FAILED'
    elif event == lora._event.EVENT_RX_DONE:
        return 'EVENT_RX_DONE'
    elif event == lora._event.EVENT_RX_TIMEOUT:
        return 'EVENT_RX_TIMEOUT'
    elif event == lora._event.EVENT_RX_FAIL:
        return 'EVENT_RX_FAIL'
    else:
        return 'UNKNOWN'

def port_any_cb(event, evt_data):
    print('lora event [ {} ] --> data: {}'
        .format(get_event_str(event), evt_data))

lora.callback(handler=port_any_cb)

# start duty cycle
lora.duty_set(10000)
lora.enable_rx_listening()
lora.duty_start()

# schedule sending some test messages
i = 1000
while i < 1010:
    lora.send('tx-message-with-id-{}'.format(i), port=1, confirm=True, id =␣
↪i)
    i = i + 1

# --- end of file ------------------------------------------------------
↪-- #
```

For LoRaWAN v1.1.x:

```
# ------------------------------------------------------------------------
↪-- #
```

(continues on next page)

(continued from previous page)

```
# Copyright (c) 2023-2024 SG Wireless - All Rights Reserved
#
# Permission is hereby granted, free of charge, to any person obtaining a␣
↪copy
# of this software and associated documentation files(the "Software"), to␣
↪deal
# in the Software without restriction, including without limitation the␣
↪rights
# to use,  copy,  modify,  merge, publish, distribute, sublicense, and/or␣
↪sell
# copies  of  the  Software,  and  to  permit  persons to whom the Software␣
↪is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS",  WITHOUT WARRANTY OF ANY KIND,  EXPRESS␣
↪OR
# IMPLIED,  INCLUDING BUT NOT LIMITED TO  THE  WARRANTIES  OF ␣
↪MERCHANTABILITY
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL␣
↪THE
# AUTHORS  OR  COPYRIGHT  HOLDERS  BE  LIABLE FOR ANY CLAIM,  DAMAGES OR␣
↪OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING␣
↪FROM,
# OUT OF OR IN  CONNECTION WITH  THE SOFTWARE OR  THE USE OR OTHER DEALINGS␣
↪IN
# THE SOFTWARE.
#
# Desc      Implements simple lora OTAA activation.
# ------------------------------------------------------------------------
↪-- #

# required imports
import logs          # for system logging management
import lora          # for lora-stack
import ubinascii     # for hex/string conversions
import time          # for time manipulation

# disable fw logs
logs.filter_subsystem('lora', False)

# switch to lora-wan if not
lora.mode(lora._mode.WAN)

# configure the stack on region EU-868
lora.wan_params(region=lora._region.REGION_EU868, lwclass=lora._class.CLASS_
```

(continues on next page)

(continued from previous page)

```
↪A)

# start end-device commissioning
# for DevEUI, AppKey and NwkKey, please replace zeros with your specific
# device information
lora.commission(
    type    = lora._commission.OTAA,
    version = lora._version.VERSION_1_1_X,
    DevEUI  = ubinascii.unhexlify('0000000000000000'),
    JoinEUI = ubinascii.unhexlify('0000000000000000'),
    AppKey  = ubinascii.unhexlify('00000000000000000000000000000000'),
    NwkKey  = ubinascii.unhexlify('00000000000000000000000000000000')
    )

# start join procedure
lora.join()

# wait until join
while lora.is_joined() == False:
    print("wait joining ...")
    time.sleep(2)
    pass
print("-- JOINED --")
lora.stats()

# open a working port
lora.port_open(1)

# attach required callback
def get_event_str(event):
    if event == lora._event.EVENT_TX_CONFIRM:
        return 'EVENT_TX_CONFIRM'
    elif event == lora._event.EVENT_TX_DONE:
        return 'EVENT_TX_DONE'
    elif event == lora._event.EVENT_TX_TIMEOUT:
        return 'EVENT_TX_TIMEOUT'
    elif event == lora._event.EVENT_TX_FAILED:
        return 'EVENT_TX_FAILED'
    elif event == lora._event.EVENT_RX_DONE:
        return 'EVENT_RX_DONE'
    elif event == lora._event.EVENT_RX_TIMEOUT:
        return 'EVENT_RX_TIMEOUT'
    elif event == lora._event.EVENT_RX_FAIL:
        return 'EVENT_RX_FAIL'
    else:
        return 'UNKNOWN'

def port_any_cb(event, evt_data):
    print('lora event [ {} ] --> data: {}'
```

(continues on next page)

(continued from previous page)

```
        .format(get_event_str(event), evt_data))

lora.callback(handler=port_any_cb)

# start duty cycle
lora.duty_set(10000)
lora.enable_rx_listening()
lora.duty_start()

# schedule sending some test messages
i = 1000
while i < 1010:
    lora.send('tx-message-with-id-{}'.format(i), port=1, confirm=True, id =␣
↪i)
    i = i + 1

# --- end of file ------------------------------------------------------------
↪-- #
```

LoRaWAN with ABP

ABP stands for Authentication By Personalization. It means that the encryption keys are configured manually on the device and can start sending frames to the Gateway without needing a 'handshake' procedure to exchange the keys (such as the one performed during an OTAA join procedure).

The example below attempts to get any data received after sending the frame. Keep in mind that the Gateway might not be sending any data back, therefore we make the socket non-blocking before attempting to receive, in order to prevent getting stuck waiting for a packet that will never arrive.

> **Note**
>
> **US915 / AU915 regions:** Most LoRaWAN gateways are configured to listen to 8 channels only, while the region supports up to 64 uplink channels. In order to receive packets, please confirm the frequency plan of your gateway with the channels configured on your device. By default, our devices will transmit on all 64 channels, meaning you might receive packets intermittently. The most common configuration is `FSB2`, or channels 8-15. Uncomment the respective section in the example below to select these uplink channels.

For LoRaWAN v1.0.x:

```
# ------------------------------------------------------------------------
↪-- #
# Copyright (c) 2023-2024 SG Wireless - All Rights Reserved
#
# Permission is hereby granted, free of charge, to any person obtaining a␣
↪copy
# of this software and associated documentation files(the "Software"), to␣
↪deal
# in the Software without restriction, including without limitation the␣
```

(continues on next page)

(continued from previous page)

```
↪rights
# to use,  copy,  modify,  merge, publish, distribute, sublicense, and/or␣
↪sell
# copies  of  the  Software,  and  to  permit  persons to whom the Software␣
↪is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS",  WITHOUT WARRANTY OF ANY KIND,  EXPRESS␣
↪OR
# IMPLIED,  INCLUDING BUT NOT LIMITED TO  THE  WARRANTIES  OF ␣
↪MERCHANTABILITY
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL␣
↪THE
# AUTHORS  OR  COPYRIGHT  HOLDERS  BE  LIABLE FOR ANY CLAIM,  DAMAGES OR␣
↪OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING␣
↪FROM,
# OUT OF OR IN  CONNECTION WITH  THE SOFTWARE OR  THE USE OR OTHER DEALINGS␣
↪IN
# THE SOFTWARE.
#
# Desc      Implements simple lora ABP activation.
# ------------------------------------------------------------------------
↪-- #

# required imports
import logs          # for system logging management
import lora          # for lora-stack
import ubinascii     # for hex/string conversions
import time          # for time manipulation

# disable fw logs
logs.filter_subsystem('lora', False)

# switch to lora-wan if not
lora.mode(lora._mode.WAN)

# configure the stack on region EU-868
lora.wan_params(region=lora._region.REGION_EU868, lwclass=lora._class.CLASS_
↪A)

# start end-device commissioning
lora.commission(
    type    = lora._commission.ABP,
    version = lora._version.VERSION_1_0_X,
    DevAddr = 0x00000000,
```

(continues on next page)

(continued from previous page)

```
    DevEUI  = ubinascii.unhexlify('0000000000000000'),
    AppSKey = ubinascii.unhexlify('00000000000000000000000000000000'),
    NwkSKey = ubinascii.unhexlify('00000000000000000000000000000000')
    )

# start join procedure
lora.join()

# wait until join
while lora.is_joined() == False:
    print("wait joining ...")
    time.sleep(2)
    pass
print("-- JOINED --")
lora.stats()

# open a working port
lora.port_open(1)

# attach required callback
def get_event_str(event):
    if event == lora._event.EVENT_TX_CONFIRM:
        return 'EVENT_TX_CONFIRM'
    elif event == lora._event.EVENT_TX_DONE:
        return 'EVENT_TX_DONE'
    elif event == lora._event.EVENT_TX_TIMEOUT:
        return 'EVENT_TX_TIMEOUT'
    elif event == lora._event.EVENT_TX_FAILED:
        return 'EVENT_TX_FAILED'
    elif event == lora._event.EVENT_RX_DONE:
        return 'EVENT_RX_DONE'
    elif event == lora._event.EVENT_RX_TIMEOUT:
        return 'EVENT_RX_TIMEOUT'
    elif event == lora._event.EVENT_RX_FAIL:
        return 'EVENT_RX_FAIL'
    else:
        return 'UNKNOWN'

def port_any_cb(event, evt_data):
    print('lora event [ {} ] --> data: {}'
        .format(get_event_str(event), evt_data))

lora.callback(handler=port_any_cb)

# start duty cycle
lora.duty_set(10000)
lora.enable_rx_listening()
lora.duty_start()
```

(continues on next page)

(continued from previous page)

```
# schedule sending some test messages
i = 1000
while i < 1010:
    lora.send('tx-message-with-id-{}'.format(i), port=1, confirm=True, id =␣
↪i)
    i = i + 1

# --- end of file ------------------------------------------------------------
↪-- #
```

For LoRaWAN v1.1.x:

```
# ------------------------------------------------------------------------------
↪-- #
# Copyright (c) 2023-2024 SG Wireless - All Rights Reserved
#
# Permission is hereby granted, free of charge, to any person obtaining a␣
↪copy
# of this software and associated documentation files(the "Software"), to␣
↪deal
# in the Software without restriction, including without limitation the␣
↪rights
# to use,  copy,  modify,  merge, publish, distribute, sublicense, and/or␣
↪sell
# copies  of  the  Software,  and  to  permit  persons to whom the Software␣
↪is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS",  WITHOUT WARRANTY OF ANY KIND,  EXPRESS␣
↪OR
# IMPLIED,  INCLUDING BUT NOT LIMITED TO  THE  WARRANTIES  OF ␣
↪MERCHANTABILITY
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL␣
↪THE
# AUTHORS  OR  COPYRIGHT  HOLDERS  BE  LIABLE FOR ANY CLAIM,  DAMAGES OR␣
↪OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING␣
↪FROM,
# OUT OF OR IN  CONNECTION WITH  THE SOFTWARE OR  THE USE OR OTHER DEALINGS␣
↪IN
# THE SOFTWARE.
#
# Desc      Implements simple lora ABP activation.
# ------------------------------------------------------------------------------
↪-- #

# required imports
```

(continues on next page)

(continued from previous page)

```
import logs          # for system logging management
import lora          # for lora-stack
import ubinascii     # for hex/string conversions
import time          # for time manipulation

# disable fw logs
logs.filter_subsystem('lora', False)

# switch to lora-wan if not
lora.mode(lora._mode.WAN)

# configure the stack on region EU-868
lora.wan_params(region=lora._region.REGION_EU868, lwclass=lora._class.CLASS_
↪A)

# start end-device commissioning
lora.commission(
    type    = lora._commission.ABP,
    version = lora._version.VERSION_1_1_X,
    DevAddr = 0x00000000,
    DevEUI  = ubinascii.unhexlify('0000000000000000'),
    AppSKey = ubinascii.unhexlify('00000000000000000000000000000000'),
    NwkSKey = ubinascii.unhexlify('00000000000000000000000000000000')
    )

# start join procedure
lora.join()

# wait until join
while lora.is_joined() == False:
    print("wait joining ...")
    time.sleep(2)
    pass
print("-- JOINED --")
lora.stats()

# open a working port
lora.port_open(1)

# attach required callback
def get_event_str(event):
    if event == lora._event.EVENT_TX_CONFIRM:
        return 'EVENT_TX_CONFIRM'
    elif event == lora._event.EVENT_TX_DONE:
        return 'EVENT_TX_DONE'
    elif event == lora._event.EVENT_TX_TIMEOUT:
        return 'EVENT_TX_TIMEOUT'
    elif event == lora._event.EVENT_TX_FAILED:
        return 'EVENT_TX_FAILED'
```

(continues on next page)

(continued from previous page)

```
    elif event == lora._event.EVENT_RX_DONE:
        return 'EVENT_RX_DONE'
    elif event == lora._event.EVENT_RX_TIMEOUT:
        return 'EVENT_RX_TIMEOUT'
    elif event == lora._event.EVENT_RX_FAIL:
        return 'EVENT_RX_FAIL'
    else:
        return 'UNKNOWN'

def port_any_cb(event, evt_data):
    print('lora event [ {} ] --> data: {}'
        .format(get_event_str(event), evt_data))

lora.callback(handler=port_any_cb)

# start duty cycle
lora.duty_set(10000)
lora.enable_rx_listening()
lora.duty_start()

# schedule sending some test messages
i = 1000
while i < 1010:
    lora.send('tx-message-with-id-{}'.format(i), port=1, confirm=True, id =␣
↪i)
    i = i + 1

# --- end of file ------------------------------------------------------------
↪-- #
```

LoRa-MAC (Raw LoRa)

Basic LoRa connection example, sending and receiving data. In LoRa-MAC mode the LoRaWAN layer is bypassed and the radio is used directly. The data sent is not formatted or encrypted in any way, and no addressing information is added to the frame.

For the example below, you will need two F1 starter kits. Run the code below on the two F1 starter kits, you will see the word 'Hello from LoRa chat' being received on both sides.

```
# ------------------------------------------------------------------------------
↪-- #
# Copyright (c) 2023-2024 SG Wireless - All Rights Reserved
#
# Permission is hereby granted, free of charge, to any person obtaining a␣
↪copy
# of this software and associated documentation files(the "Software"), to␣
↪deal
# in the Software without restriction, including without limitation the␣
↪rights
# to use,  copy,  modify,  merge, publish, distribute, sublicense, and/or␣
```

(continues on next page)

(continued from previous page)

```
↪sell
# copies  of  the  Software,  and  to  permit  persons to whom the Software␣
↪is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS",  WITHOUT WARRANTY OF ANY KIND,  EXPRESS␣
↪OR
# IMPLIED,  INCLUDING BUT NOT LIMITED TO  THE  WARRANTIES  OF ␣
↪MERCHANTABILITY
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL␣
↪THE
# AUTHORS  OR  COPYRIGHT  HOLDERS  BE  LIABLE FOR ANY CLAIM,  DAMAGES OR␣
↪OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING␣
↪FROM,
# OUT OF OR IN  CONNECTION WITH  THE SOFTWARE OR  THE USE OR OTHER DEALINGS␣
↪IN
# THE SOFTWARE.
#
# Desc      Implements simple lora chat demo for the lora-raw mode
#           should be run on two different devices.
# ------------------------------------------------------------------------
↪-- #

# disable logs
import logs
logs.filter_subsystem('lora', False)

# import the responsible module
import lora

# switch to lora raw if not there
if lora.mode() != lora._mode.RAW:
    lora.mode(lora._mode.RAW)

# define the callback
def lora_callback(event, bytes):
    if event == lora._event.EVENT_RX_DONE:
        print(bytes)
    pass
lora.callback(handler=lora_callback)

# start continuous reception
lora.recv_cont_start()

# start chatting by sending
```

(continues on next page)

(continued from previous page)

```
lora.send('Hello from LoRa chat')

# --- end of file ------------------------------------------------------
↪-- #
```

11.3.4 LTE Examples

The following tutorial demonstrates the use of the LTE CAT-M1 and NB-IoT functionality on cellular enabled SG modules.

> **Note**
>
> For F1 Starter Kit purchasers, a pre-registered SIM card is provided in the F1 Starter Kit. In case you would like to use your own SIM card, please make sure that your SIM card is registered and activated with your carrier.

This page discusses the usage of the LTE modem in more detail:

- *General remarks*
- *SG SIM card*
- *LTE Connectivity check*
- *LTE Mode check and LTE Mode switching*
- *Custom APN setting*
- *Event-driven connection*

General remarks

To check the current modem firmware, you can use the following commands:

```
import lte

lte.init()
print(lte.send_at_cmd('ATI1'))
```

To check whether the LTE connection has been established, you can use the following command:

```
import lte

lte.init()
print(lte.isconnected())  # if True is returned, LTE connection is␣
↪established
```

> **Note**
>
> The first time, it can take a long while to attach to the network.

SG SIM card

For F1 Starter Kit purchasers, a pre-registered and pre-setup SIM card is placed inside the SIM card slot on the F1 Starter Kit.

There is pre-stored value in each SIM card for use. The SIM card is intended to support nominal development/evaluation and operating under nominal IoT application data usage. In case you need to implement data-heavy applications or would like to have batch deployment of LTE-enabled devices, please contact the SG sales representative for enquiry and special discussions.

LTE Connectivity check

To check whether the LTE connection has been established, you can use the following command:

```
import lte

lte.init()
print(lte.isconnected())  # if True is returned, LTE connection is␣
↪established
```

You can also get comprehensive status information:

```
import lte

lte.init()
status = lte.get_status()
print('Powered:', status['powered'])
print('Network:', status['network_attached'])
print('PPP:', status['ppp_connected'])
print('Signal:', status['rssi'])
```

LTE Mode check and LTE Mode switching

For the F1 Starter Kit pre-registered and pre-setup SIM card, the LTE mode is set up for users based on the purchasing order.

In case you are using your own SIM card and would like to check the current LTE mode (CAT-M1 or NB-IoT) and switch your LTE mode, you can use the following commands:

```
import lte

lte.init()
current = lte.mode()  # Return current mode
if current == lte.CATM1:
    print('Modem is in CAT-M1 mode!')
if current == lte.NBIOT:
    print('Modem is in NB-IoT mode!')

# Setting a new mode causes the modem to reset
lte.mode(lte.CATM1)  # switch to CAT-M1
lte.mode(lte.NBIOT)  # switch to NB-IoT
```

> **Note**
>
> If you are using your own SIM card and would like to change LTE mode, please check carefully with your network service provider whether the LTE mode you would like to change to is supported or not.

Custom APN setting

In case you are using your own SIM card with a different service provider, the APN (Access Point Name) should be changed and set correctly.

Below you will find the command for setting a custom APN:

```
import lte
import time

lte.init()
lte.attach(apn='iot.1nce.net')  # use 'iot.1nce.net' for SG SIM card,
                                # change to your ISP's APN if using your␣
↪own SIM
for i in range(180):  # 3 minute timeout
    if lte.isattached():
        print('Attached!')
        break
    time.sleep(1)

lte.connect()
for i in range(60):
    if lte.isconnected():
        print('Connected!')
        print(lte.ifconfig())
        break
    time.sleep(1)
```

Event-driven connection

The C implementation supports event-driven LTE management. Instead of polling, you can register an event handler to receive connection status changes in real-time:

```
import lte

def lte_event_handler(event):
    event_type = event['type']

    if event_type == lte.EVENT_REGISTRATION_STATUS:
        stat = event['stat']
        if stat == 1:
            print('Registered (home network)')
        elif stat == 5:
            print('Registered (roaming)')
        elif stat == 2:
            print('Searching...')
```

(continues on next page)

(continued from previous page)

```
    elif event_type == lte.EVENT_PPP_CONNECTED:
        print('Connected! IP: {}'.format(event['ip']))

    elif event_type == lte.EVENT_PPP_DISCONNECTED:
        print('Disconnected')

    elif event_type == lte.EVENT_SIGNAL_QUALITY:
        print('Signal: {} dBm'.format(event['rssi_dbm']))

lte.init()
lte.set_event_handler(lte_event_handler, lte.EVENT_ALL)
lte.attach(apn='iot.1nce.net')
lte.connect()
```

CHAPTER

TWELVE

CONTACT

Email: info@sgwireless.com
Website: https://sgwireless.com/
LinkedIn: https://www.linkedin.com/company/sgwireless/

CHAPTER

THIRTEEN

LICENSE