SG Wireless Documentation

Release 1.4.0

SG Wireless

May 22, 2026

CONTENTS

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

CHAPTER

ONE

F1 SMART MODULE

Datasheet — March 2024 V1.1

Download PDF Datasheet

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.

1.1.1 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

1.1.2 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

1.2 General Features

1.2.1 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

1.3 Pinout & Pin Definitions

`Module Pinout Diagram - PDF`

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

1.3.1 Pin Definitions

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

continues on next page

Table 1 – continued from previous page

Pin Number	Pin Name	MCU Pin	LTE Module Pin	Type	Description
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
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

continues on next page

Table 1 – continued from previous page

Pin Number	Pin Name	MCU Pin	LTE Module Pin	Type	Description
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
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

continues on next page

Table 1 – continued from previous page

Pin Number	Pin Name	MCU Pin	LTE Module Pin	Type	Description
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

1.4 Electrical Specifications

1.4.1 Rating & Operating Conditions

Table 1: Absolute Rating and Operating Conditions Specifications

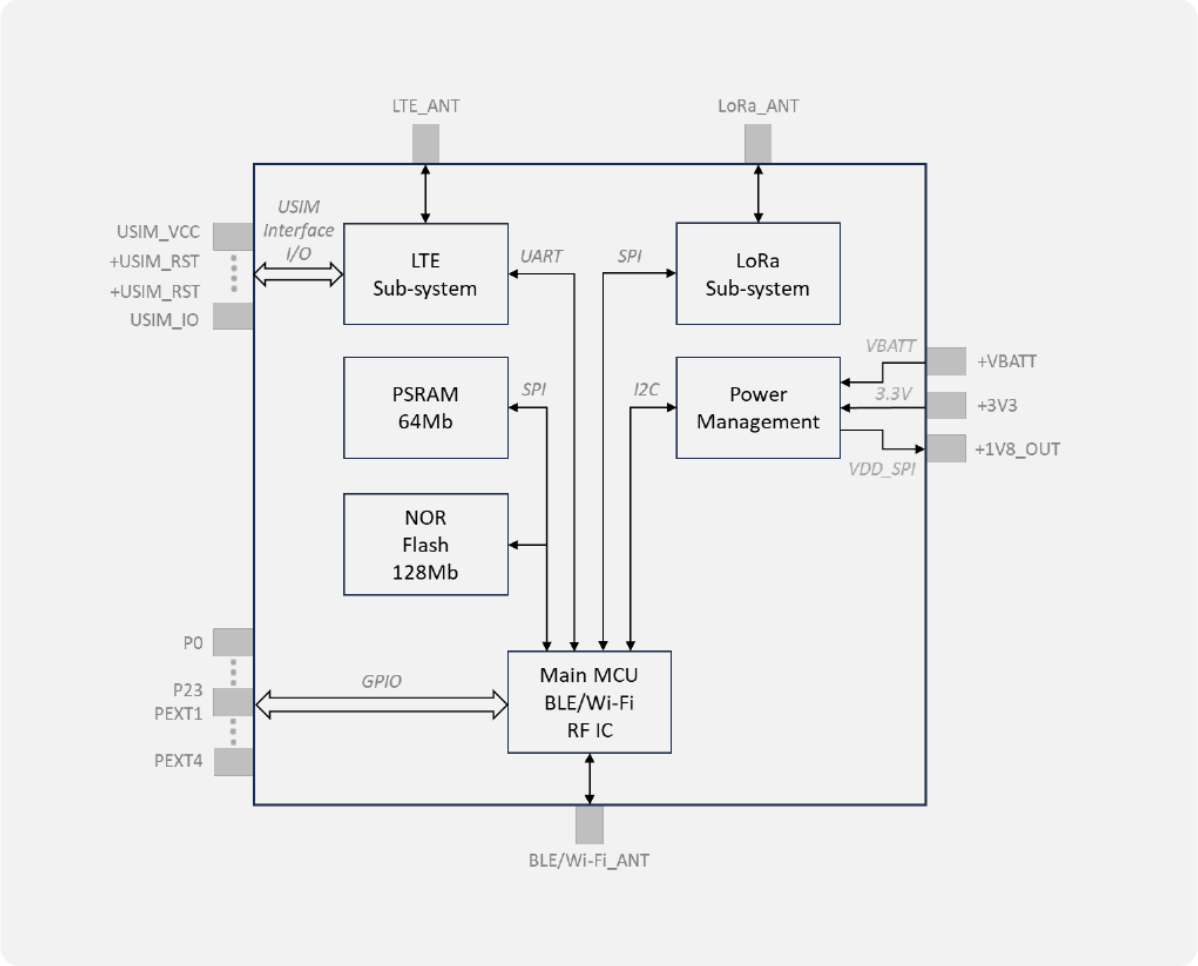


Fig. 1: *Figure 1: F1 Smart Module Block Diagram*

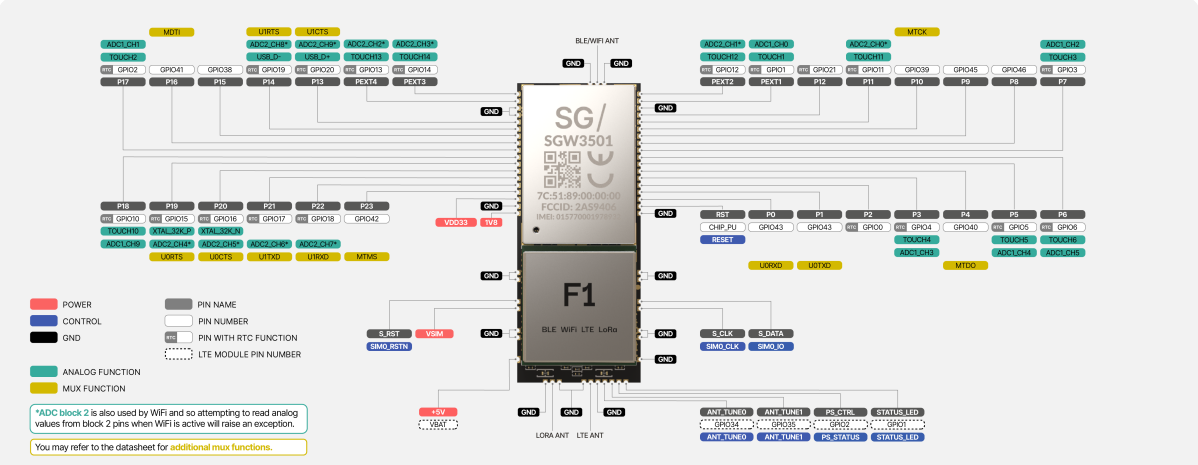


Fig. 2: *Figure 1: F1 Smart Module Pinout Diagram*

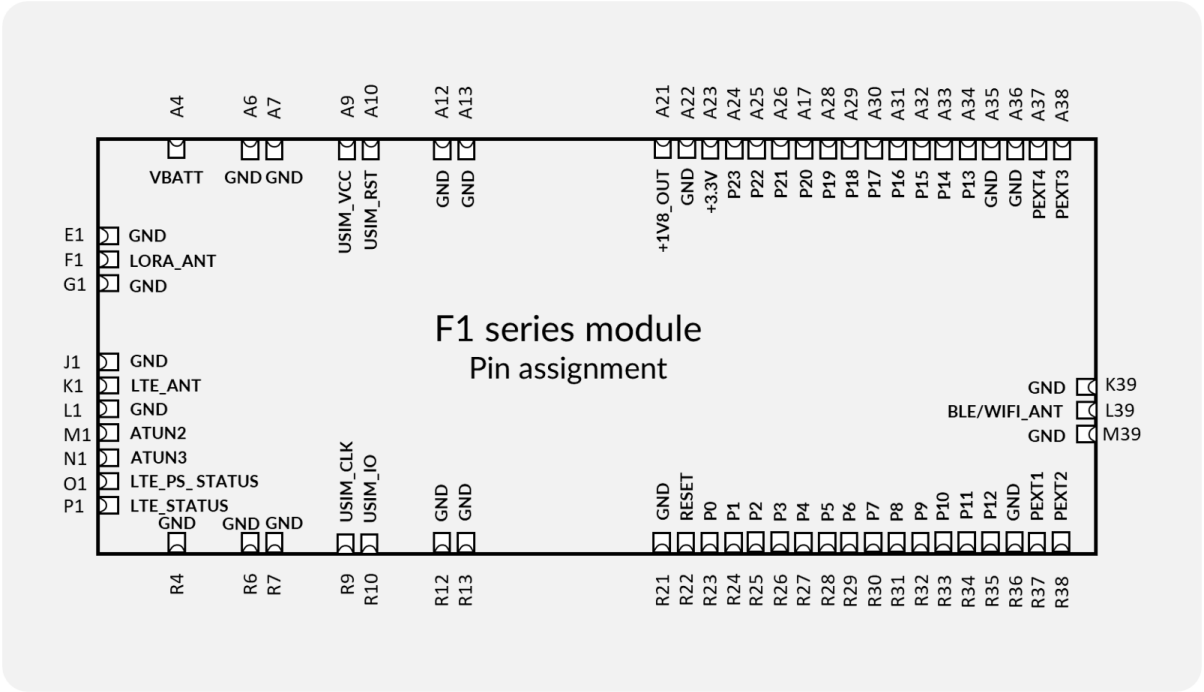


Fig. 3: *Figure 2: F1 Smart Module Pin-out (Top View)*

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

* 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.

1.4.2 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

1.4.3 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

1.4.4 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

1.4.5 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

1.5 Module Interface

1.5.1 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

1.5.2 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

1.6 Mechanical Specifications

1.6.1 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.

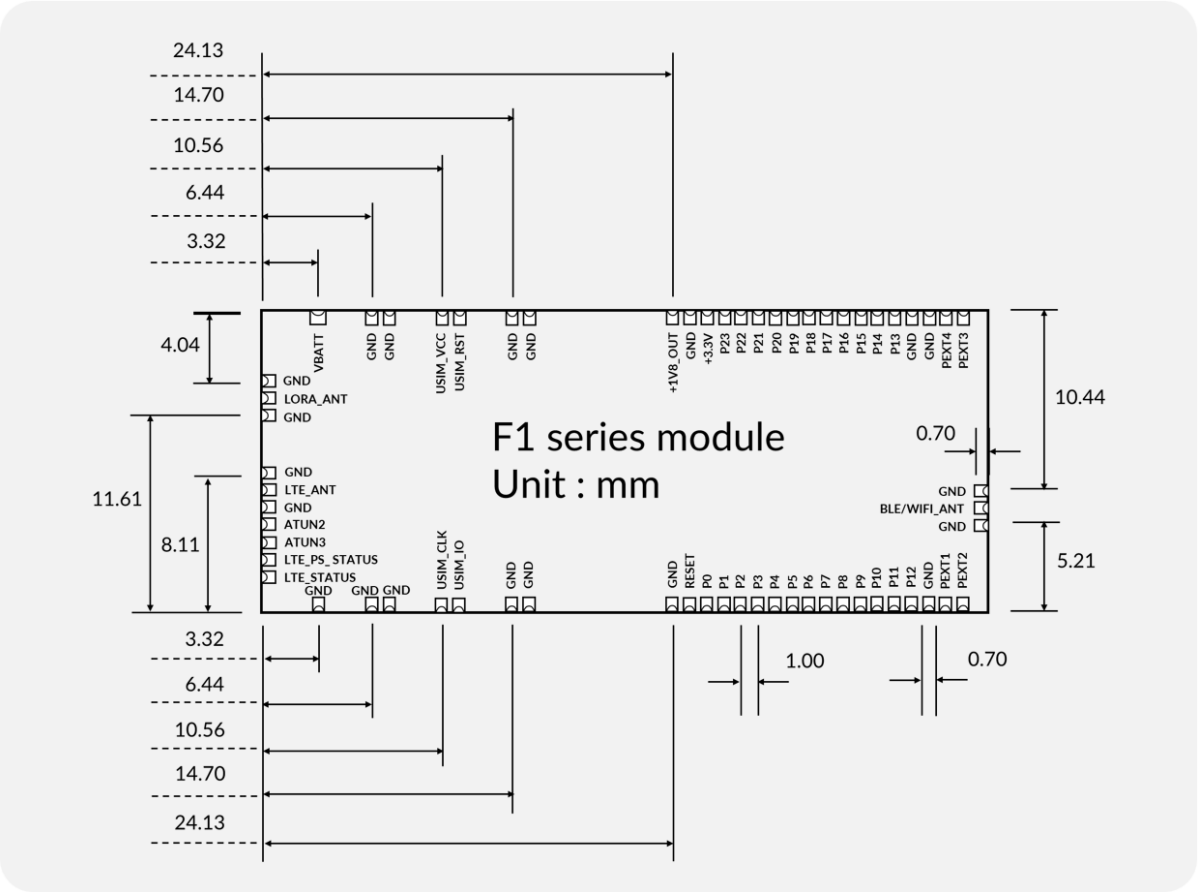


Fig. 4: *Figure 4: F1 Smart Module Dimensions*

1.6.2 Recommended PCB Landing Pattern

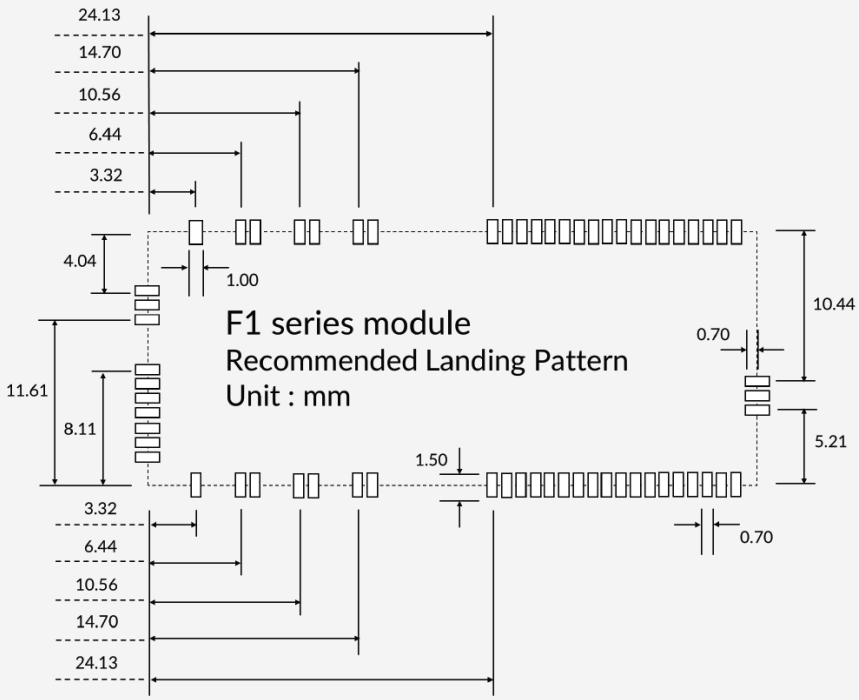


Fig. 5: *Figure 5: Recommended PCB Landing Pattern (Top View)*

1.6.3 Recommended Soldering Profile

1.7 MicroPython

1.7.1 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.

1.7.2 Module-supported Libraries

Table 9: F1 Smart Module Supported Libraries

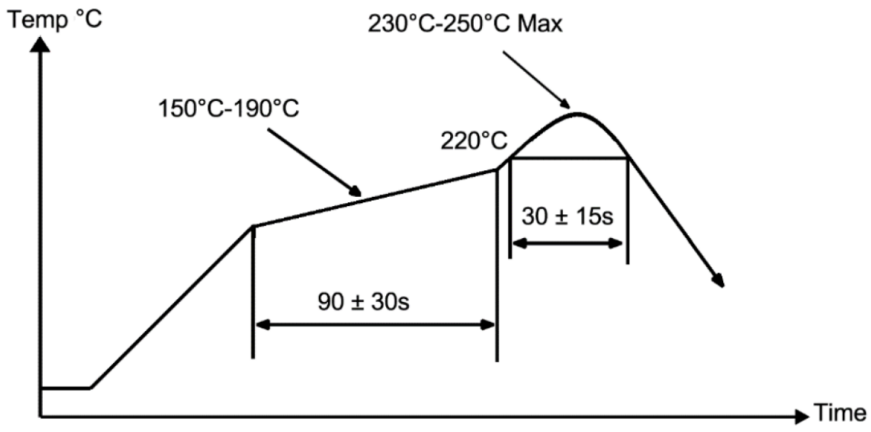


Fig. 6: *Figure 6: Recommended Soldering Profile*

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).

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.

1.9 Revision History

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

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)

CHAPTER

TWO

F1 STARTER KIT

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.

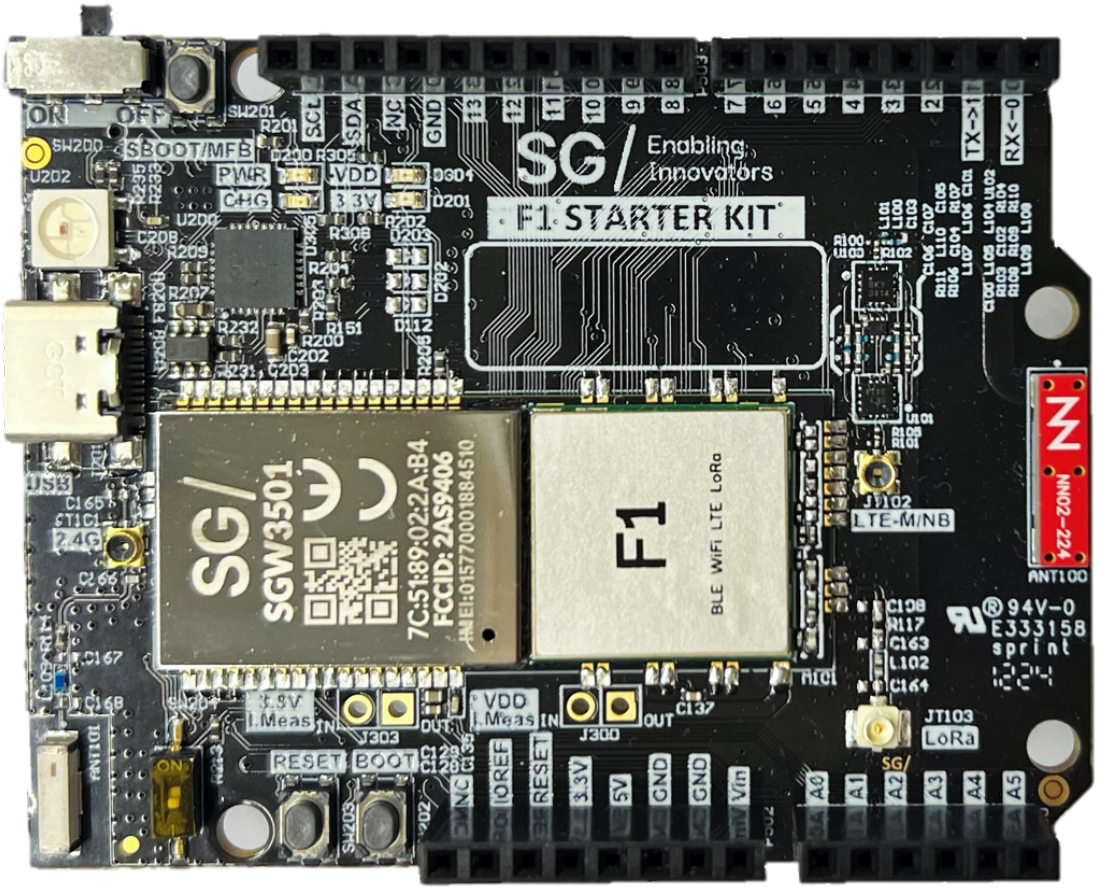


Fig. 1: *F1 Starter Kit*

2.1.1 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.

2.1.2 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.

2.2 Features

2.2.1 Connectivity

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

2.2.2 Microprocessor

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

2.2.3 Memory

- Additional 8 MB PSRAM
- Additional 16 MB NOR Flash

2.2.4 Peripherals

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

2.2.5 On-board Peripherals

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

2.2.6 Arduino UNO Shield Compatible Expansion

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

2.2.7 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

2.3 Starter Kit Overview

2.3.1 Block Diagram

2.3.2 Starter Kit Topology — Top View

Major Component List (Top)

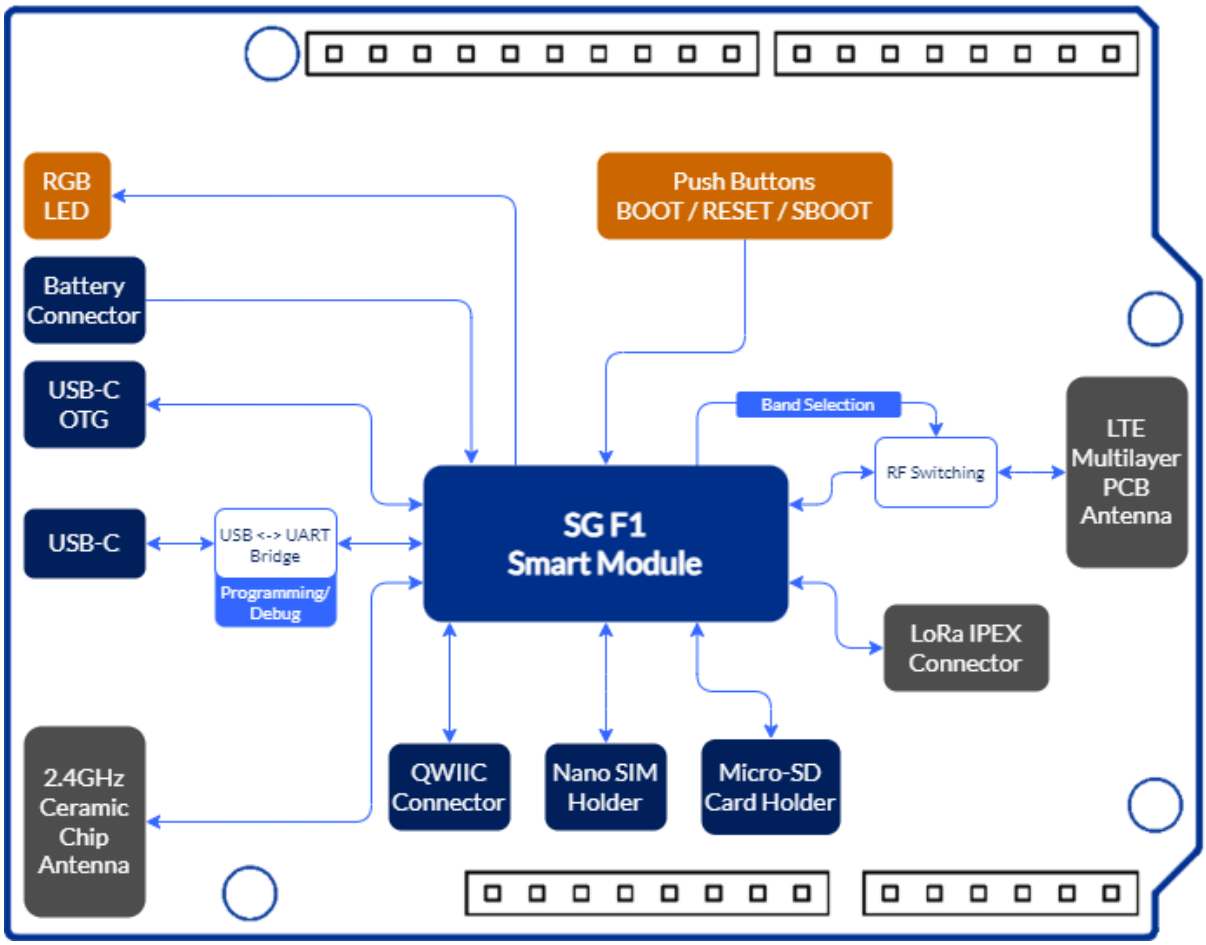


Fig. 2: *F1 Starter Kit Block Diagram*

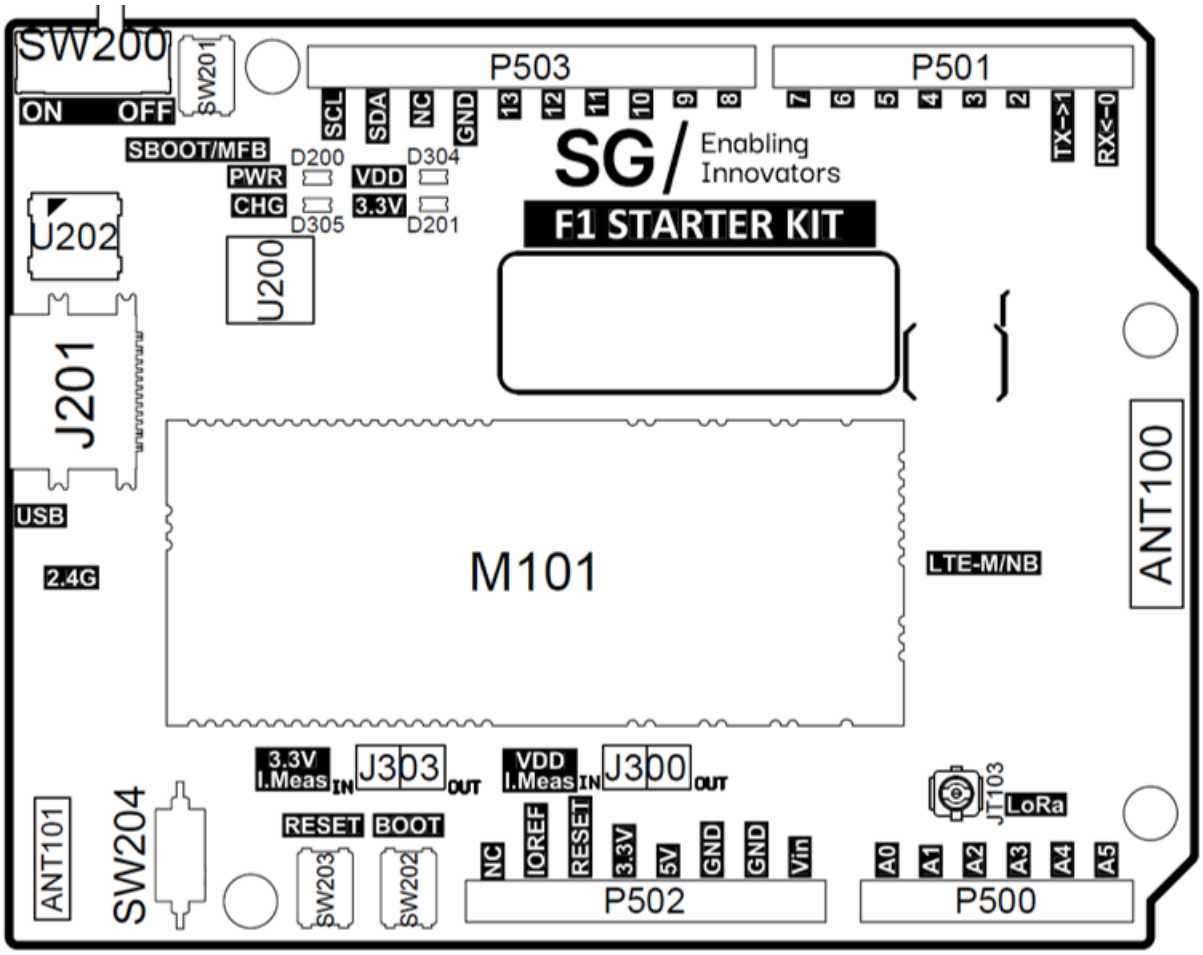


Fig. 3: *F1 Starter Kit — Top View*

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		

2.3.3 Starter Kit Topology — Bottom View

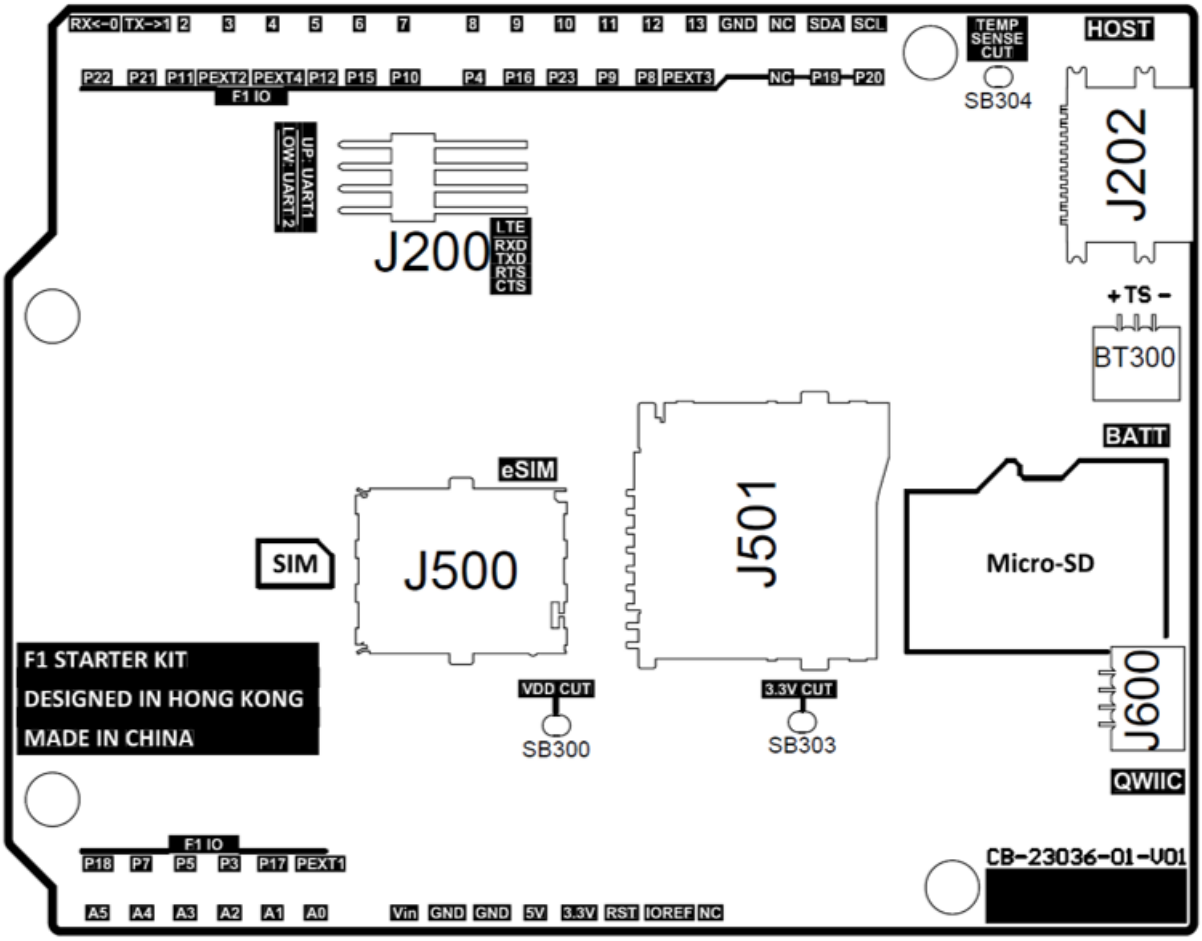


Fig. 4: *F1 Starter Kit — Bottom View*

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

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`

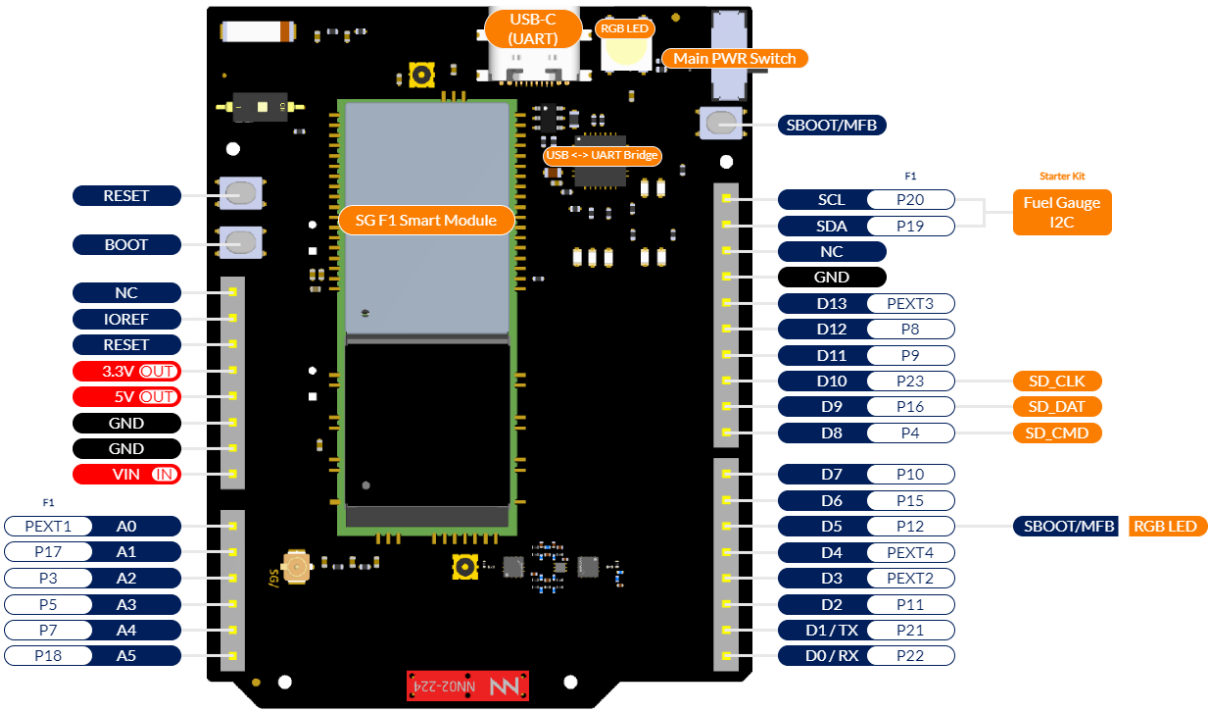


Fig. 5: *Figure 1: F1 Starter Kit Pinout Diagram*

2.4.1 Pin Definitions

Table 1: Analog / Power Pin Table

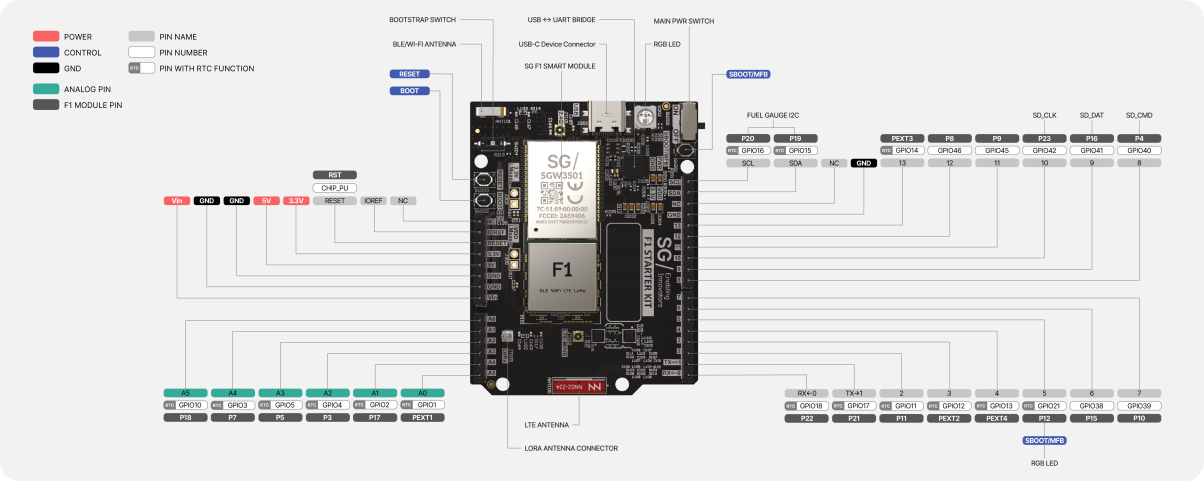


Fig. 6: *Figure 2: Top View of F1 Starter Kit*

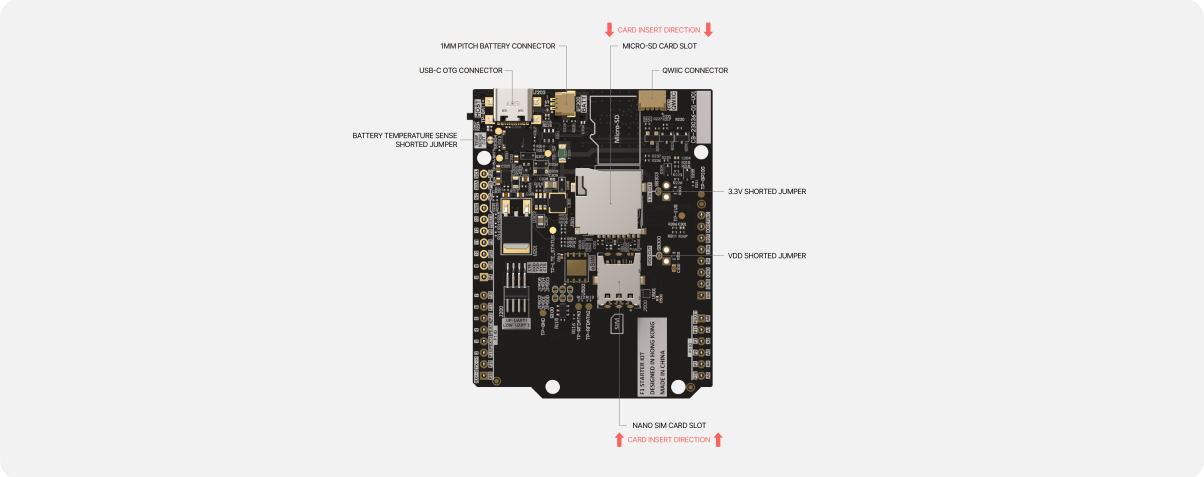


Fig. 7: *Figure 3: Bottom View of F1 Starter Kit*

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.

2.5 Board Operation

- *Getting Started* — Setup computer for F1
- *Tutorials & Examples* — Tutorials and examples
- Additional online resources at https://www.sgwireless.com

See *Getting Started* for setup instructions.

CHAPTER

THREE

GETTING STARTED

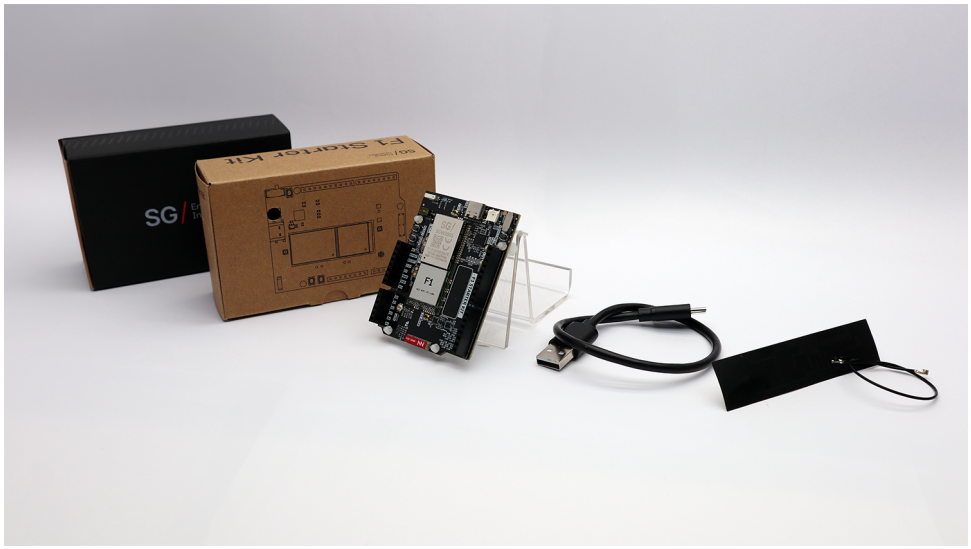
We designed the F1 Starter Kit to get you connected straight out of the box.

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.

3.1 Unboxing your Starter Kit

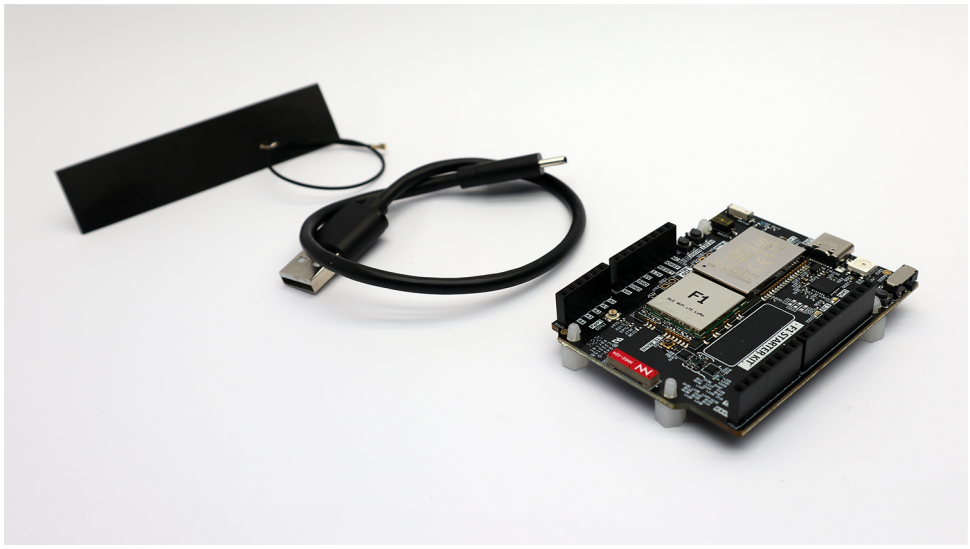
While our packaging is recyclable, don't throw it away just yet! It is in fact a ready-to-go housing that you can already use for initial prototyping. Here you'll see why.



Your F1 Starter Kit has 3 components:

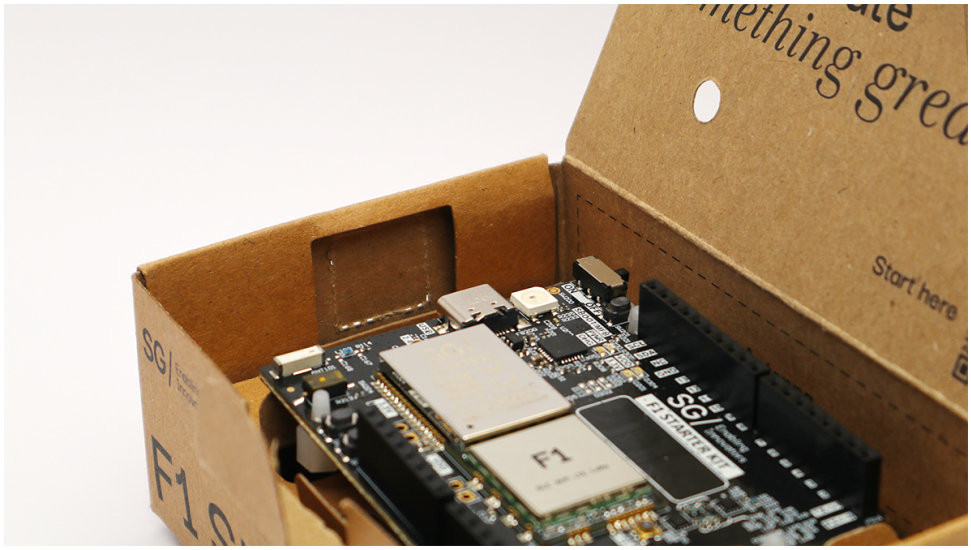
- F1 Evaluation Board with global SIM card (pre-inserted)
- USB cable (type A to type C)
- LoRa FPC antenna

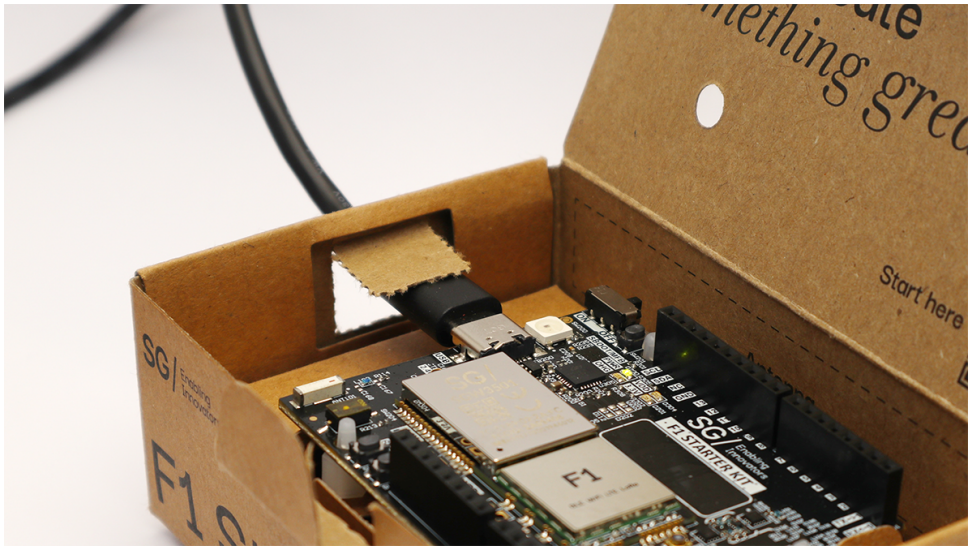
The F1 EVB is enclosed in an anti-static bag (not pictured).



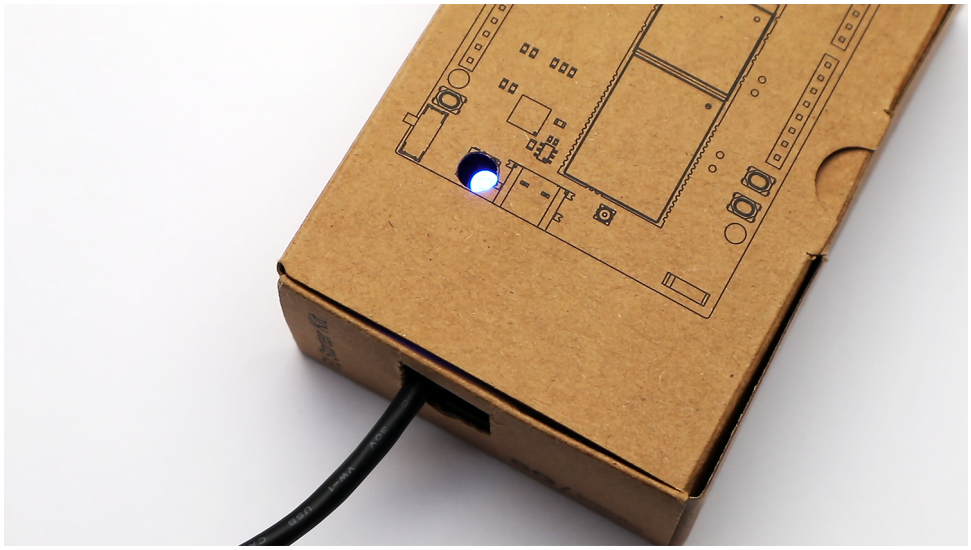
The packaging will secure your F1 Evaluation Board and keep it safe from the elements as you conduct your evaluation.

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.

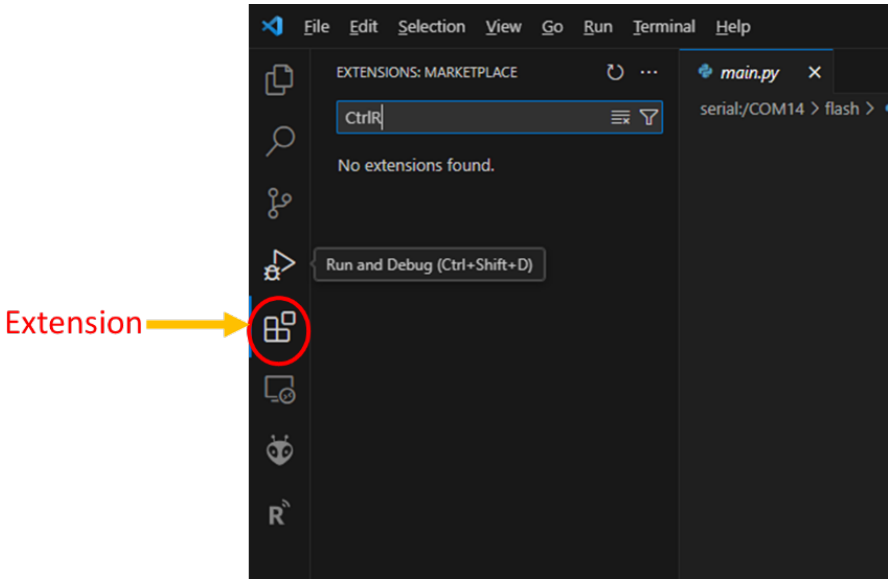
3.2 Introducing VS CtrlR

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.

3.2.1 Installing VS CtrlR Plugin

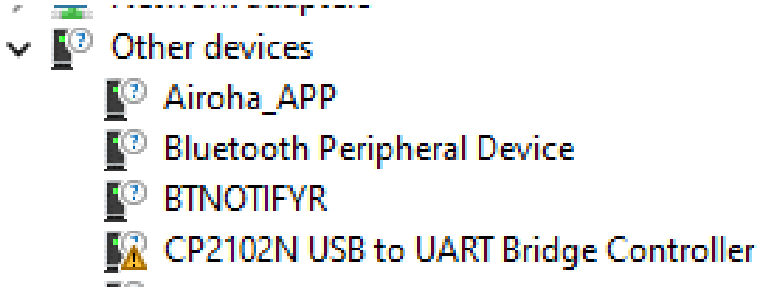
1. Launch Visual Studio Code and navigate to Extensions. Search for “CtrlR” and click Install.



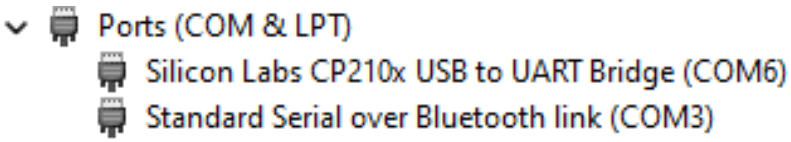
2. Press the Reload button to complete the CtrlR Plugin installation.

3.2.2 Configuring F1 Starter Kit for CtrlR Plugin on your PC

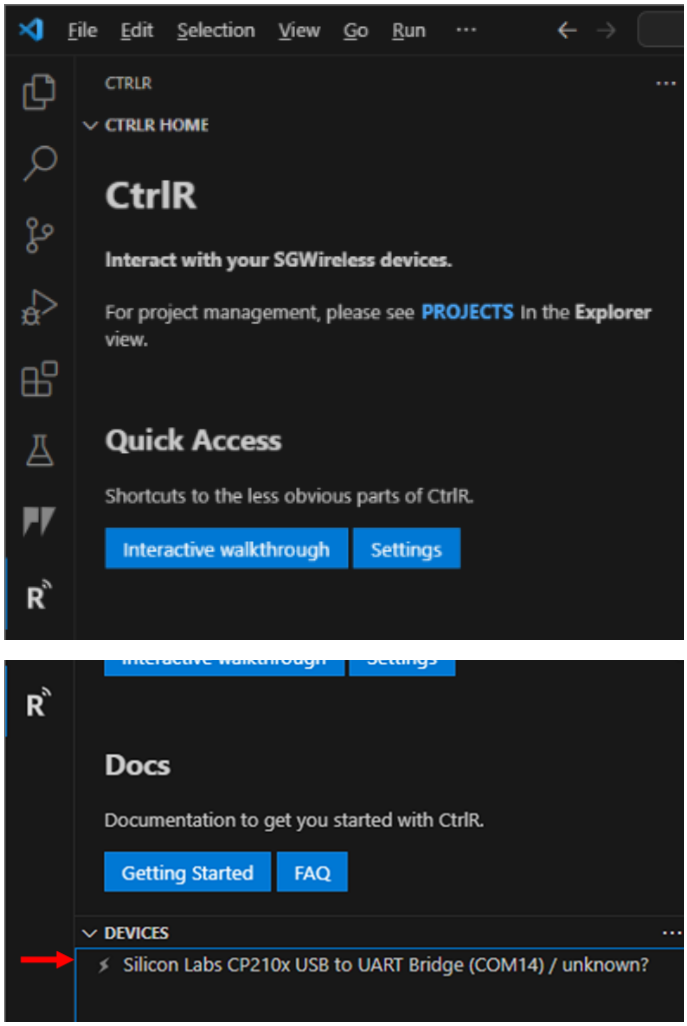
1. Connect your F1 Starter Kit to your PC and turn it on (SW200 switch from OFF to ON).
2. If your PC doesn't have the Silabs CP2102N Virtual COM port driver installed previously, an unknown device will be shown in your PC's Device Manager.



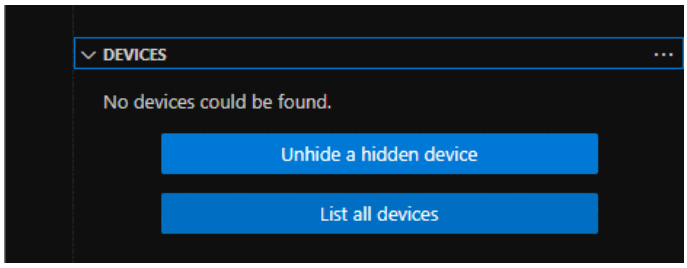
3. Download and install the Virtual COM Port driver.
4. Upon installation of the driver, a new COM port should show up in your PC's Device Manager.



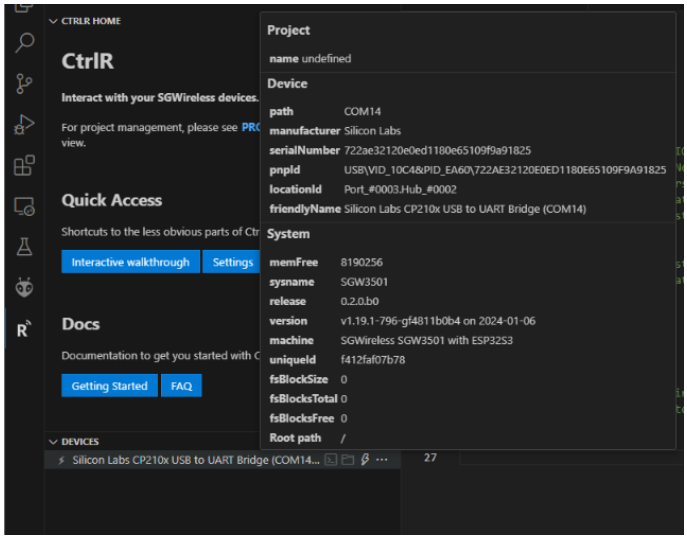
5. Go back to Visual Studio Code and ensure that the CtrlR Plugin has been correctly installed.



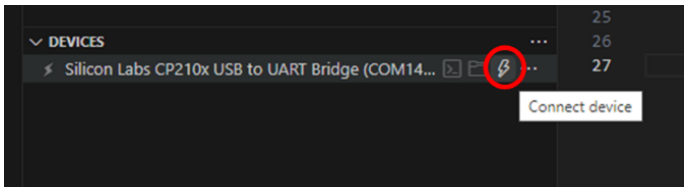
6. Normally, your device will be auto-detected as shown above. If this does not work, click **[List all devices]** in the DEVICES window.



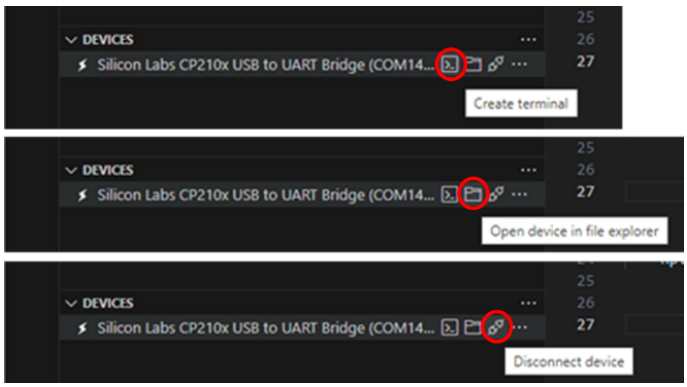
7. If everything is correct, device details will pop up when your mouse pointer is placed on the device:



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



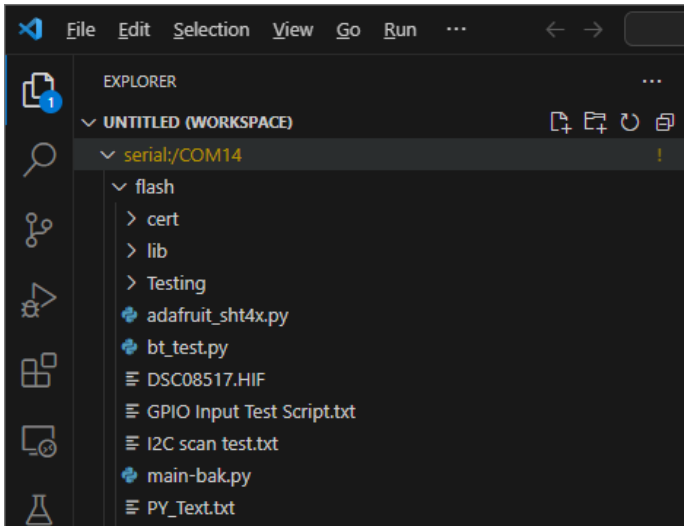
9. Next, you will be able to invoke a terminal, launch file explorer, or disconnect from the device with the 3 buttons.



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.



3.2.3 Next Steps

Program the Starter Kit to send your data to the Ctrl platform. See *Your First Sensor Data* or *Your First F1 Code*.

3.3 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 network.
- 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.

3.3.1 Regional Coverage

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

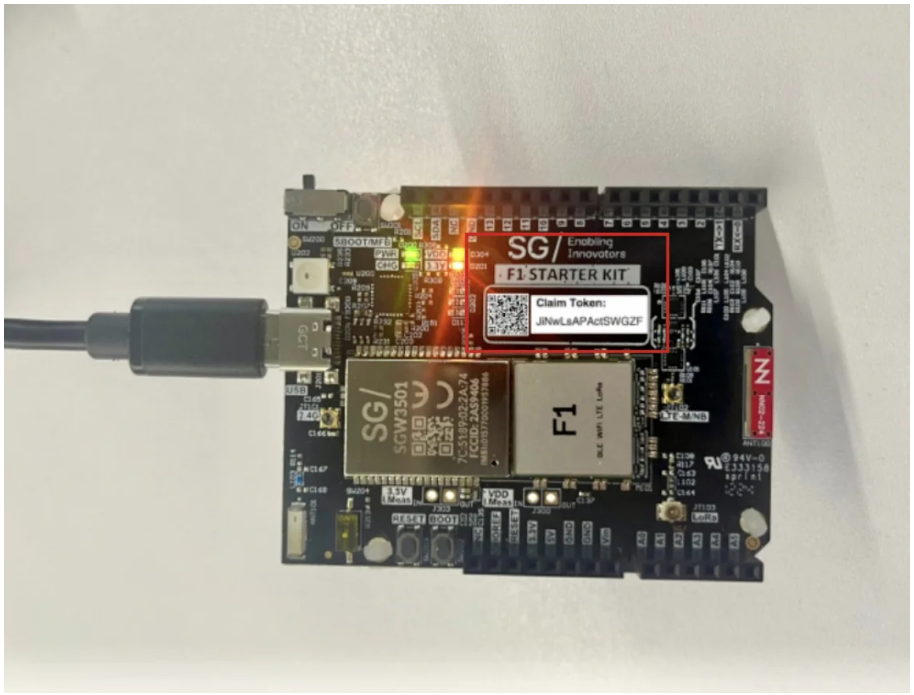
	F1 Starter Kit Configuration	Regions with Coverage
1	NA Cat M1	Canada, USA, Puerto Rico
2	EU Cat M1	Austria, Belgium, Denmark, Estonia, Latvia, Ireland, Finland, Germany, Hungary, The Netherlands, Norway, Poland, Spain, Sweden, Switzerland, UK, France, Luxembourg, Romania
3	Global Cat M1	Argentina, Brazil, 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.

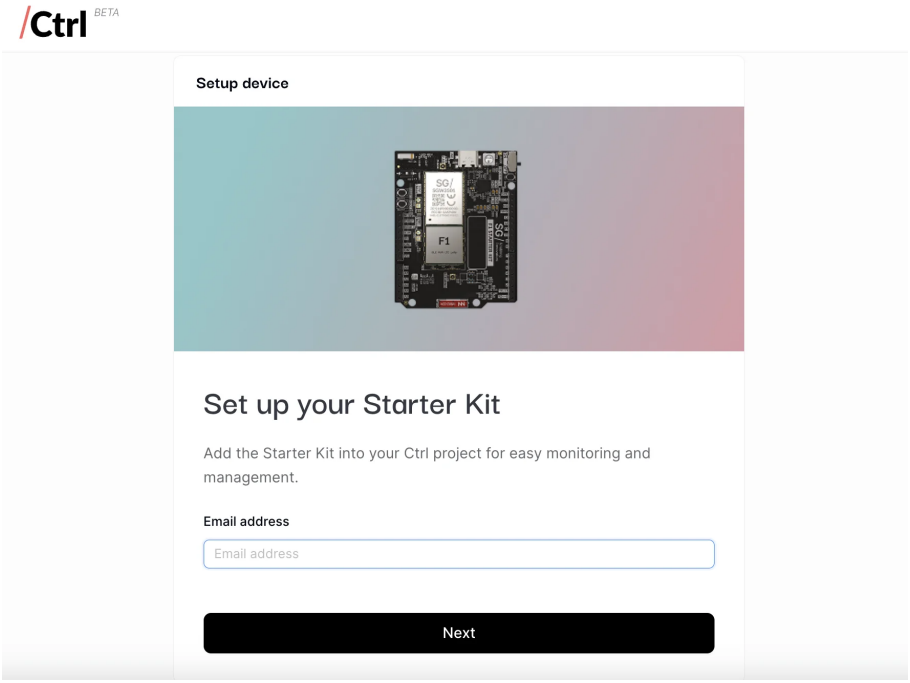
You can continue with *Manual Provisioning* if ZTP is not supported in your region.

3.3.2 Provisioning Steps

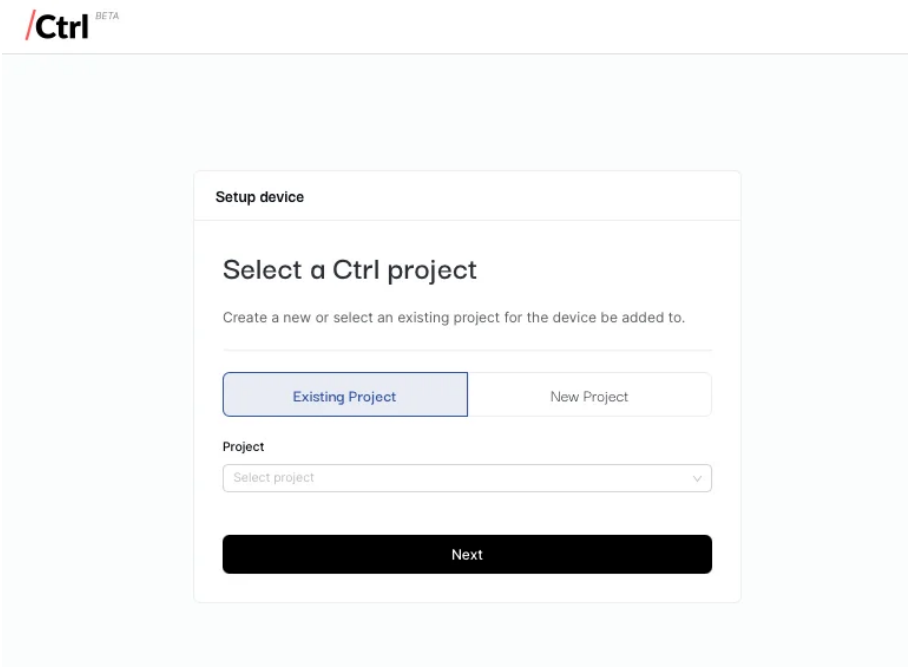
1. Unpack the F1 Starter Kit and scan the QR code on the F1 evaluation board. You'll be directed to the Ctrl IoT Platform.



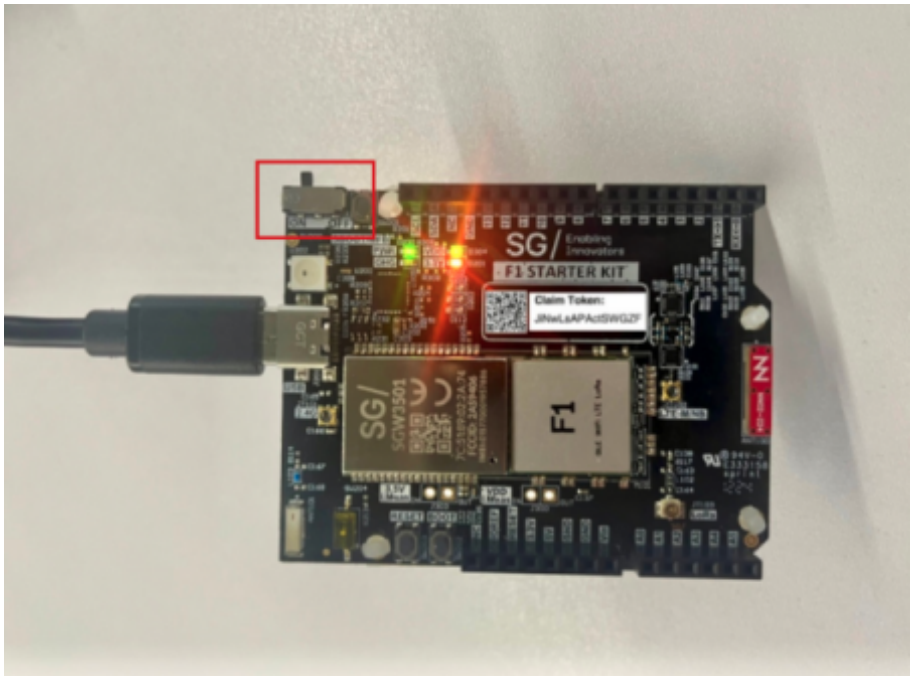
2. Input the email address to be associated with your Ctrl account. Your Starter Kit will be tied to this account. (No other Ctrl accounts can use the same Starter Kit until it is deleted from the account it is tied to.)



3. 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.)



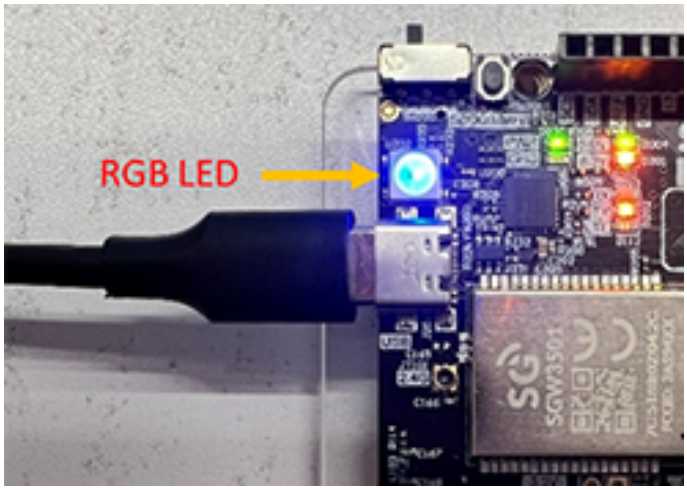
4. Connect the F1 evaluation board to a power source and toggle SW200 from OFF to ON (LEDs will light up).



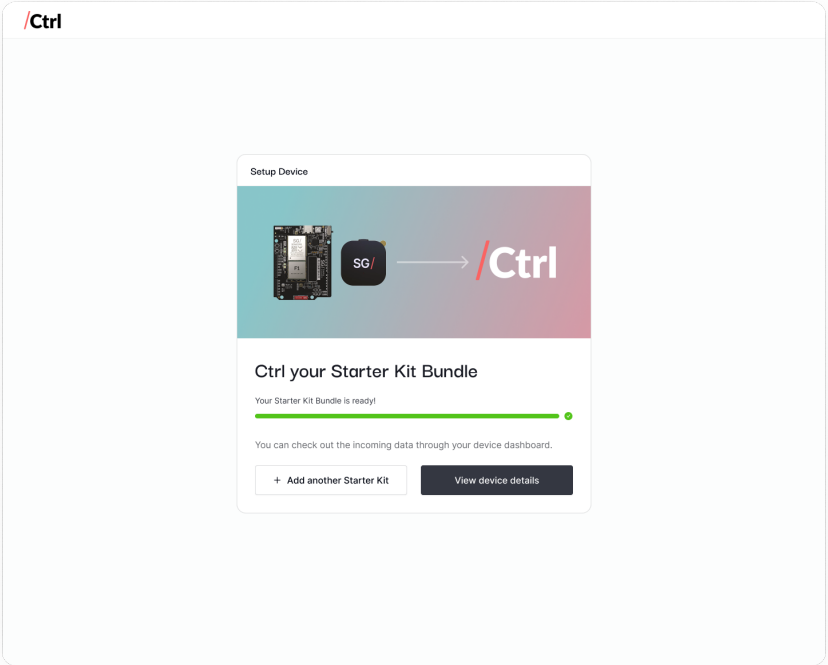
5. If you also purchased a CAP/T sensor, you can directly connect your sensor to your F1 during the provisioning flow. Remove the transparent battery cover from the sensor and press the button on the top corner. Notice that the sensor LED will be blinking green.

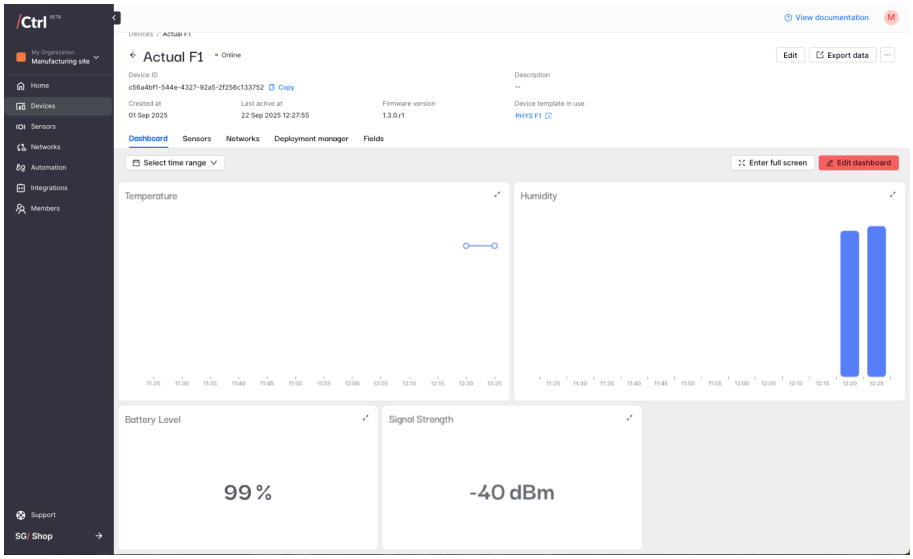
6. Monitor the provisioning status on your screen. You can also check the RGB LED on the F1 evaluation board:

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



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





3.3.3 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.

3.4 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*

3.4.1 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*

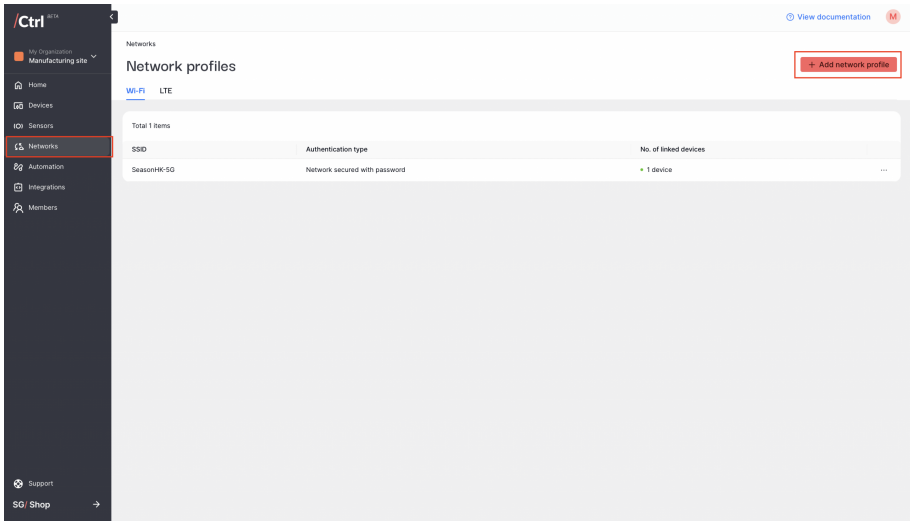
Last but not least, sign in to Ctrl here – create an account or log in, if you already have one.

3.4.2 Provisioning Steps

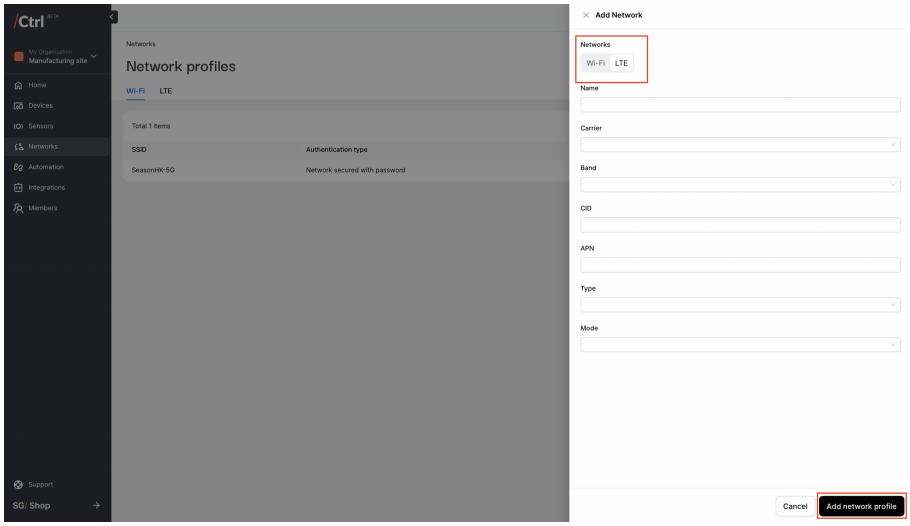
Part 1 – Configure your network profile

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

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

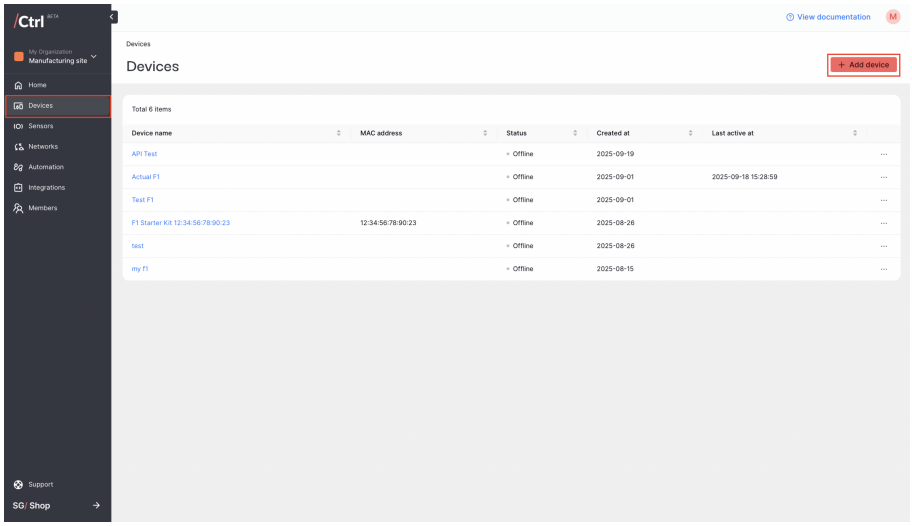


2. Choose “Wi-Fi” or “LTE”, and enter the required network credentials.
3. Click “Add network profile” – you’ll see the added network in “Networks”.

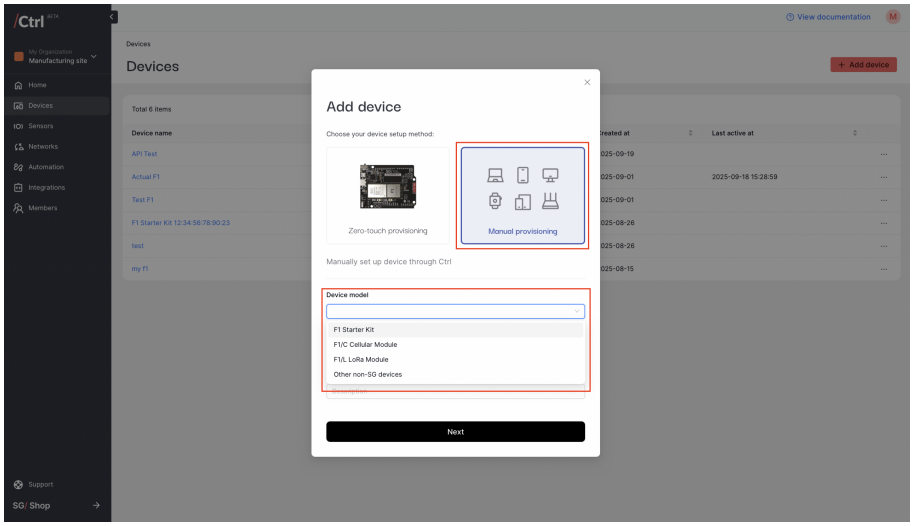


Part 2 – Add your F1 Starter Kit

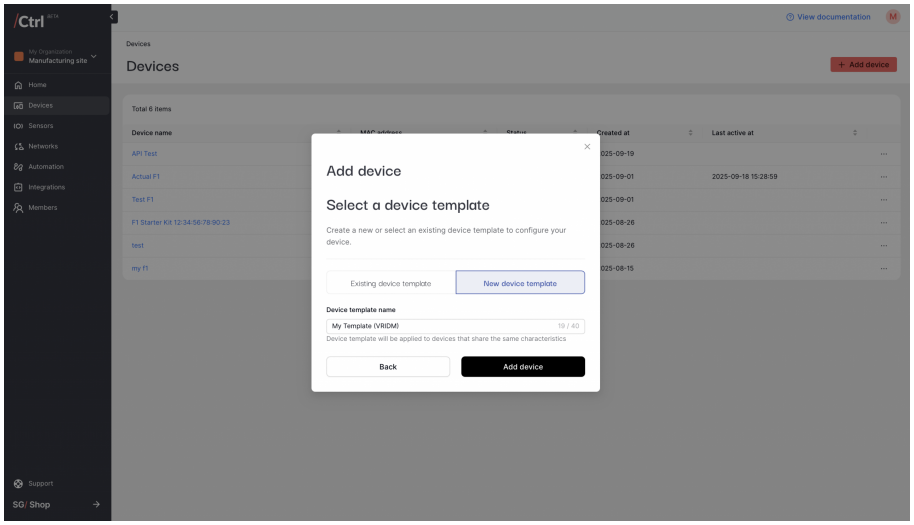
1. In the side menu, click “Devices”, then “Add device”.



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

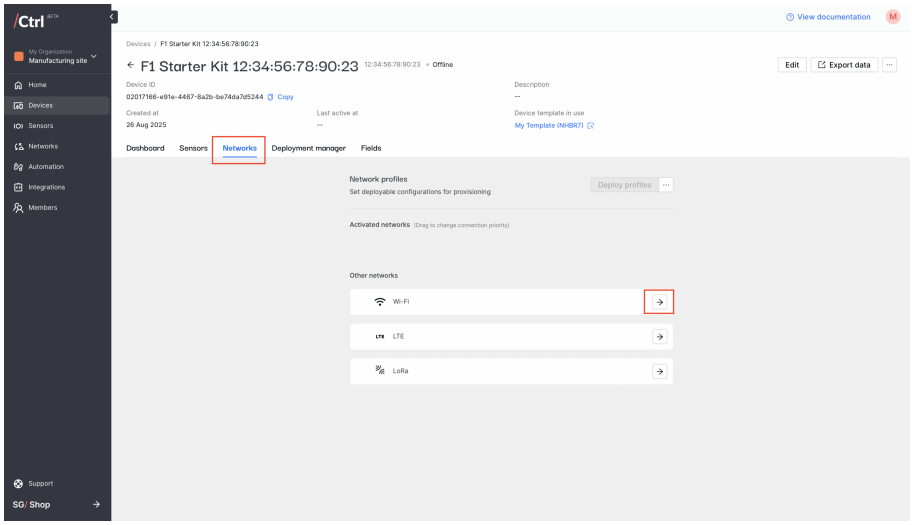


4. Select a Device Template – as this is your first device, you’ll land on “New device template” by default and you’ll be given a system-generated template identifier. (Device Templates will be important as you scale – more on this later!)
5. Click “Add device” to complete the device creation flow.

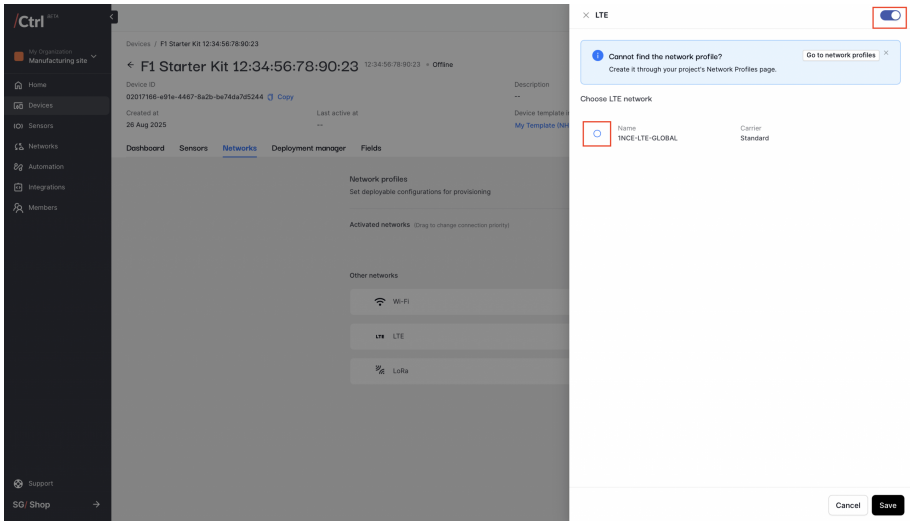


Part 3 – Configure your F1 Starter Kit Network

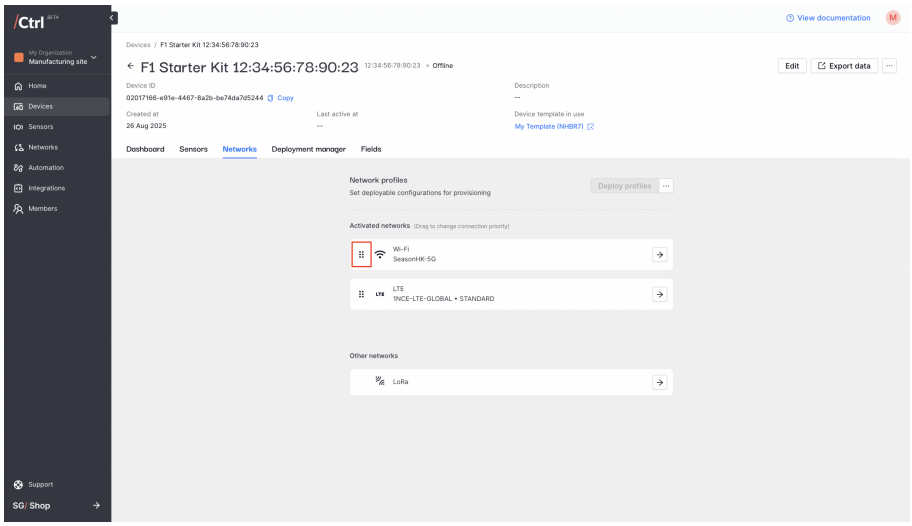
1. Click “View device details” to view your newly-created device page.
2. Click on the “Networks” tab and choose the network type that you want to activate.



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

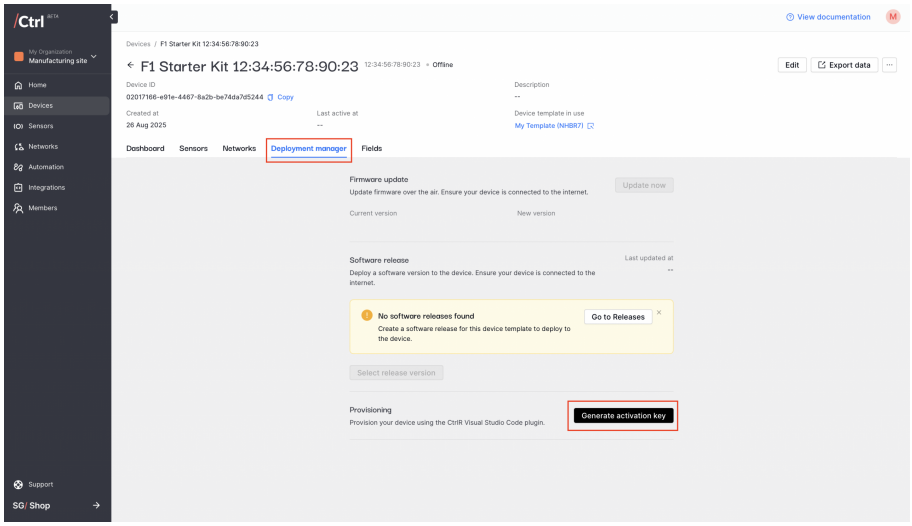


4. If you're enabling more than one network type, hold and drag the network type to adjust the priority.

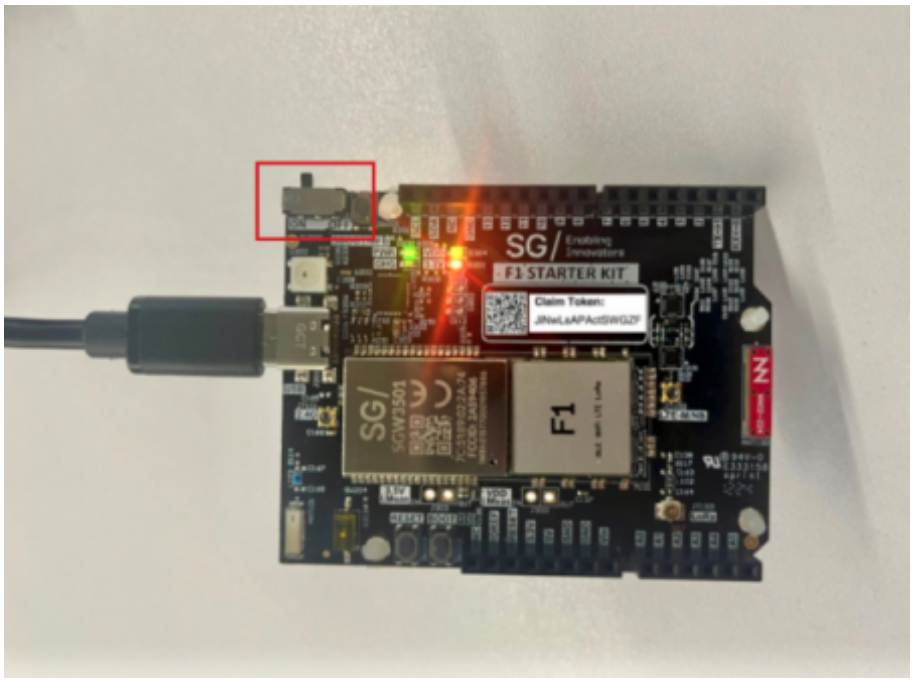


Part 4 – Deploy to your F1 Starter Kit

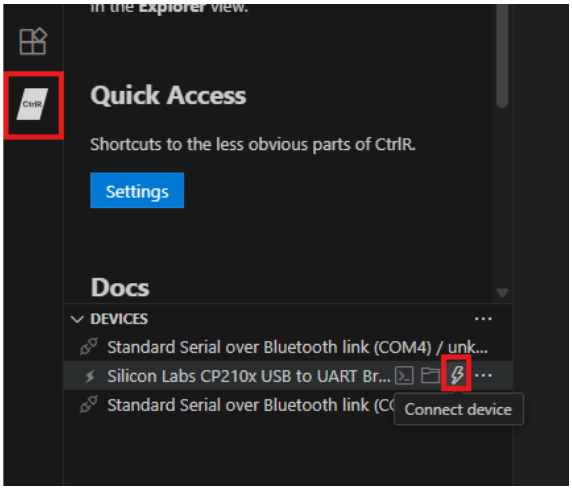
1. In the device's "Deployment Management" tab, click "Generate activation key". You'll need this key in a bit.



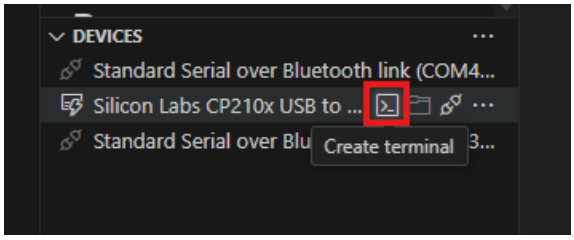
2. Connect the F1 evaluation board to a power source and toggle SW200 from OFF to ON (LEDs will light up).



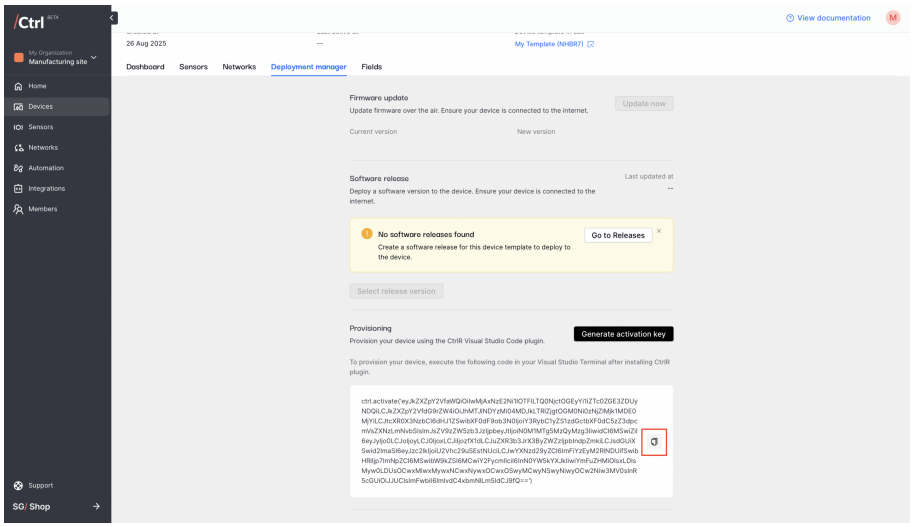
3. Open the CtrlR plugin in Visual Studio Code and click "Connect device" on the detected device.



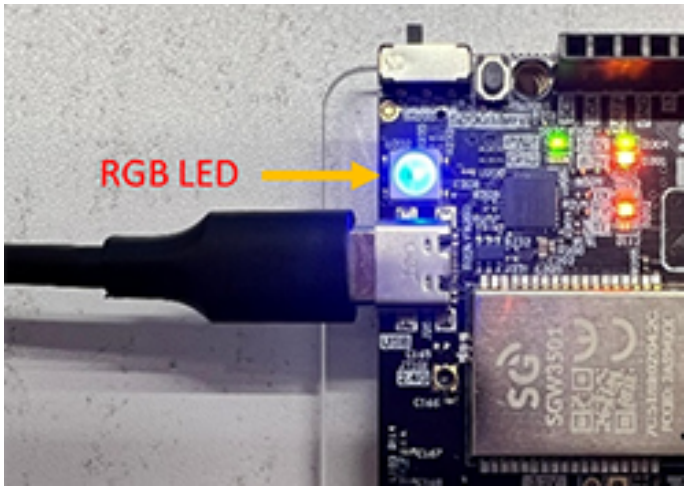
4. Invoke a terminal to access the REPL interface by clicking the “Create terminal” icon.

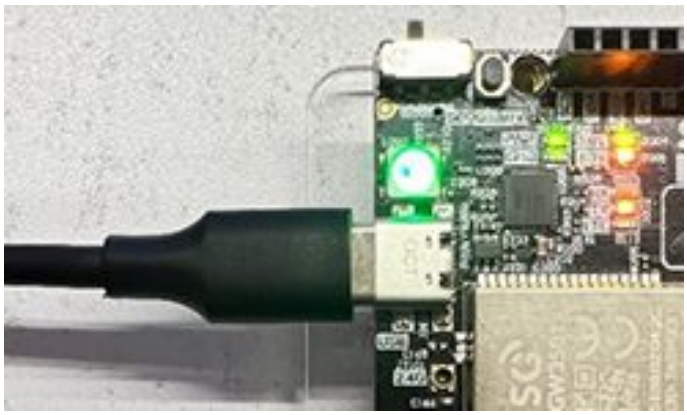


5. Remember the key you generated earlier? Copy it from your Ctrl “Deployment Manager”.
6. Paste it into the terminal, and press Enter.



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.

3.5 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.

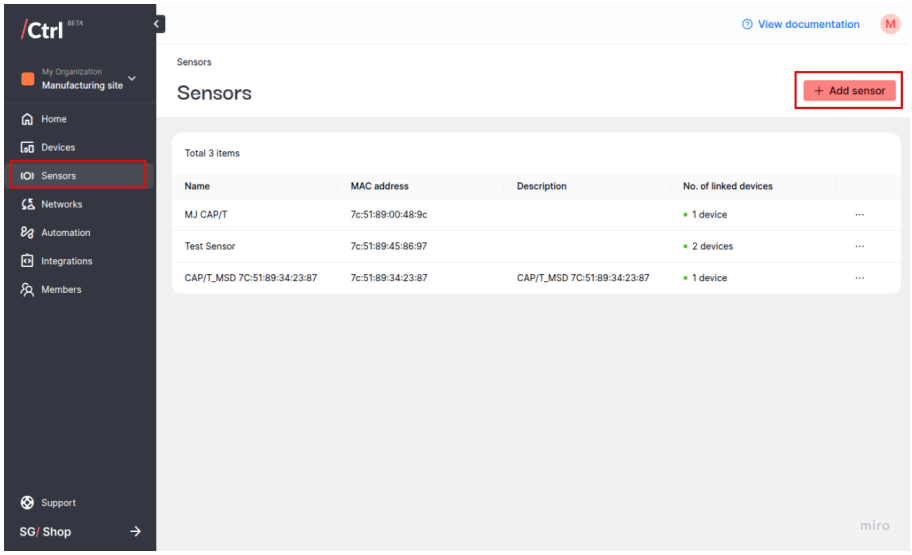
- *Add your Sensor*
- *Link your Sensor*
- *Map Sensor data*

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.

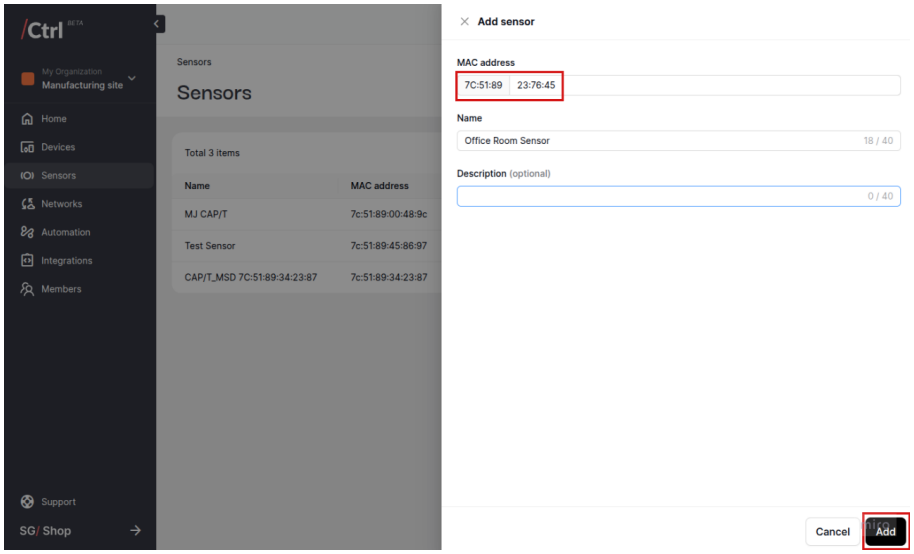
3.5.1 Add your CAP/T Sensor

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”.



3.5.2 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:

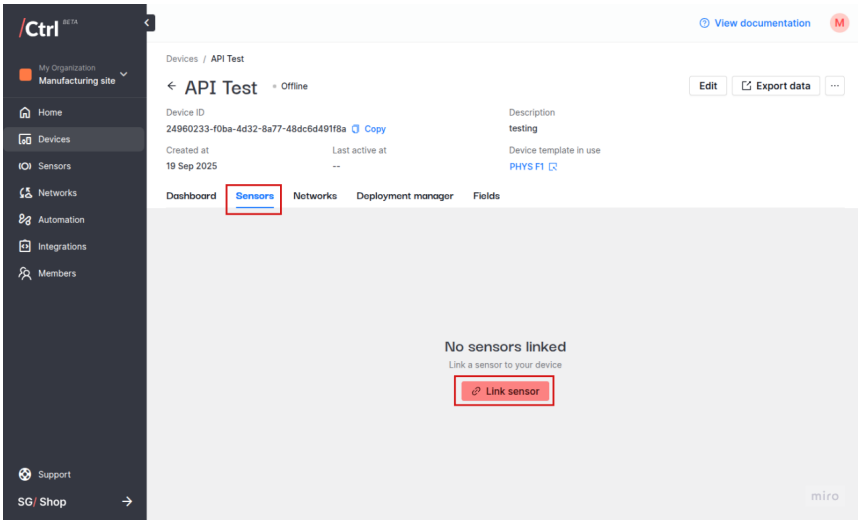
1. Since we are attempting to update the sensor configuration remotely to the Starter Kit, ensure that your Starter Kit has a stable Internet connection. Otherwise, you can regenerate the configuration key and run it through your CtrlR Visual Studio Code plugin after completing the “Map your CAP/T Sensor 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:

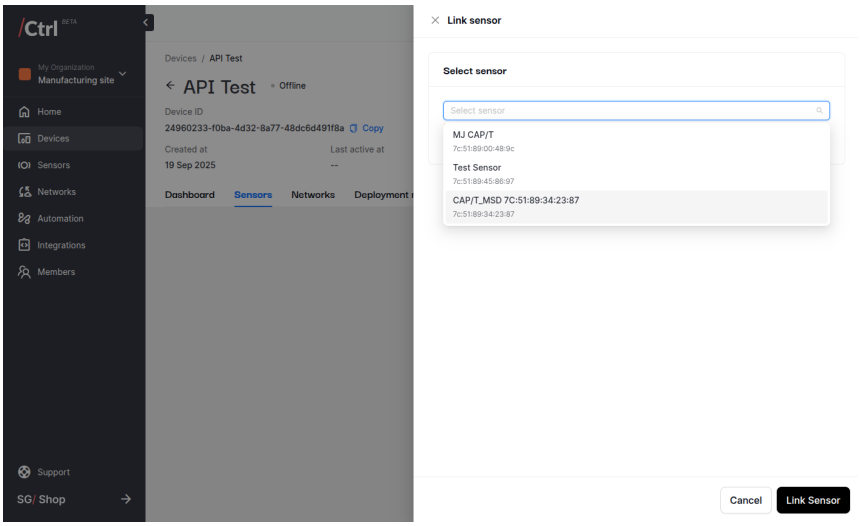
   Through "Devices":

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

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



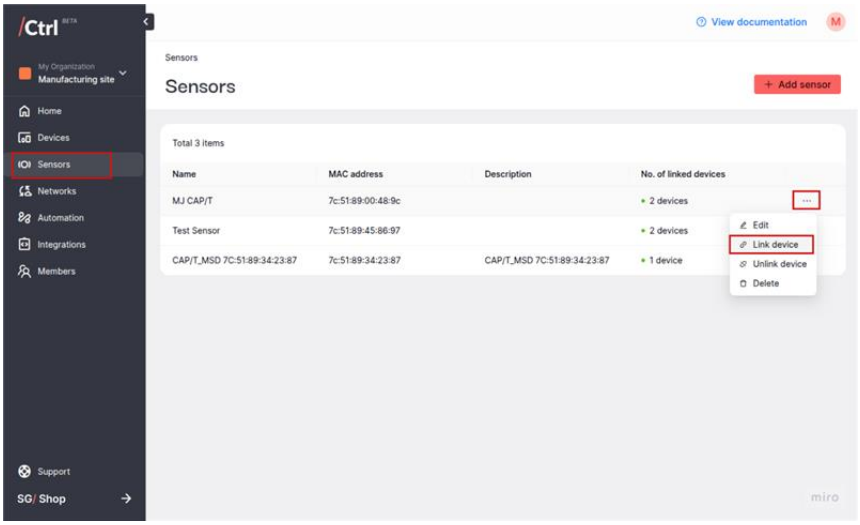
   c. Select a sensor to be linked.



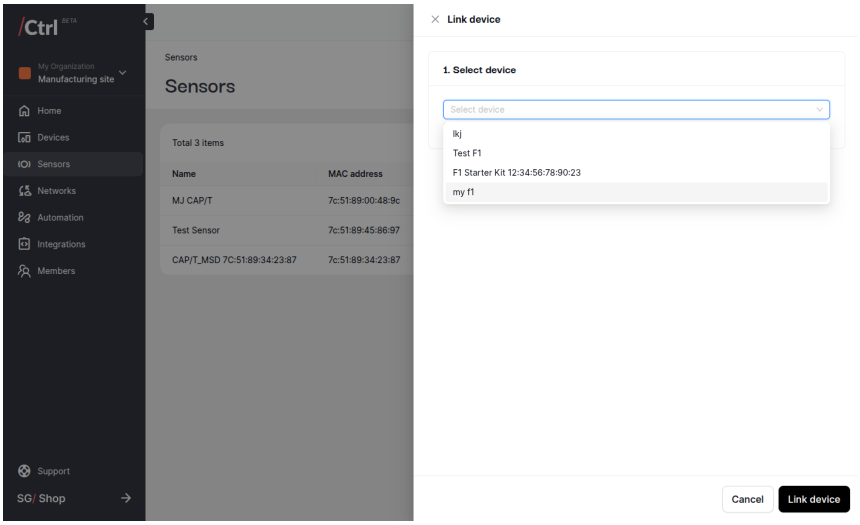
Through "Projects":

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.



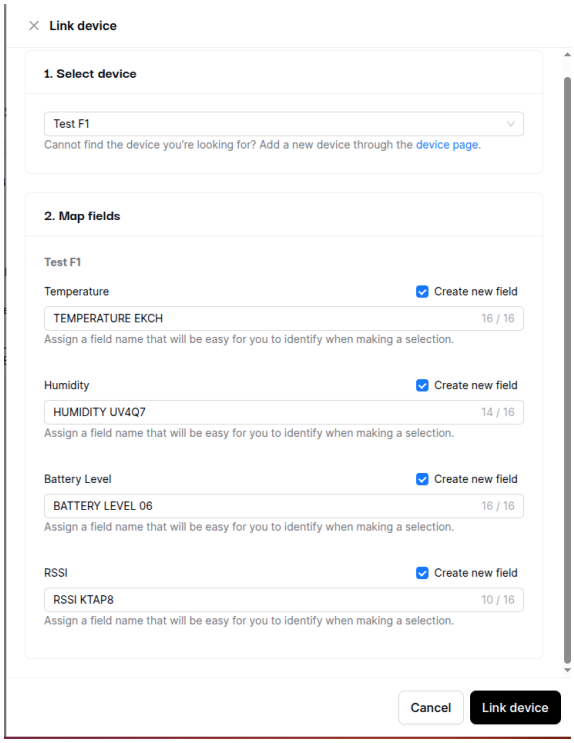
3.5.3 Map your CAP/T Sensor data

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

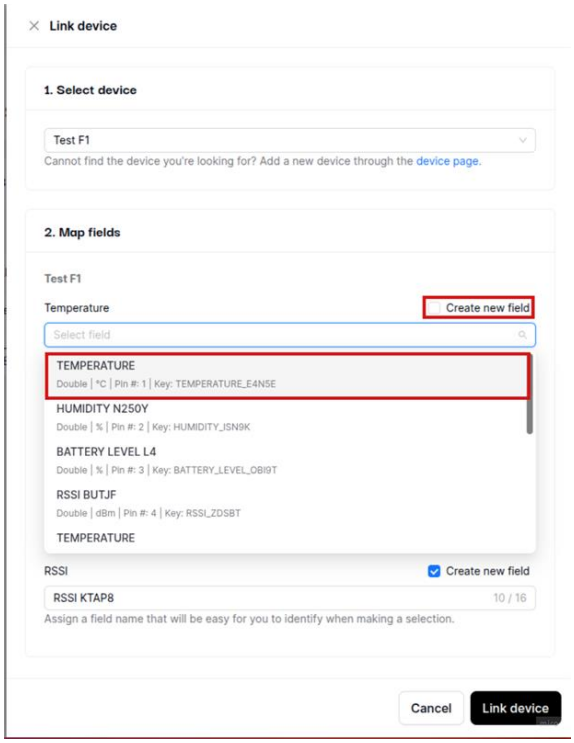
1. After you select a device or sensor to be linked, the field mapping options will be displayed.

   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:

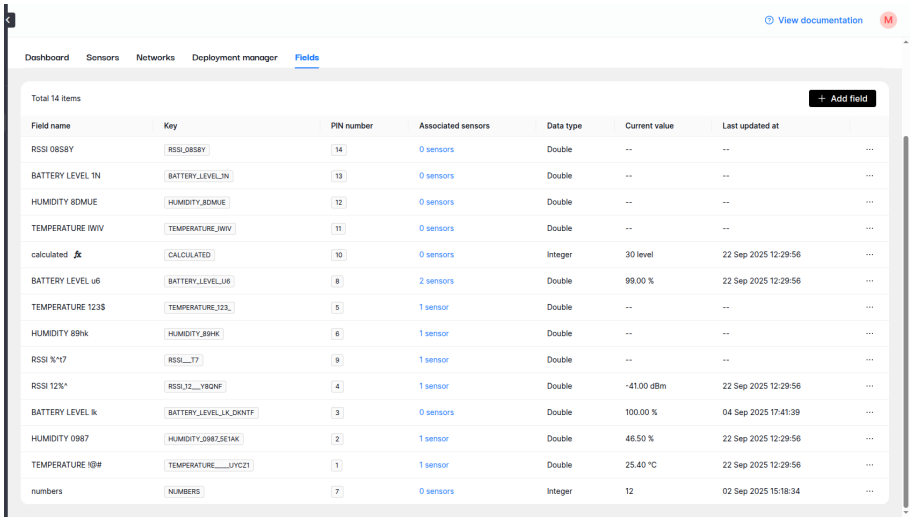
   - Temperature
   - Humidity
   - Battery Level
   - RSSI



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.



3.5.4 Next Steps

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

3.6 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.

3.6.1 Resetting the Device

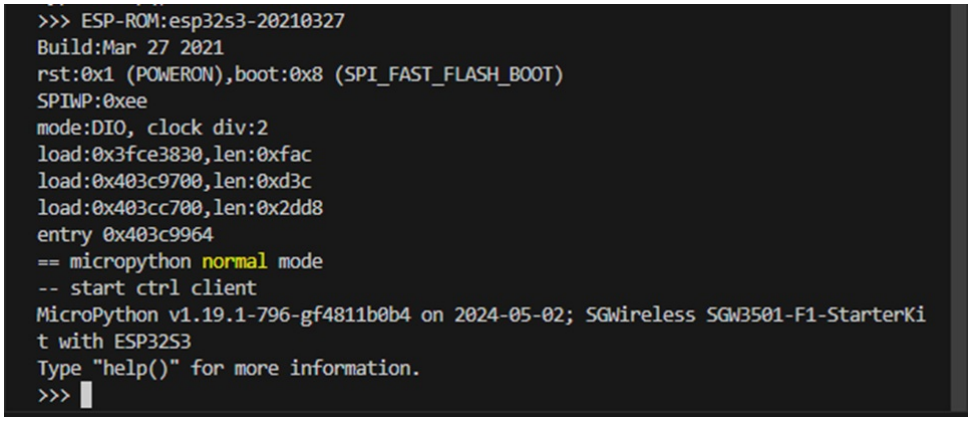
If something goes wrong, the device can be reset with two methods: hard reset and soft reboot.

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:



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:

CHAPTER

FOUR

TAKING CTRL.

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

- Visualize your sensor data
- Check the status of your entire deployment
- Distribute firmware updates on a scalable approach

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

4.1 Let's get started!

4.1.1 Managing Projects

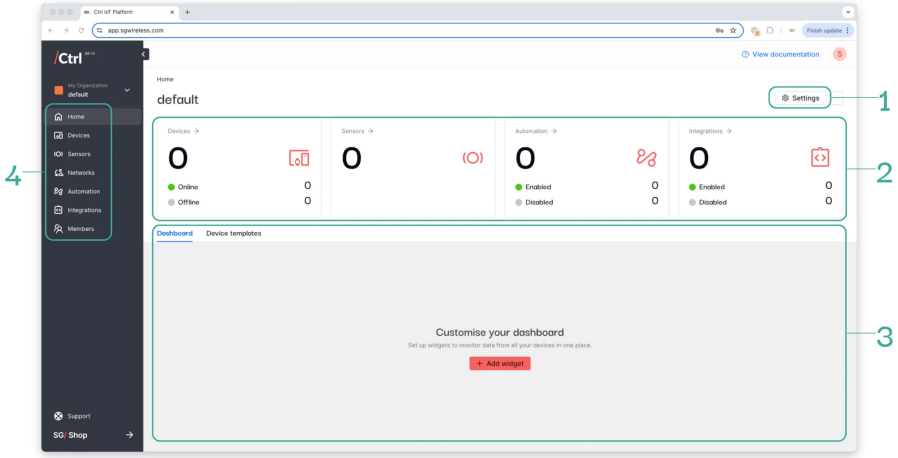
Your initial project is automatically generated when you create your account. You can establish multiple projects to organize different groups of IoT assets based on your requirements. Each project stores its IoT assets along with relevant information, such as templates, devices, sensors, network profiles,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

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

Project Components

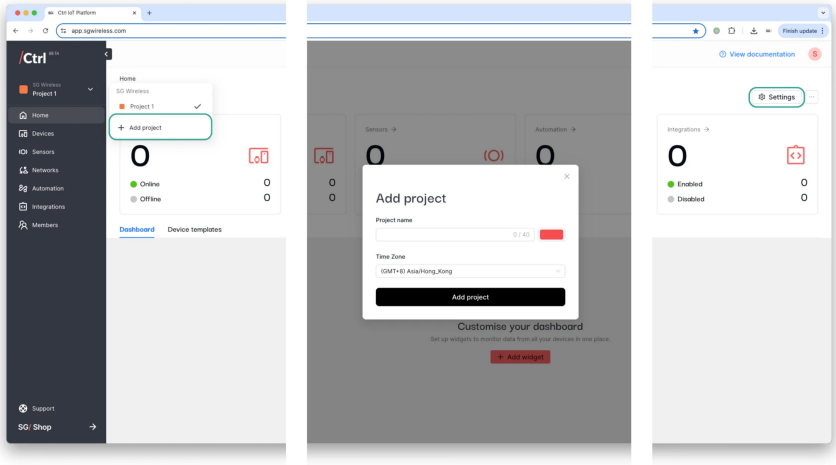
Your first project "default" is automatically created when you sign up for a Ctrl account. This is the project home page. Below are the components of a project:

- **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.
- **Customization:** Create dashboards and device templates for your use case.
- **Menu Bar:** Quick links to configure each Ctrl building block.



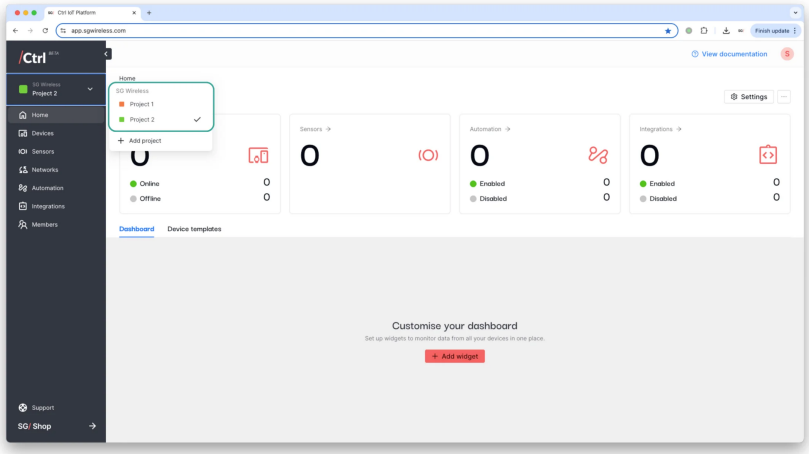
Add Project

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



Switch Project

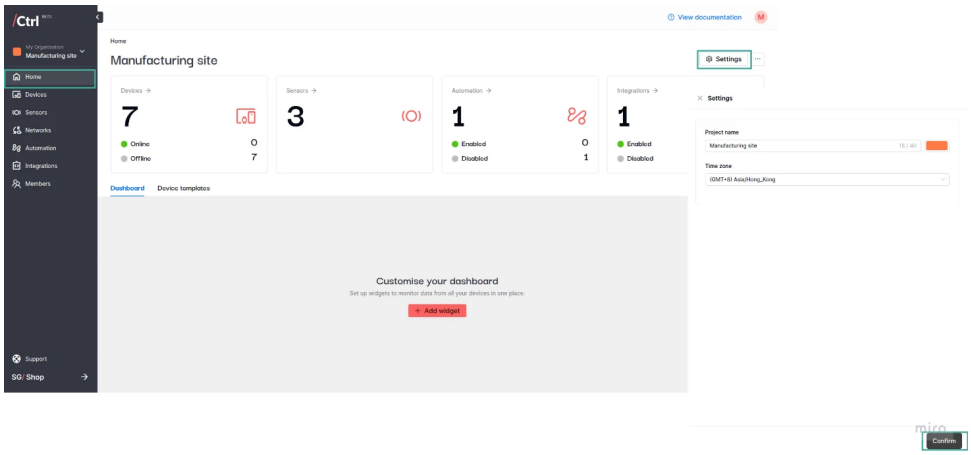
Switch between different projects simply by selecting it. The checkmark indicates the selected project.



Edit Project

Update your project settings through the following steps:

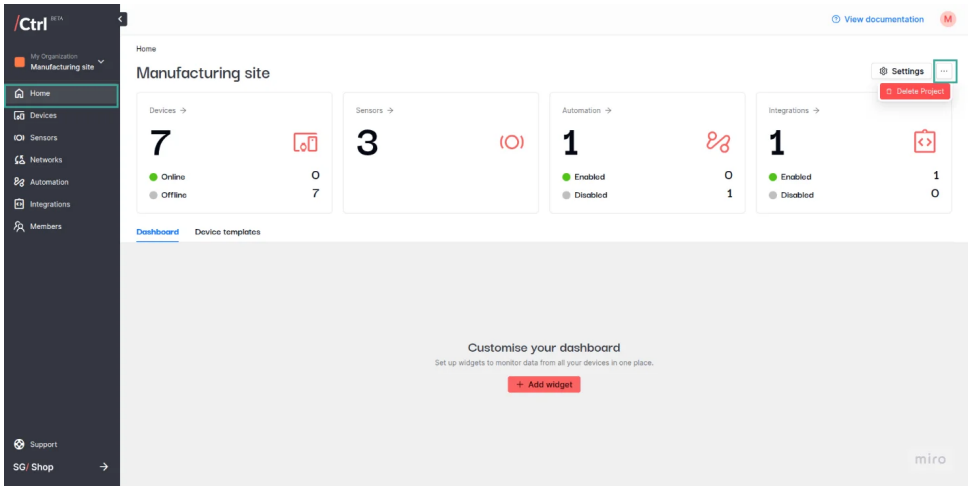
1. On the target project, go to the Home tab.
2. Click the “Settings” button on the top right corner.
3. Make necessary changes.
4. Click “Confirm” to apply the changes.



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.**



4.1.2 Managing 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.

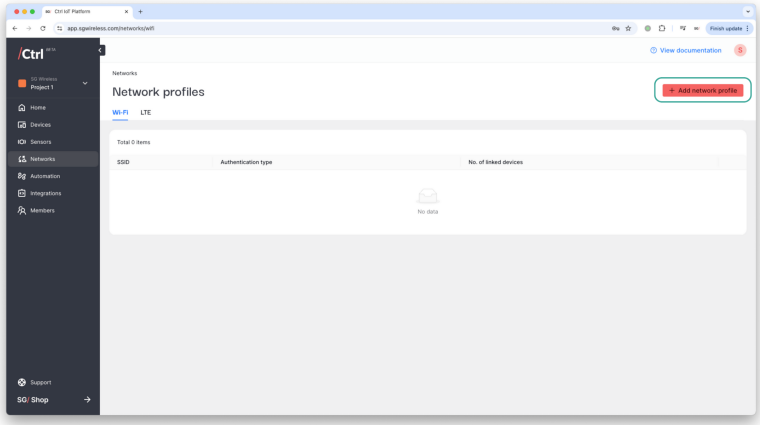
Manage Network Profiles

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

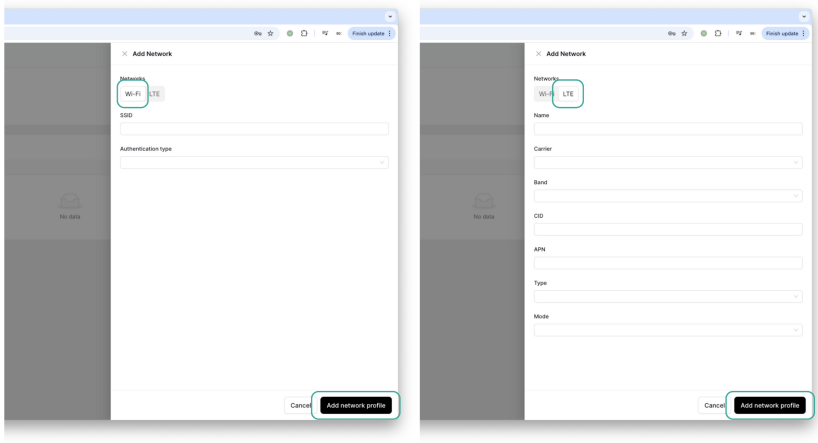
> **Note**
>
> If your device is provisioned via ZTP, an LTE profile will be automatically added for you, with name "1NCE".

Create new network profile

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



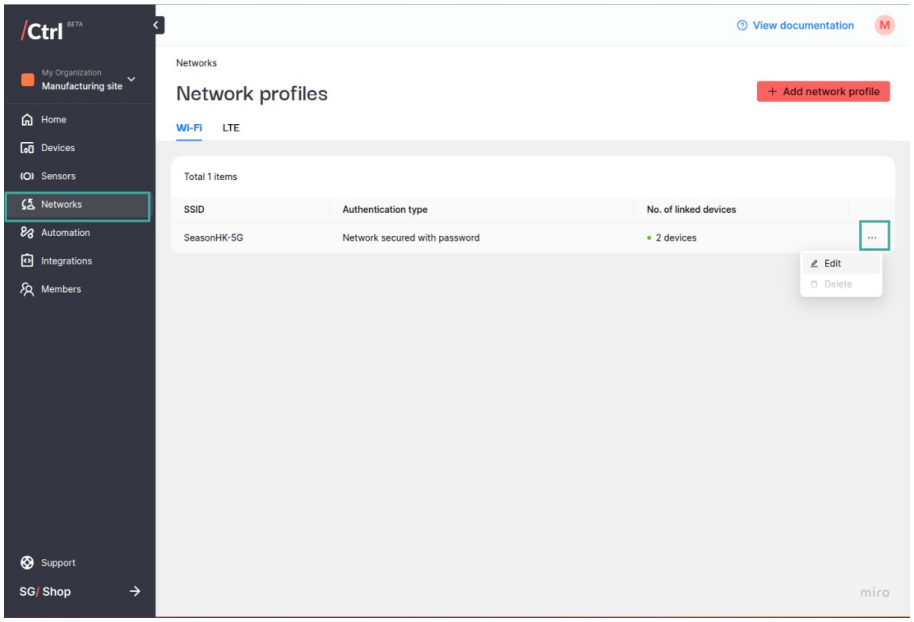
2. Choose "Wi-Fi" or "LTE", and enter the required network credentials.



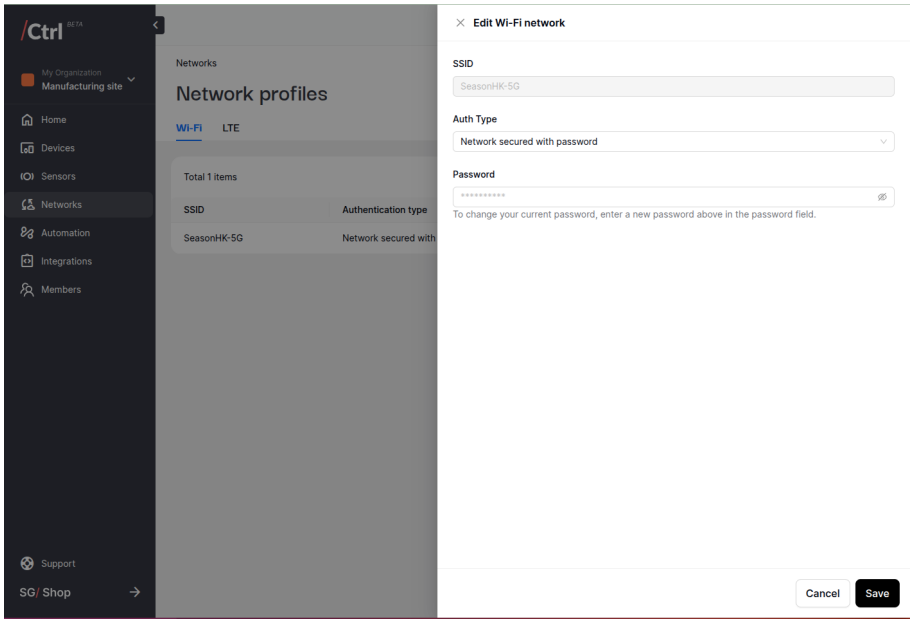
3. Click "Add network profile" once it's done. Your new network profile will be available under the corresponding tab.

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.



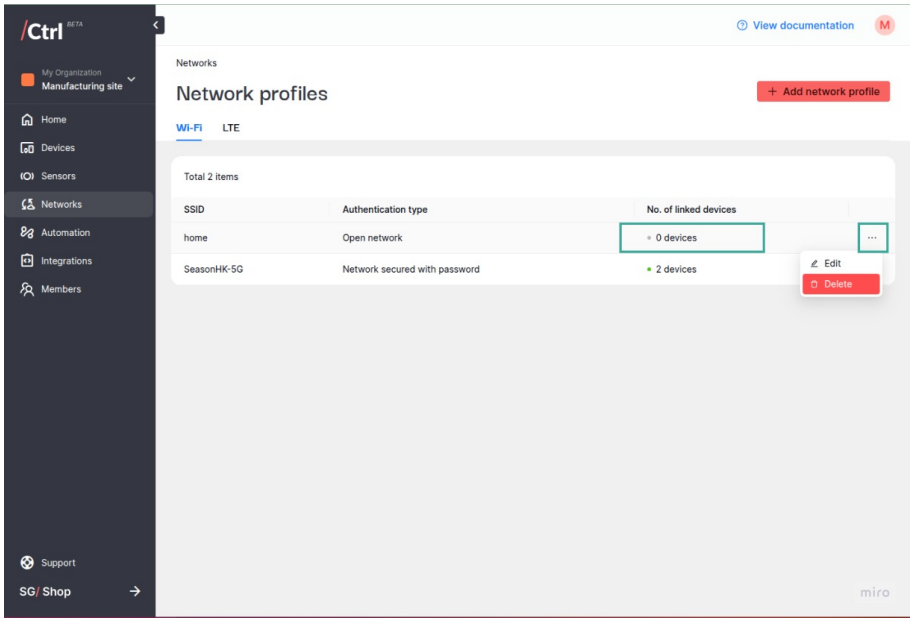
4. Click “Save” to apply changes.

> **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

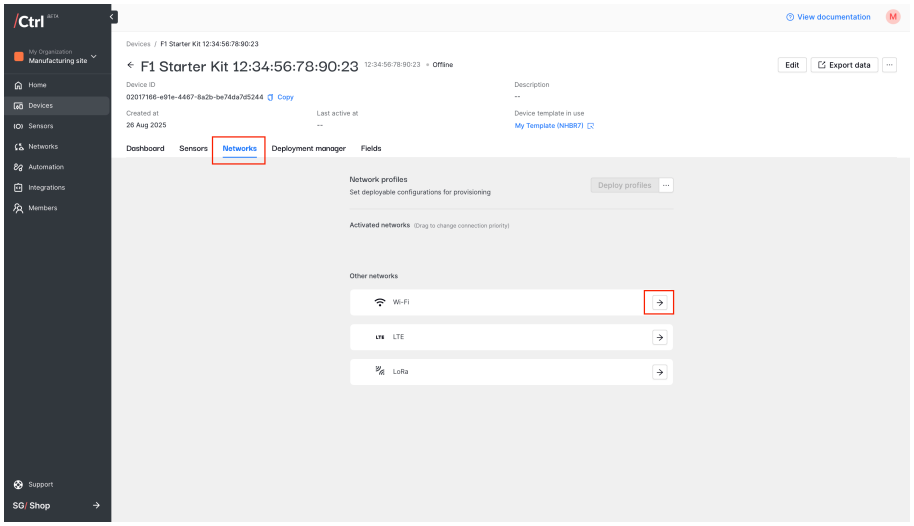
The F1 Starter Kit supports multiple network types that can be adjusted according to your need. You can *enable or disable* particular network types AND adjust the *network type priority* of your Starter Kit through Ctrl.

Accessing the device network settings

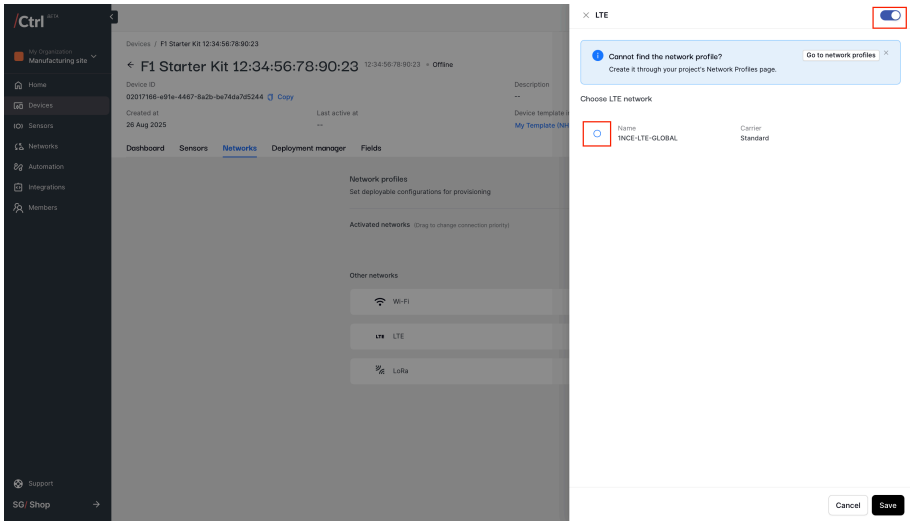
1. In the side menu, click “Devices”. Then, click on the target device.
2. Click on the device’s “Networks” tab.

Activate or deactivate a network

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



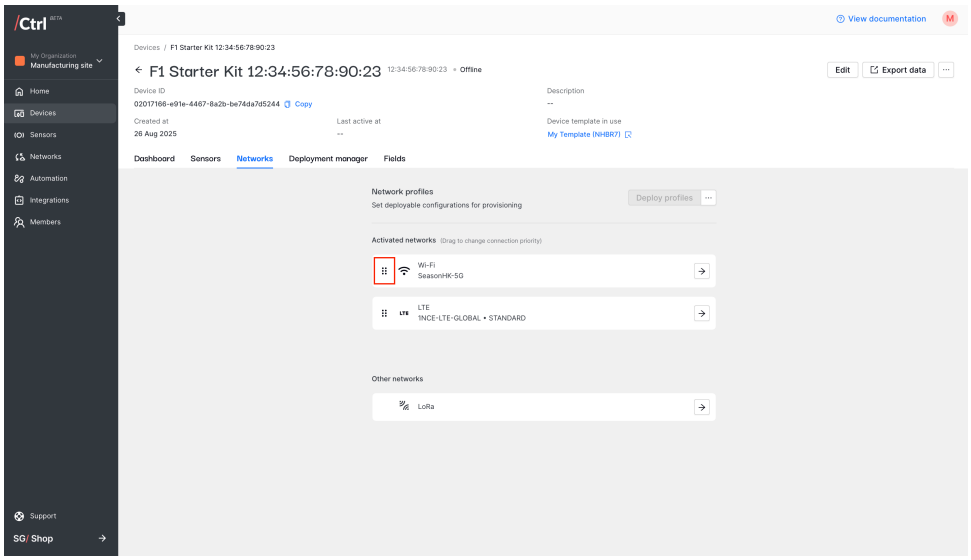
2. Press the toggle on the top right corner.



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.

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.

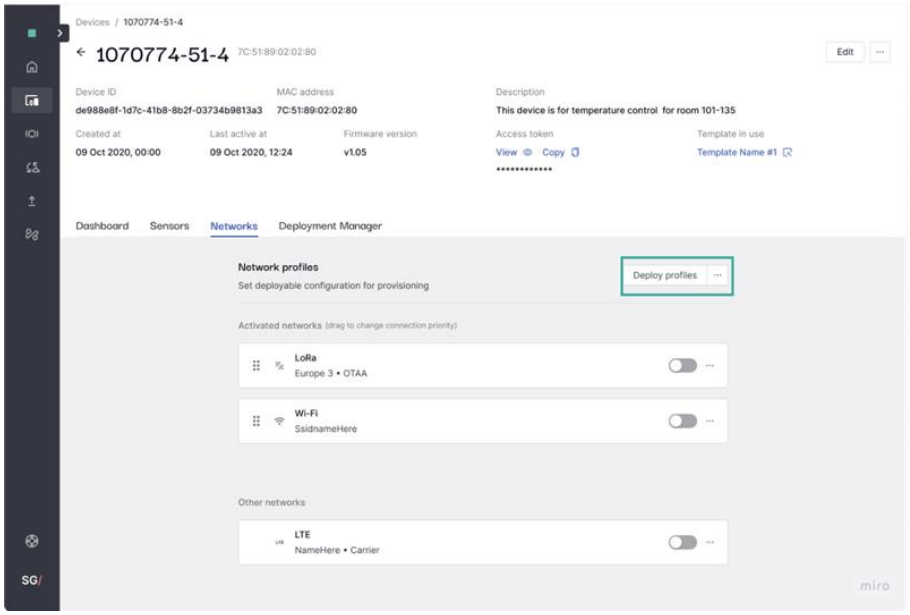


> **Note**
>
> This priority will only be used during the first-time connection. Automatic network switching is not supported after connection has been established.

Deploy the network setting

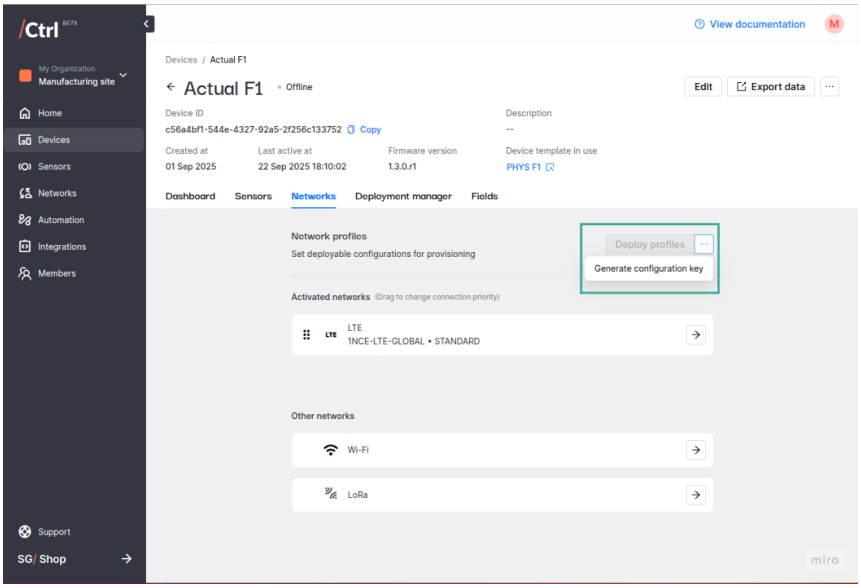
To implement the updated settings on your device, deploy it using one of the following approaches:

- **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*.

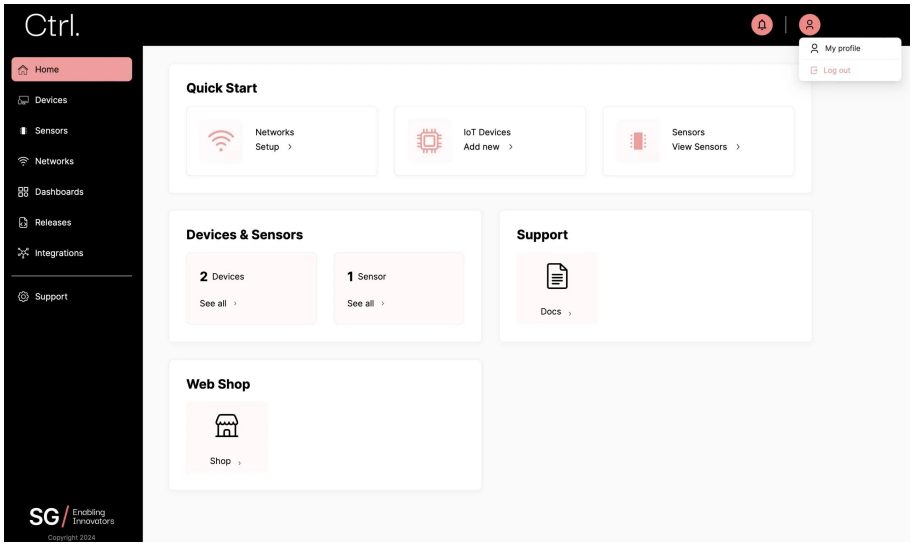
4.1.3 Account Settings

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.

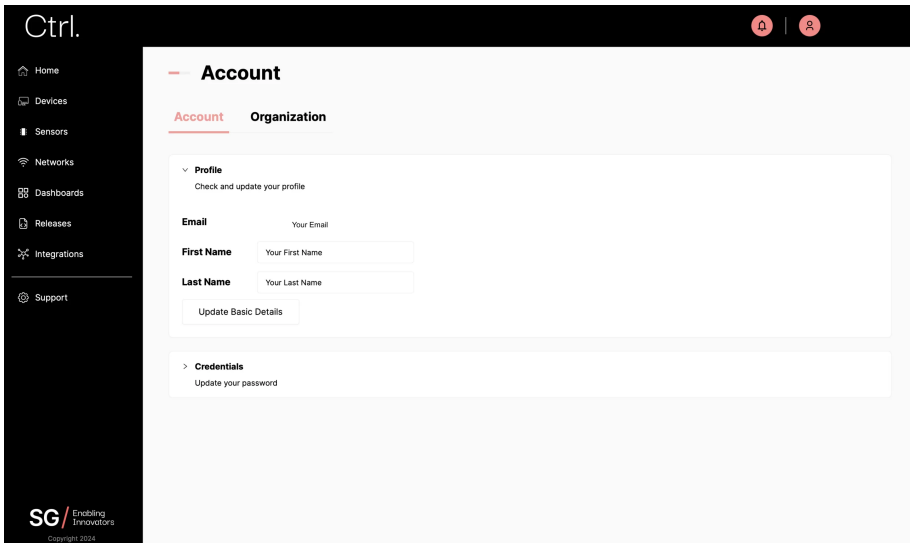
Update Personal Account Settings

Update Personal Profile

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

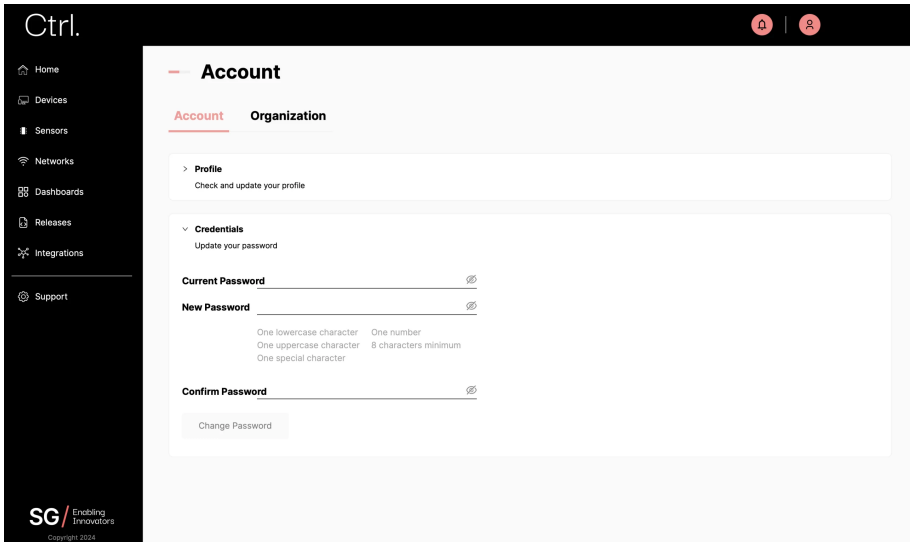


2. Under the `Account` tab, expand the `Profile` section. You can update your first and last name here. This change can be viewed by others in your organization as well.



Update Login Credentials

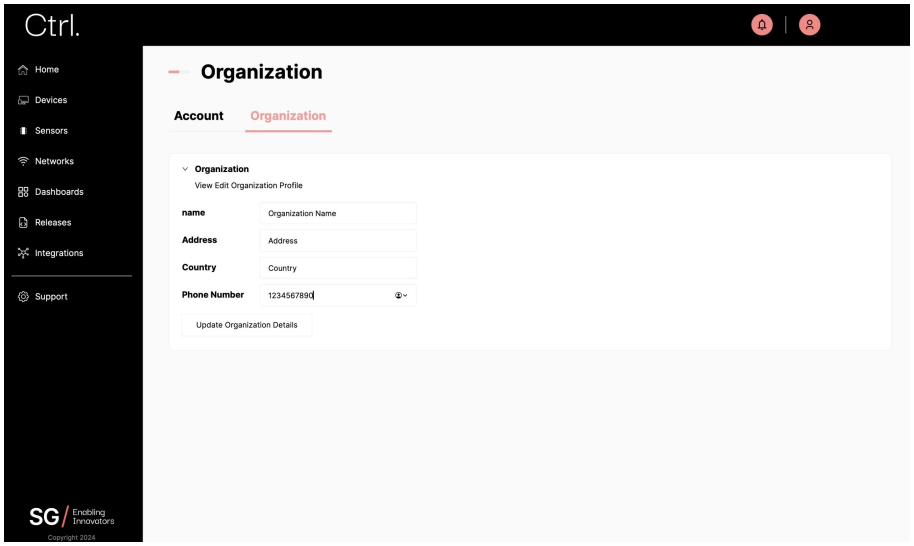
1. Click on the user icon at the top-right corner, and choose `My Profile`.
2. Under the `Account` tab, expand the `Credentials` section. You can update your password to Ctrl here. It is recommended for you to change your default password for safety reasons.



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.



4.1.4 Managing 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

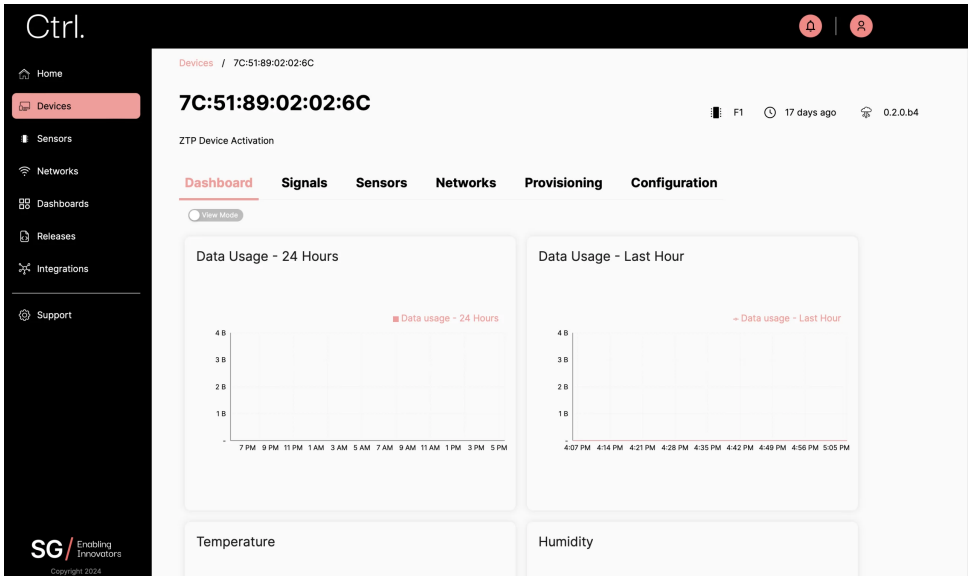
Next Steps

- *Visualize your signal data*
- *Link sensors to your device*

4.1.5 Visualize Data

By default, two line charts have been created for you. These show data usage for the last hour and the past 24 hours. In this section, we will explain how to see the data coming from your device on Ctrl.

> **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.



Step 1: Set up a CtrlR Project

Create a project in CtrlR called `CtrlR_signals`, and add the following code to `main.py`. This Python application will send data every 10 seconds to Ctrl.

```
import time
import math

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

Press the **Upload** button to upload the code into your device.

Step 2: Monitor a signal from your device

1. Go to the Ctrl device page and select your device.
2. Then go to the `Signals` tab and view signals received.
3. Select a signal number to view message history.

Done!

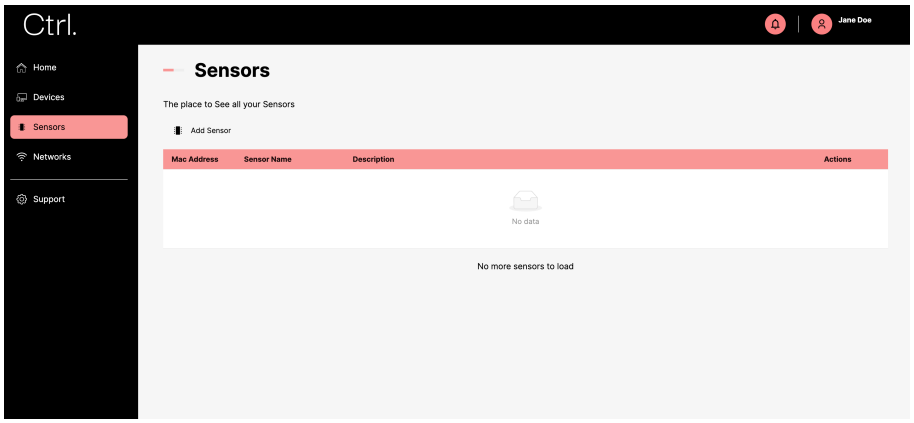
Now you've learned how to view your data on the device.

4.1.6 Sensors

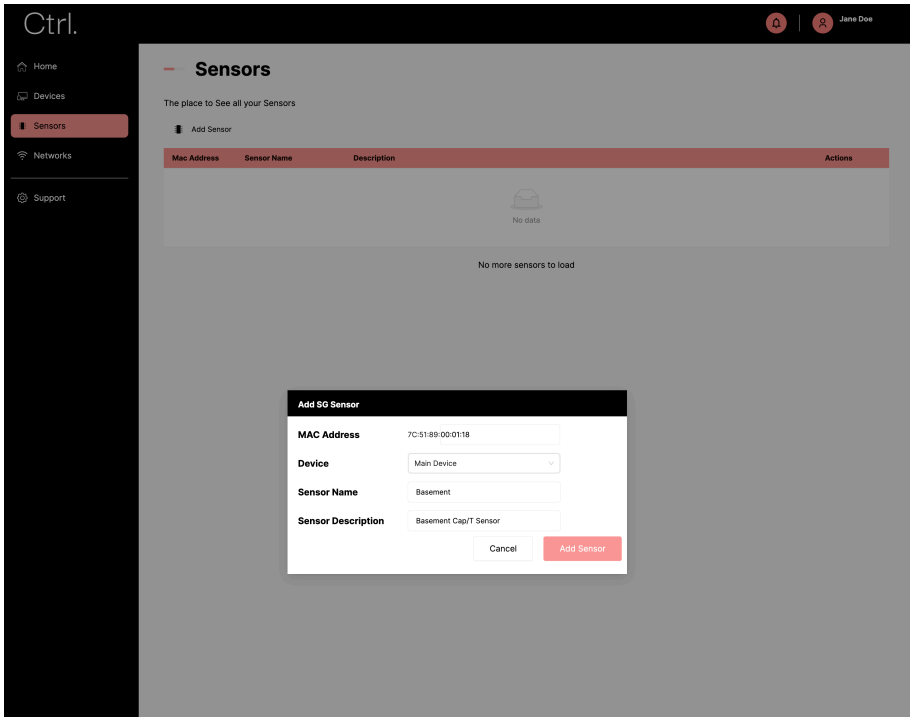
Add a Sensor to Ctrl

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

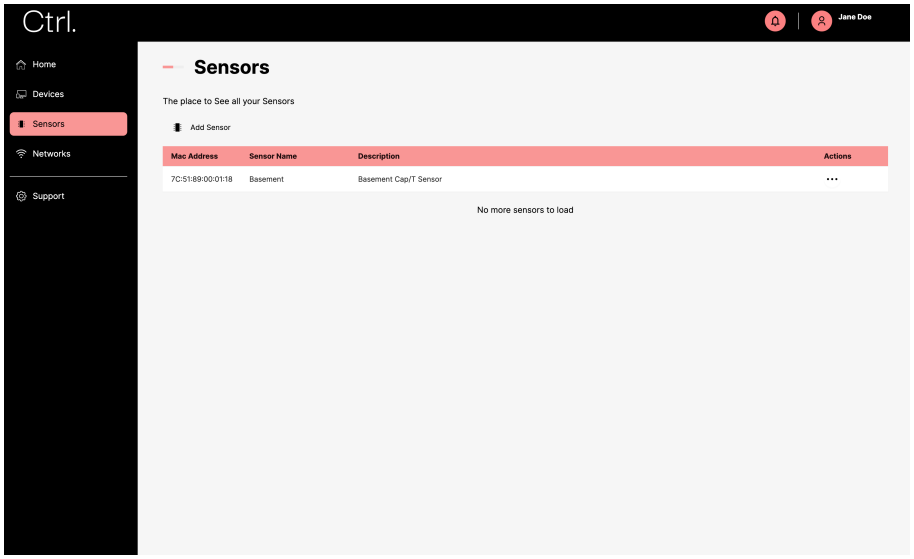
1. Navigate to **Sensors** and click on `Add Sensor`.



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 `Sensor Name` and `Sensor Description` to better identify it.



5. The new sensor will appear on the `Sensors` page and in the linked device's `Sensor` tab.

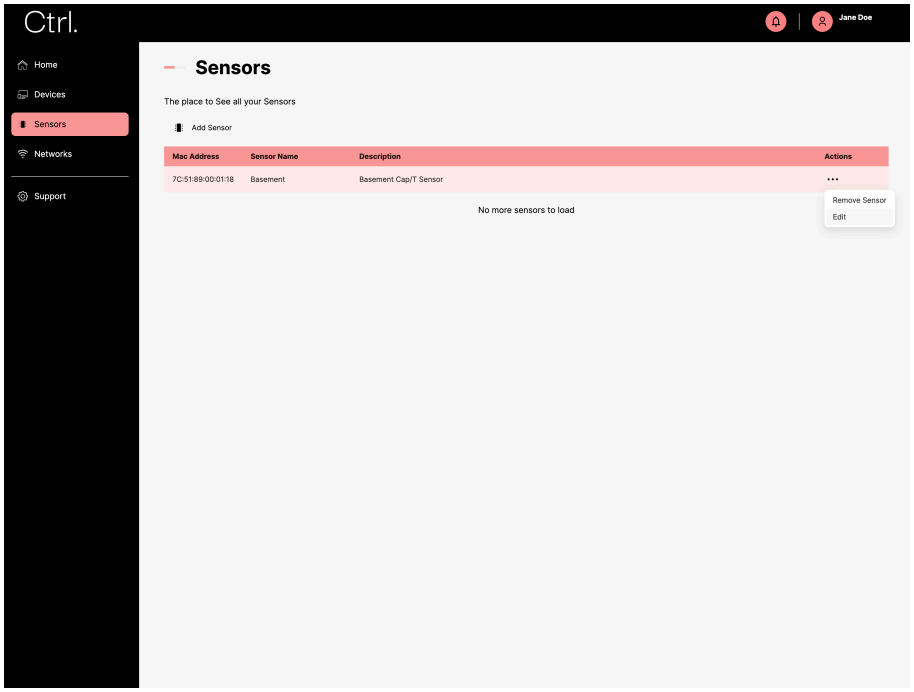


Each sensor can be linked to multiple devices, allowing for versatile integration. The data collected by the sensor will flow through the connected devices and ultimately reach Ctrl.

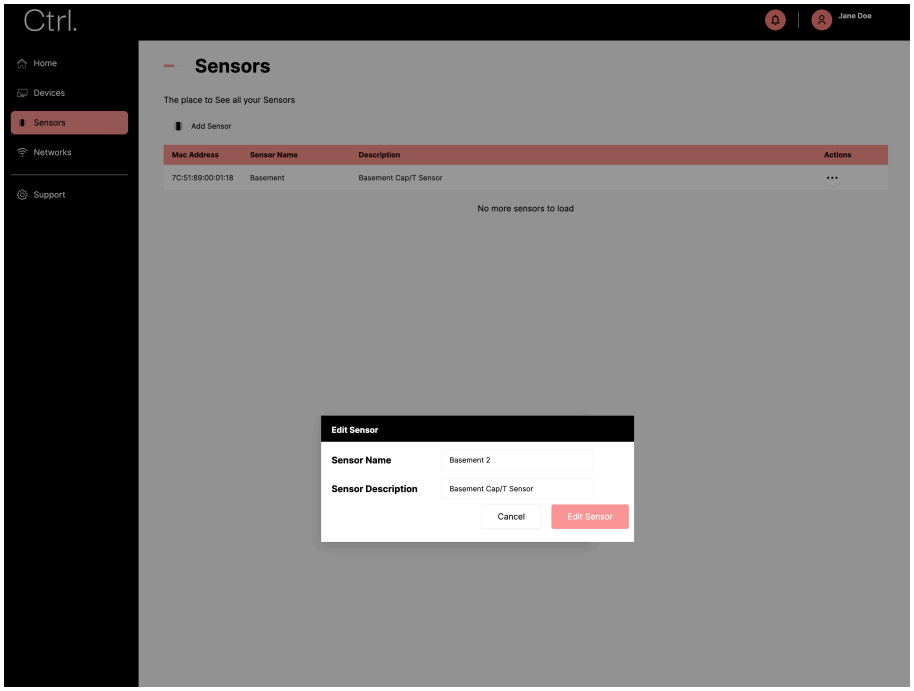
On your Dashboard, there should now be a Temperature and Humidity widget. These will update when the device sends sensor data to Ctrl. Multiple sensors will show up in a different colour with their name in the legend.

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.

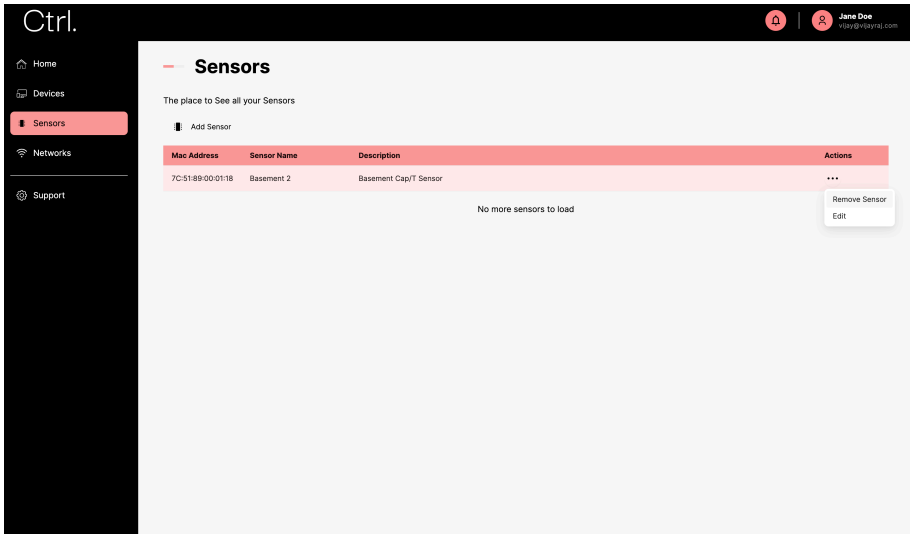


4. Update the `Name` or `Description` fields as needed.
5. Save your changes by clicking `Edit Sensor`.

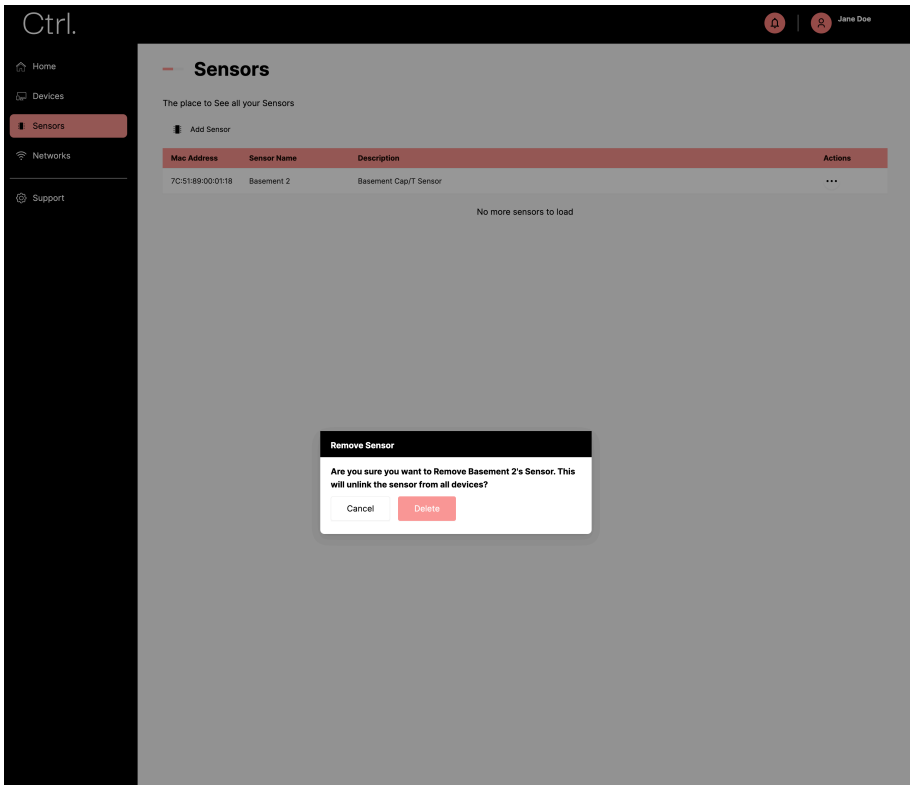


Remove a Sensor from Ctrl

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



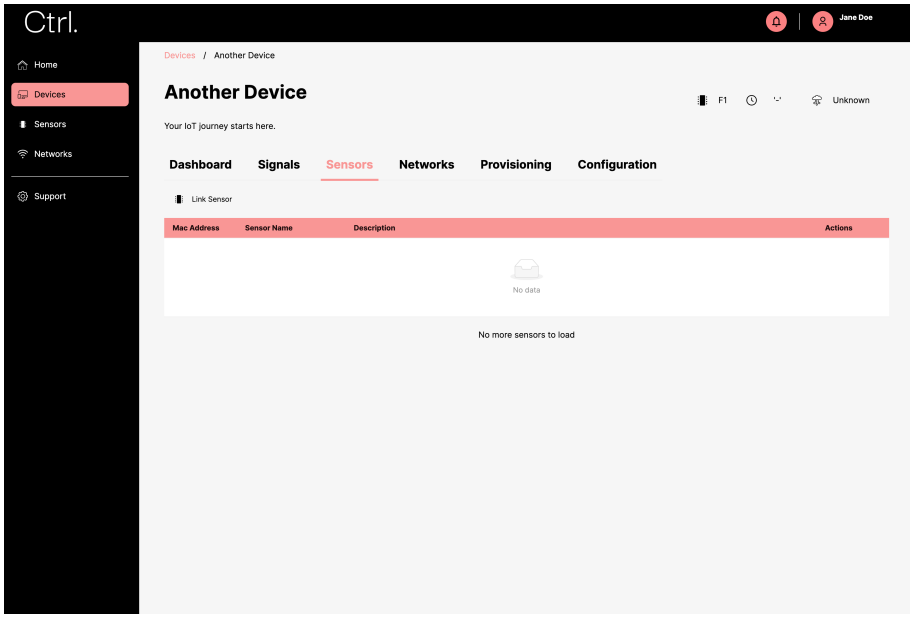
4. Confirm by clicking `Delete`.
5. Deleting will unlink sensors from any linked devices and remove the sensor from the platform.



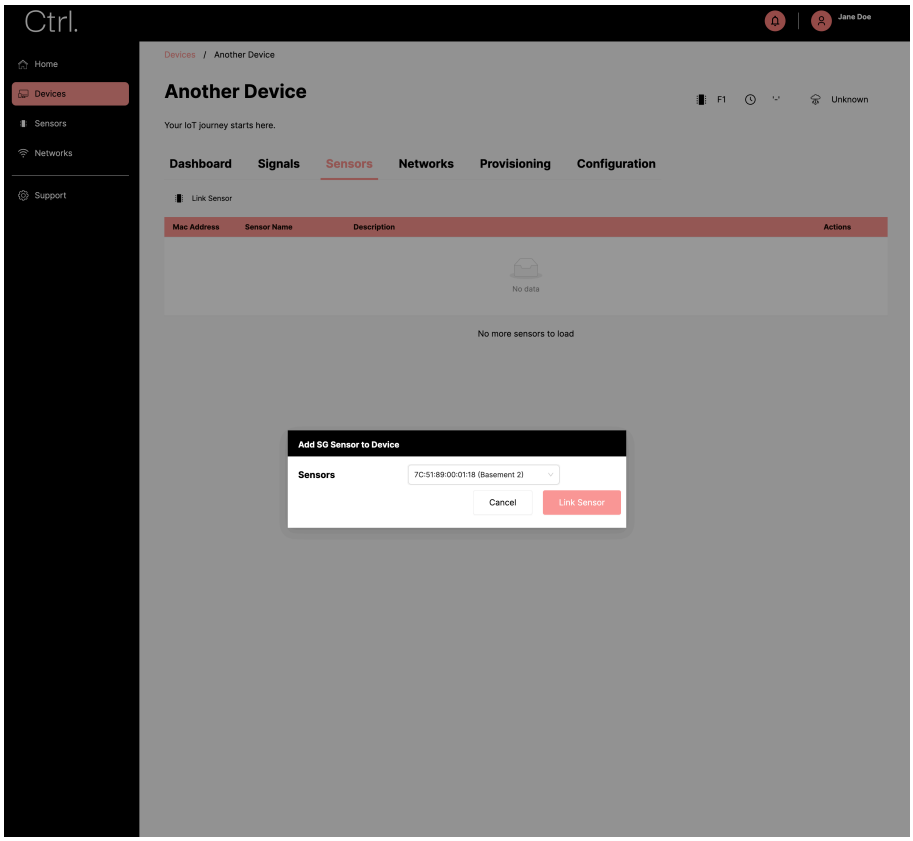
Link Sensor to a Device

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 Sensor` button to confirm.

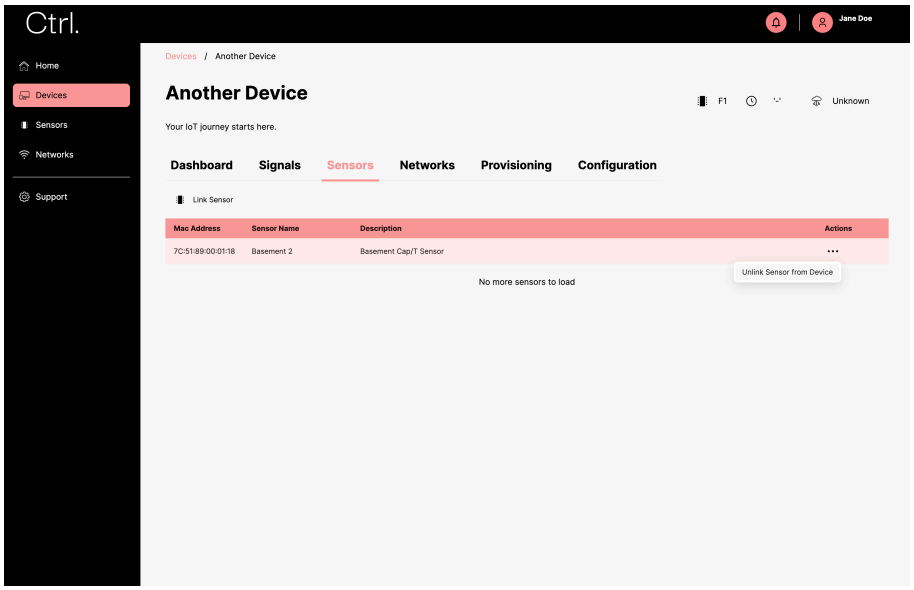


6. Ctrl will send a message to the device, instructing it to listen for the sensor's MAC address and establish the link.

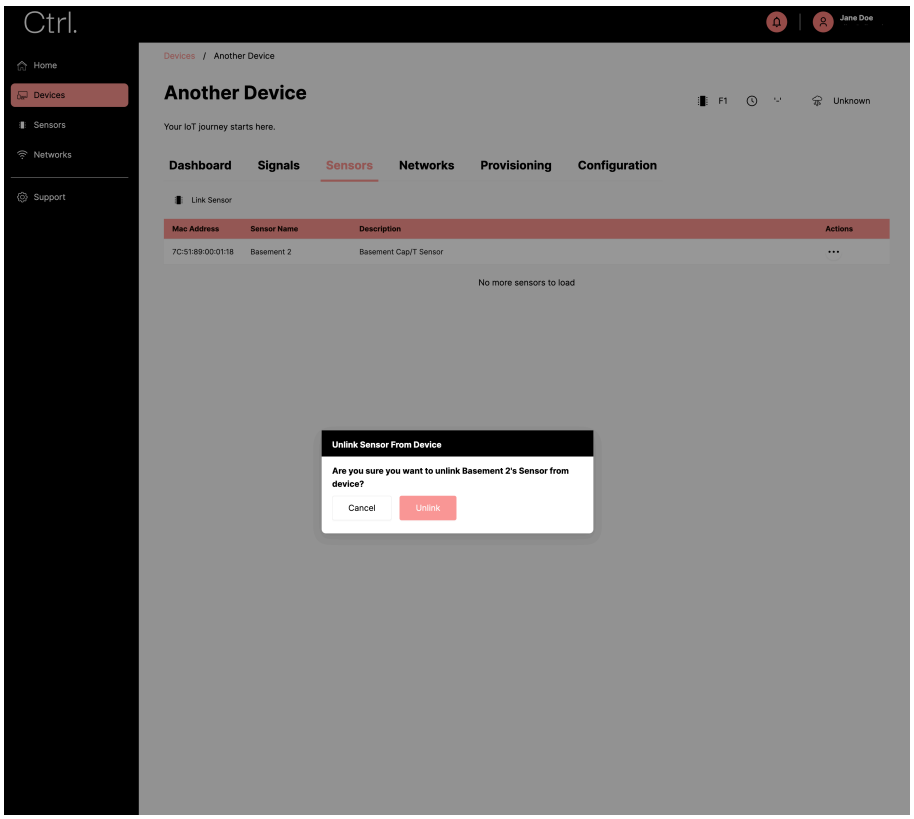
Unlink Sensor from a Device

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 Sensor from Device` option from the dropdown menu.



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



4.1.7 Integrations

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

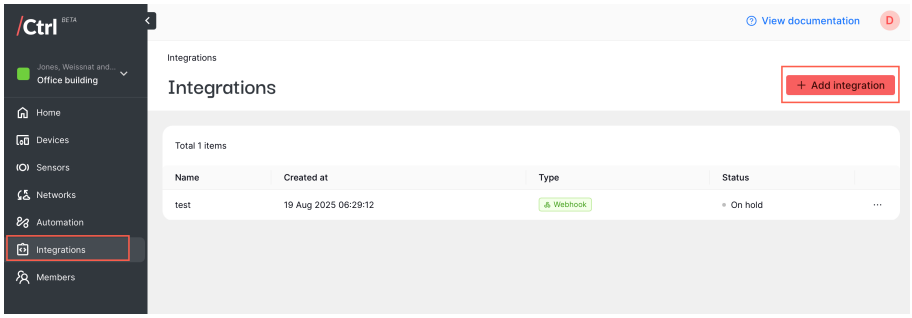
- *Webhook* – allows for user-defined HTTP callbacks to a defined remote destination. All elements of the requests (headers, query string parameters, etc.) are customisable.
- *AWS IoT Core* – a managed cloud platform that lets connected devices easily and securely interact with Cloud applications and 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

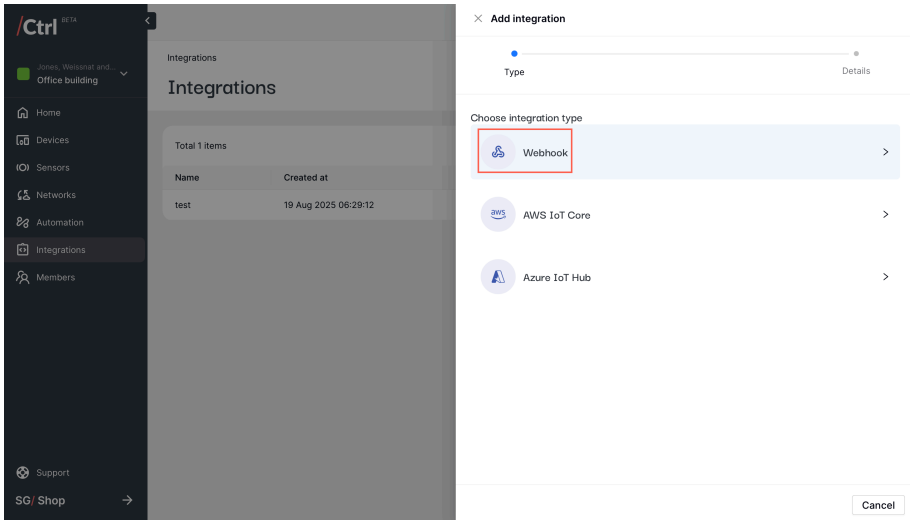
Whenever one of your integrated devices sends a signal to Ctrl, we perform an HTTP request defined by the user. You can use some presets (`DEVICE_TOKEN`, `USER_ID`, etc.), which will act like placeholders and will be dynamically replaced at the moment of performing the request with the relative content.

Setting up your integration

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

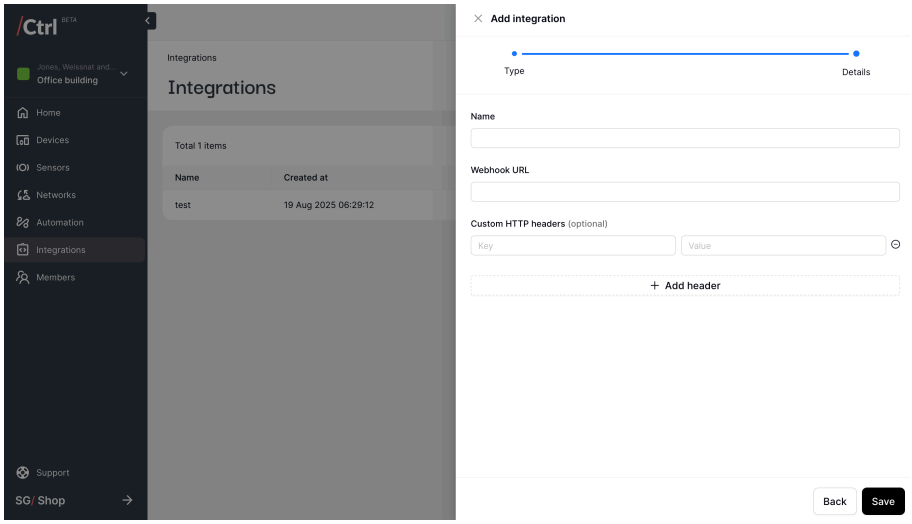


2. Select **Webhook** as the integration type.



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/.



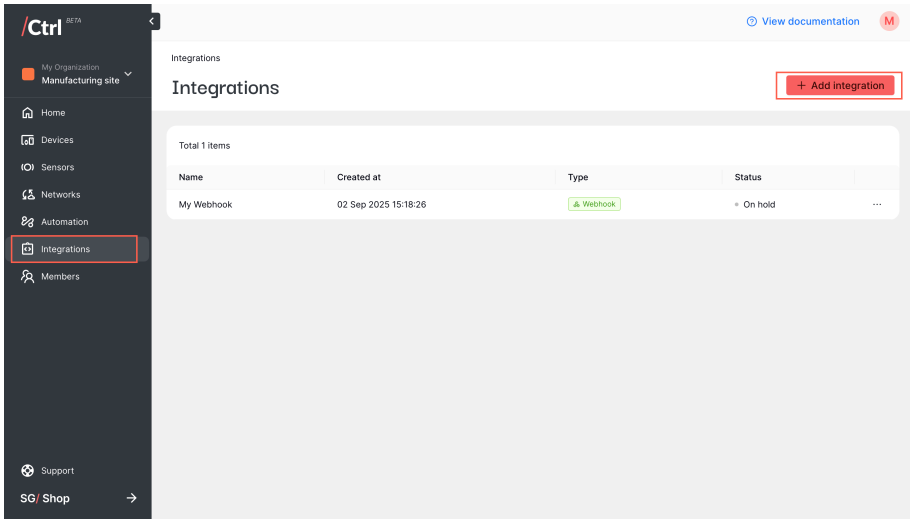
4. Click “Save” when you are done. You should be able to see the incoming Ctrl telemetry data on your destination system.

AWS IoT Core

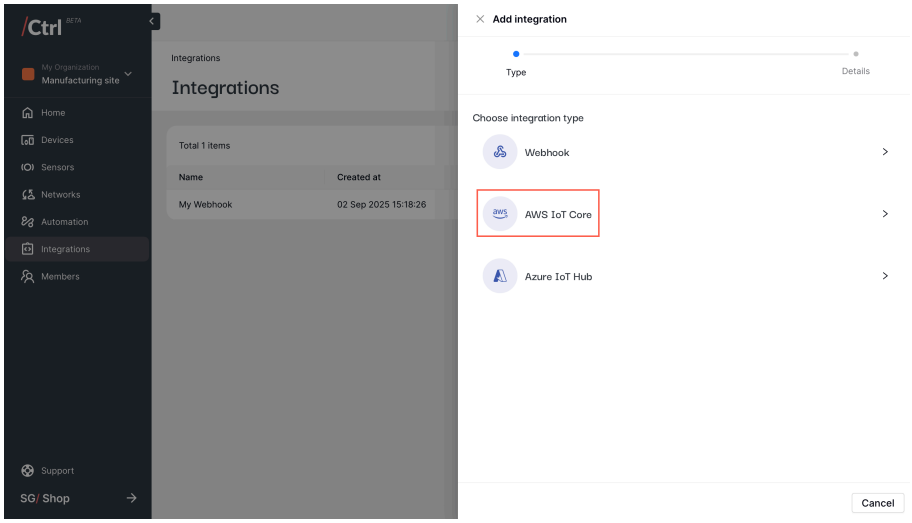
Whenever one of your integrated devices sends a signal to our broker, we republish the binary payload to the endpoint specified for its integration through MQTT connection.

Setting up your integration

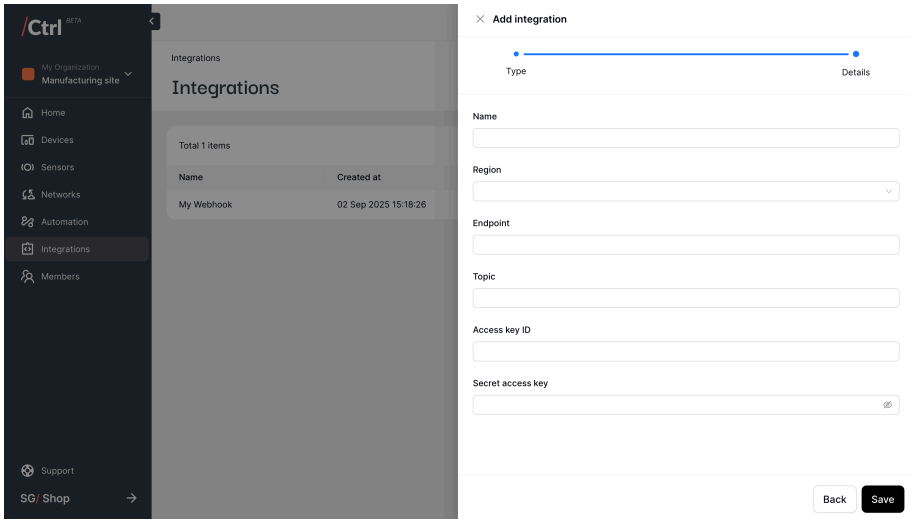
1. Click **Integrations** on the side menu, then click “Add integration” on the top right corner.



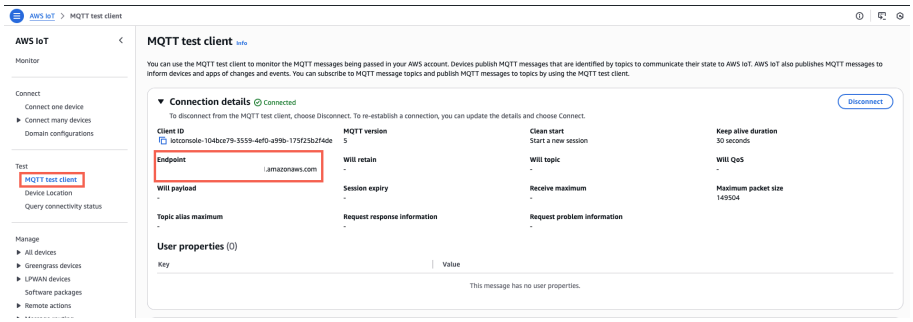
2. Select **AWS IoT Core** as the integration type.



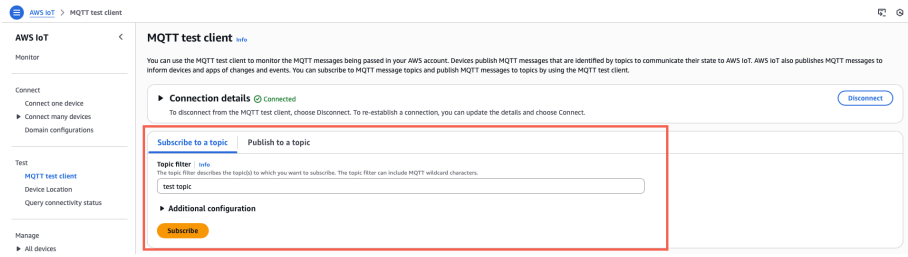
3. Complete your AWS IoT Core information.



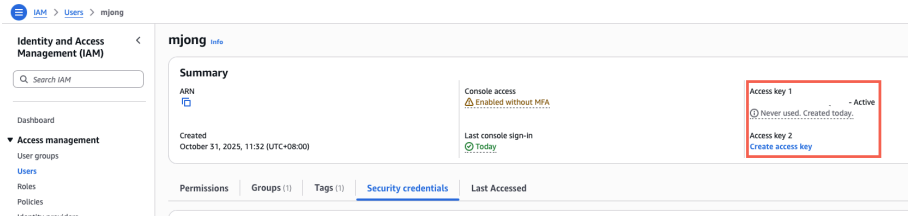
- **Name:** User-friendly name to identify a specific integration config.
- **Region:** Region of your AWS instance.
- **Endpoint:** AWS IoT Core MQTT endpoint found on the “Connection details” section on the upper part of the “MQTT test client” tab.



- **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.

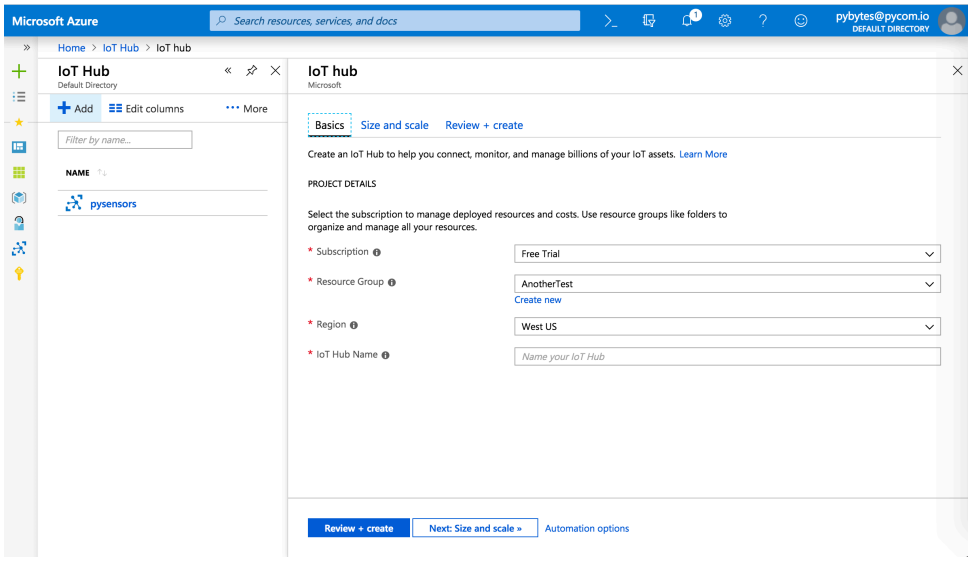
Azure IoT Hub

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

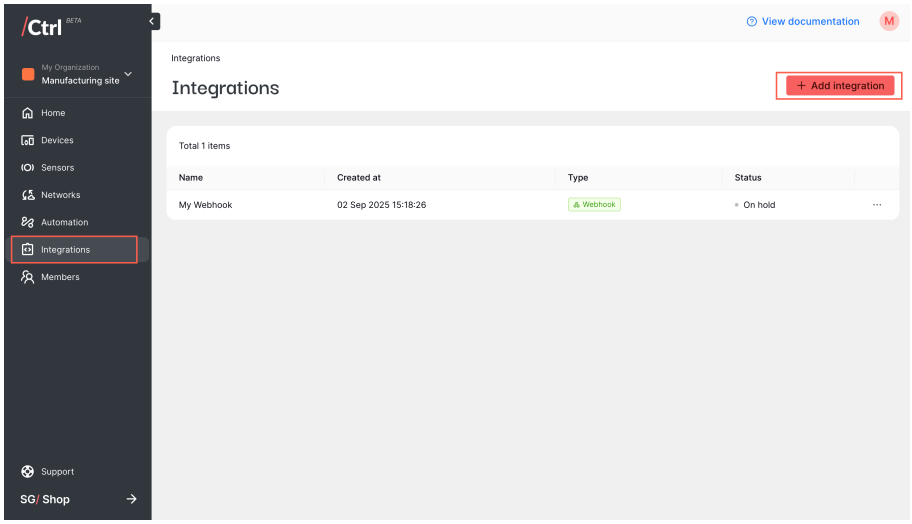
The first step requires you to create an IoT Hub. This is an Azure service that enables you to gather high volumes of telemetry data from your IoT devices. It then moves them into the cloud for storage or processing. In order to do that, follow the official documentation. To summarise, you’ll need to:

- 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.

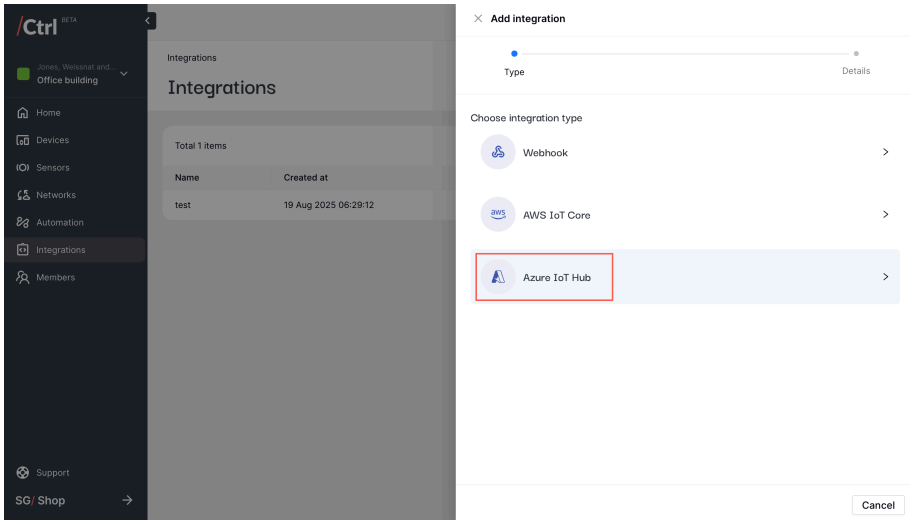


Setting up your Ctrl integration

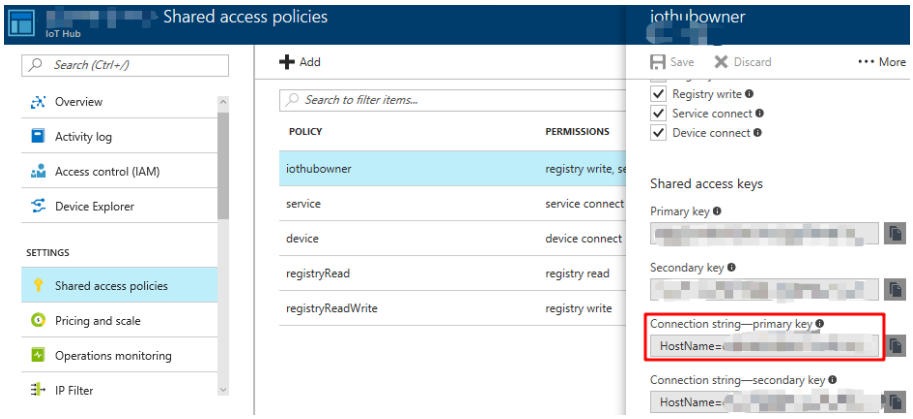
1. On your Ctrl page, click **Integrations** on the side menu, then click “Add integration” on the top right corner.



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

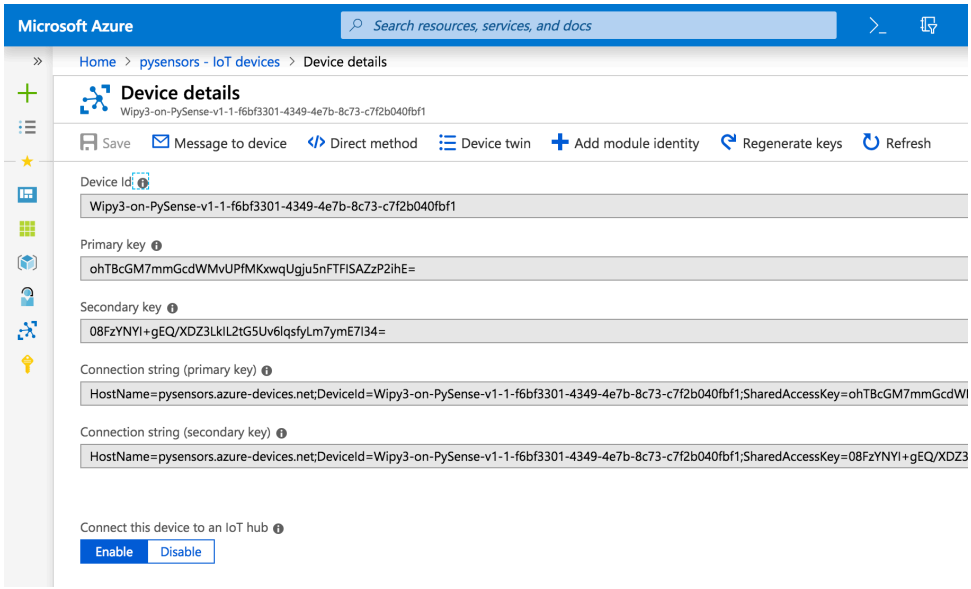


3. Add a user-friendly name to identify the integration instance, then copy and paste the connection string of your IoT Hub to the Ctrl form.



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.

CHAPTER

FIVE

CUSTOM PROJECTS

Getting creative and creating your own IoT networks.

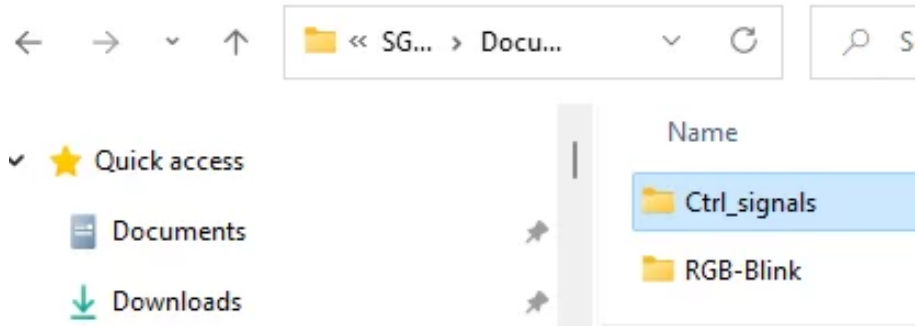
- *Creating custom projects* – set up a CtrlR project and upload data to Ctrl.
- *Programming the basics* – learn to control the on-board RGB LED using MicroPython.
- *Setting up custom networks* – configure LTE connectivity with custom SIM cards.

5.1 Creating Custom Projects

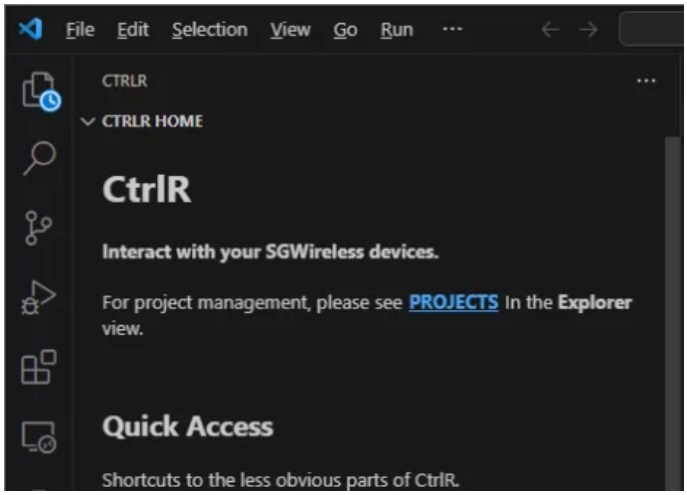
The VS CtrlR Plugin enables the F1 Starter Kit to work with third-party sensors to create custom projects on the Ctrl Cloud Platform.

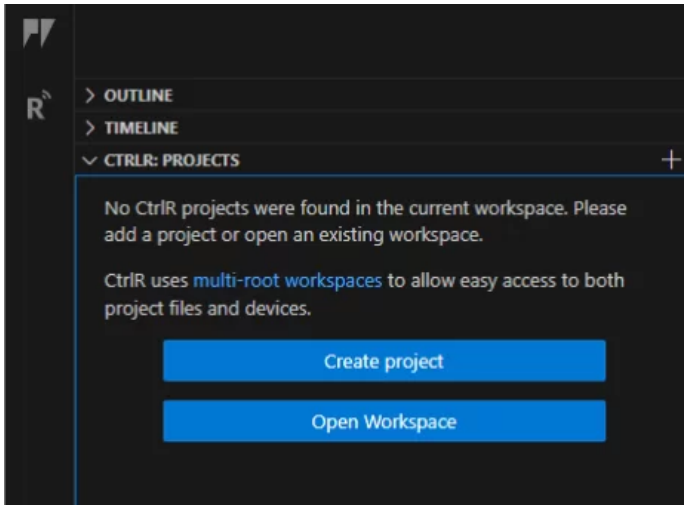
5.1.1 Creating a data upload project in CtrlR

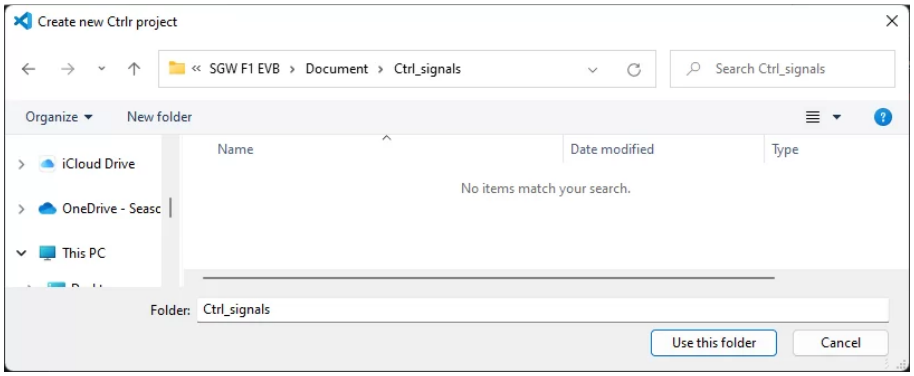
1. In a known destination on your computer, create a new project folder called **Ctrl_signals**.



2. Launch VS Code and open the **Ctrl_signals** project folder you created.

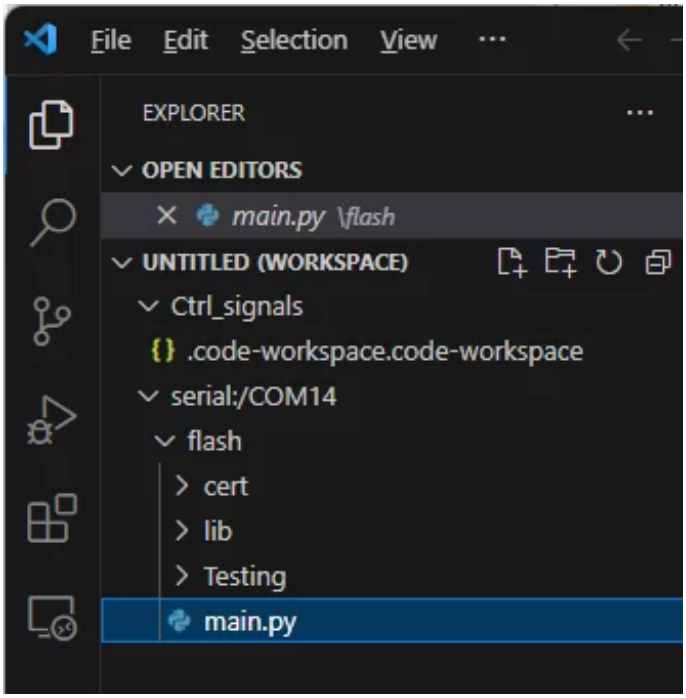






3. Modify the built-in python file **main.py** by adding the code below.

   The `main.py` script runs directly after `boot.py` and should contain the main code to run on the F1 Starter Kit. The modification will program the F1 Starter Kit for data upload.



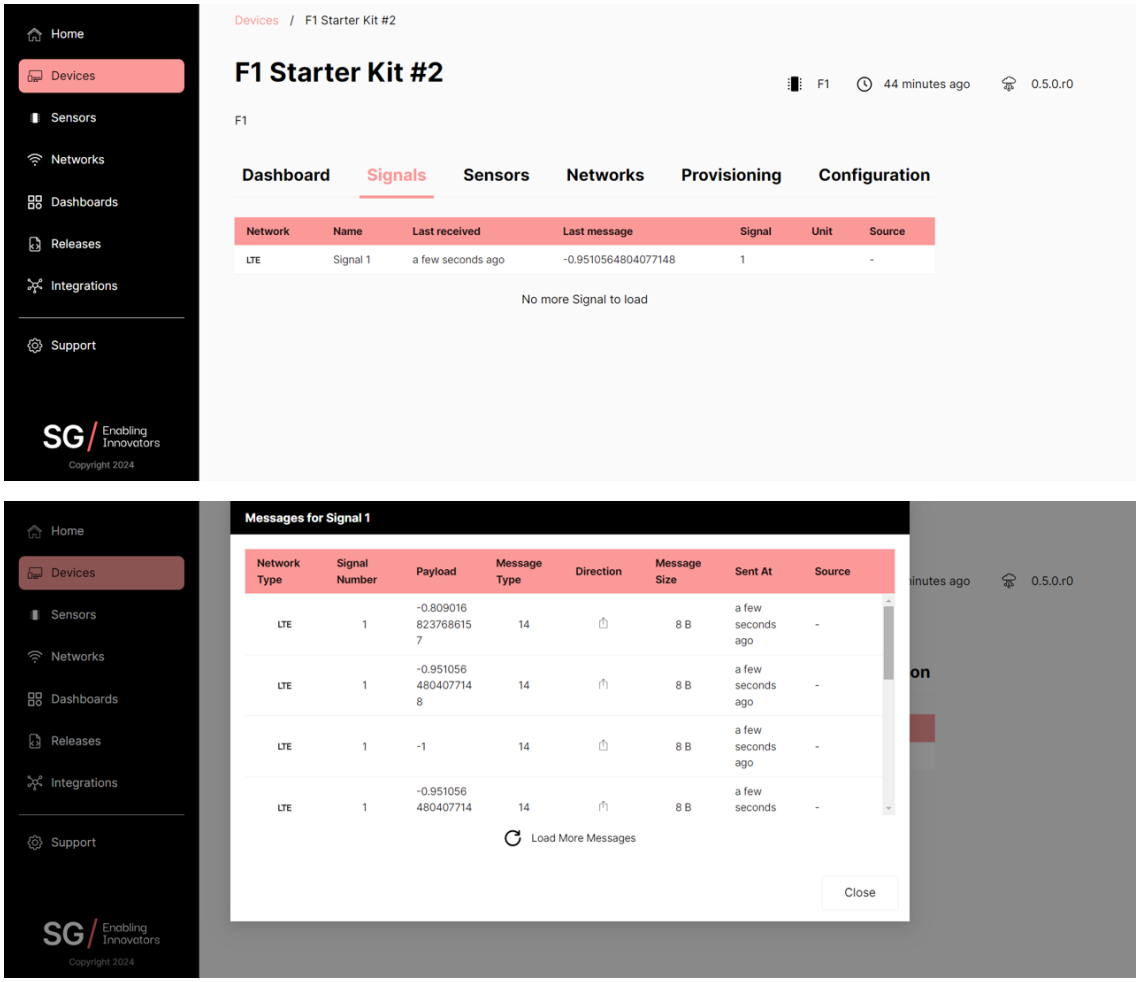

```
# Import what is necessary to create a thread
import time
import math

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

4. To test the code, use **File > Save** or `Ctrl+S` on your keyboard to save your edit to `main.py`, and press the Reset button on the F1 Starter Kit to reboot it.

   `main.py` will persist and run on the F1 Starter Kit when it is booted up. To stop the script, click onto the CtrlR terminal and press `Ctrl+C` on your keyboard.

5.1.2 Visualizing data on Ctrl

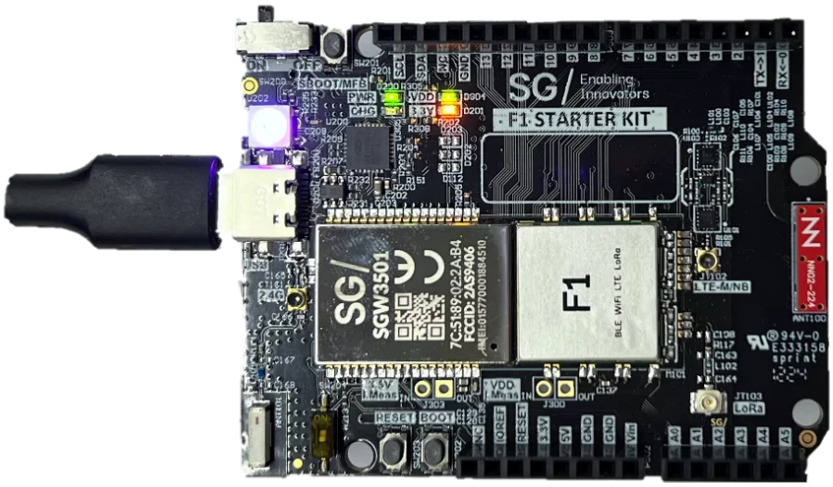
1. Back in your Ctrl account, navigate to **Devices** and select your connected F1 Starter Kit.
2. Navigate to **Signals** to see the last signal received by the F1 Starter Kit. Click on the signal to display historical messages.



5.2 Programming the Basics

Now that the F1 Starter Kit is connected to the Ctrl Cloud Platform, this tutorial covers how to program basic I/O on the F1 Starter Kit.

5.2.1 Light up the Onboard RGB LED



The F1 Starter Kit has an onboard, fully addressable RGB LED. You can set different colours and brightness levels.

1. Open the CtrlR REPL terminal and type the following commands to experiment with the RGB LED:

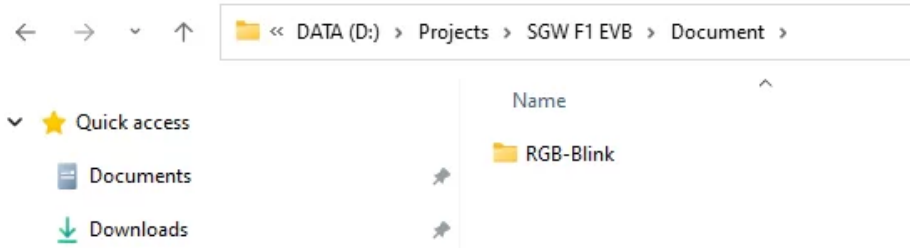
```
# Test RGB LED heartbeat
rgbled.heartbeat(True)
```

The RGB LED should now be pulsing.

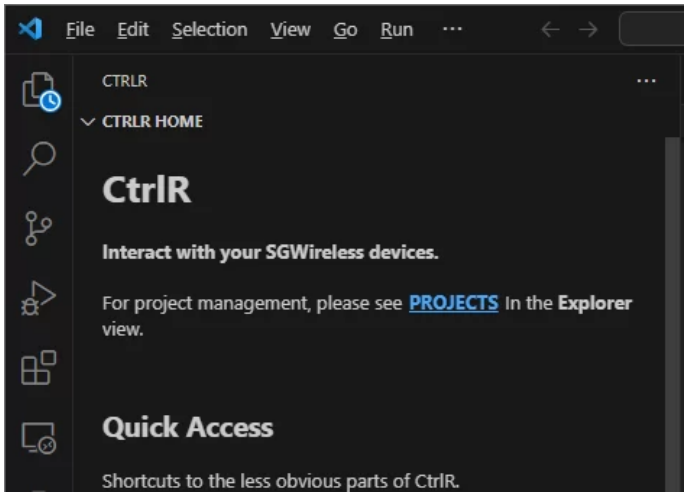
```
# Stop heartbeat and set a solid colour (R, G, B)
rgbled.heartbeat(False)
rgbled.color(0xff0000)    # Red
rgbled.color(0x00ff00)    # Green
rgbled.color(0x0000ff)    # Blue
```

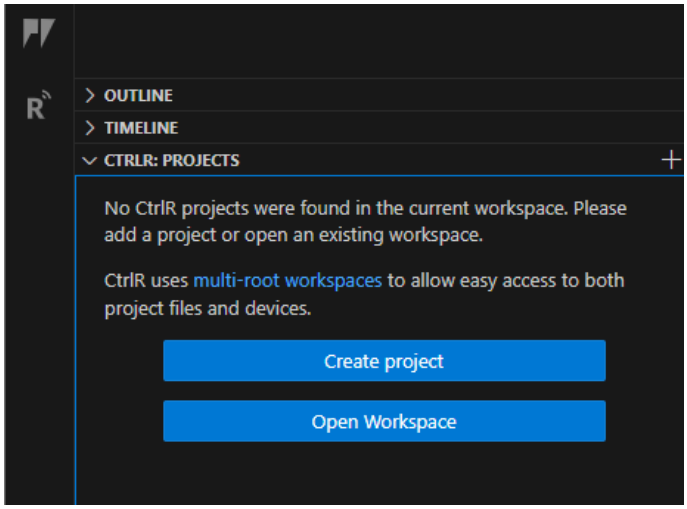
5.2.2 Creating a project in CtrlR

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

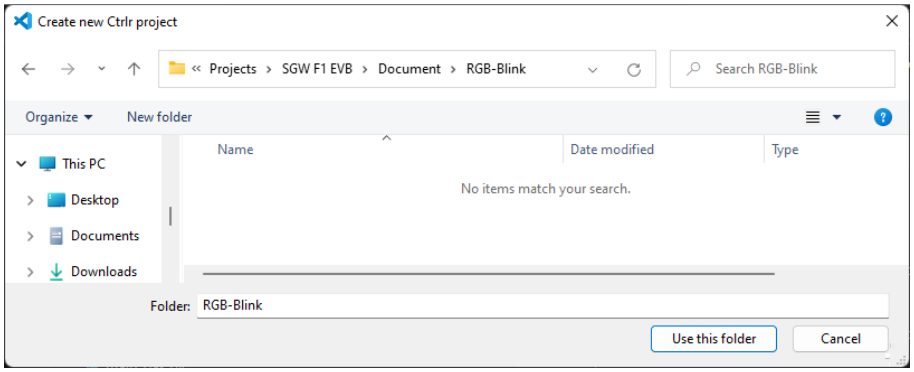


2. Launch VS Code and open the **RGB-Blink** project folder you created.





3. Create a new file called `main.py` and add the following code:

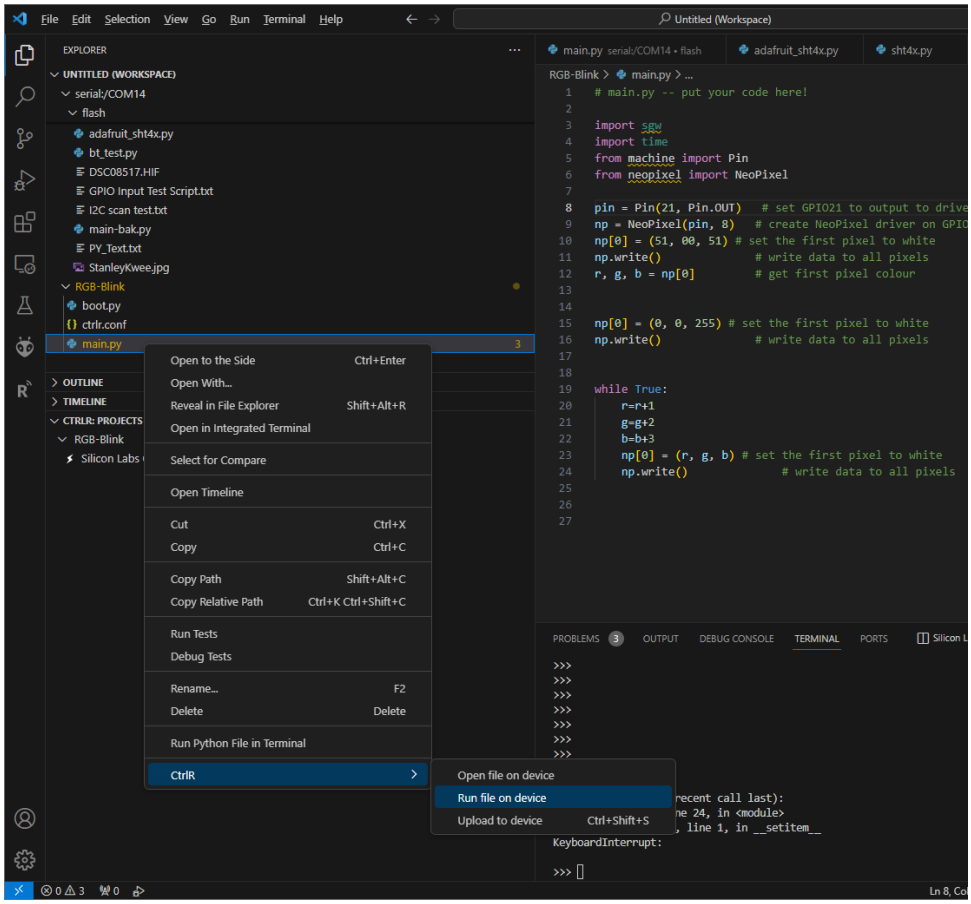



```
import time

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

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

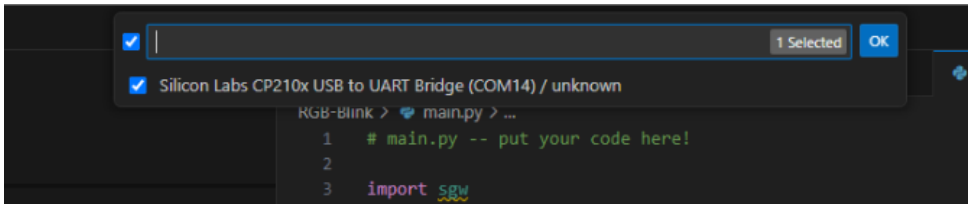
5.2.3 Uploading code to your F1 Starter Kit

You can test code on the F1 Starter Kit by choosing one of two options in CtrlR:

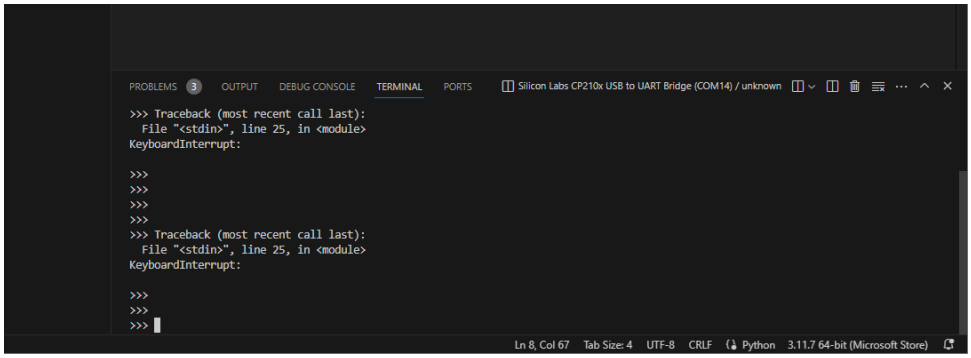
Run file on device



This option 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.



Upload to device



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.

5.3 Setting up Custom Networks

If you are using a third-party SIM card or need to connect to a specific LTE network, this guide covers how to configure the F1 Starter Kit's cellular connectivity.

5.3.1 Ctrl connection status

The F1 Starter Kit indicates its Ctrl connection status through the onboard RGB LED:

- **Pulsing Blue** — Connected to Ctrl
- **Pulsing Red** — Not connected to Ctrl (network issue or no SIM)
- **Pulsing Green** — Firmware update in progress

5.3.2 Configure for CAT-M1 or NB-IoT

The F1 Starter Kit supports both CAT-M1 and NB-IoT modes. By default it uses CAT-M1. To switch modes, use the REPL terminal:

```
import LTE

# Check current mode
LTE.mode()

# Set to CAT-M1
LTE.mode(LTE.CATM1)

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

After changing the mode, reboot the device for the setting to take effect.

5.3.3 Customize APN

If your SIM card requires a specific APN (Access Point Name), you can configure it using the following code:

```
import LTE

# Attach with a custom APN
lte = LTE.lte()
lte.attach(apn='your.apn.here')
```

Replace `'your.apn.here'` with the APN provided by your network operator.

> **Note**
>
> If you are using the SG Wireless SIM card that comes with the F1 Starter Kit, the APN is pre-configured and no changes are needed.

CHAPTER

SIX

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.

6.1 Basic Tutorials

6.1.1 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`.

6.1.2 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!

6.1.3 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)
```

6.1.4 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:

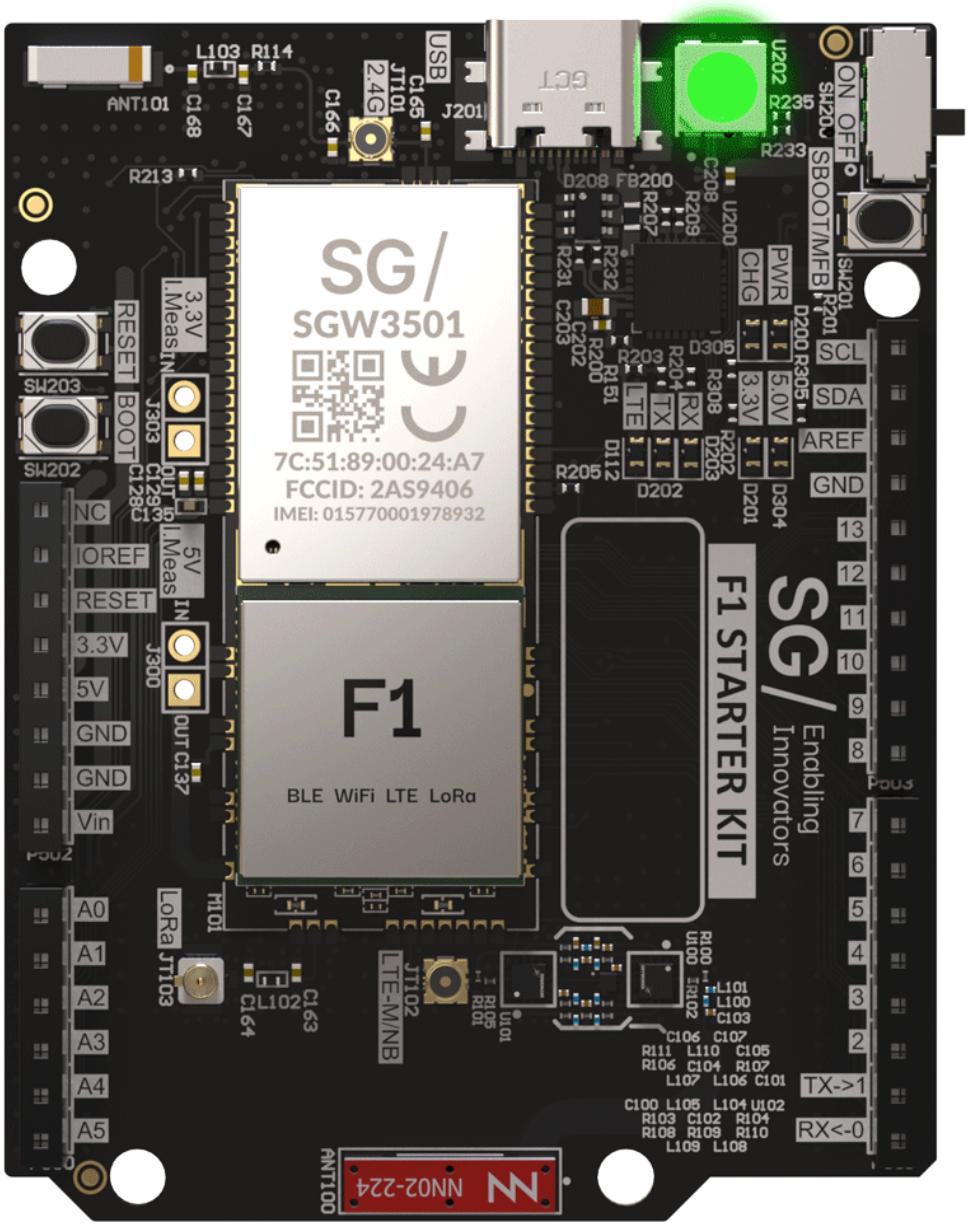
`rgbled.decoration(<blink-desc-list>, <repeat>)`

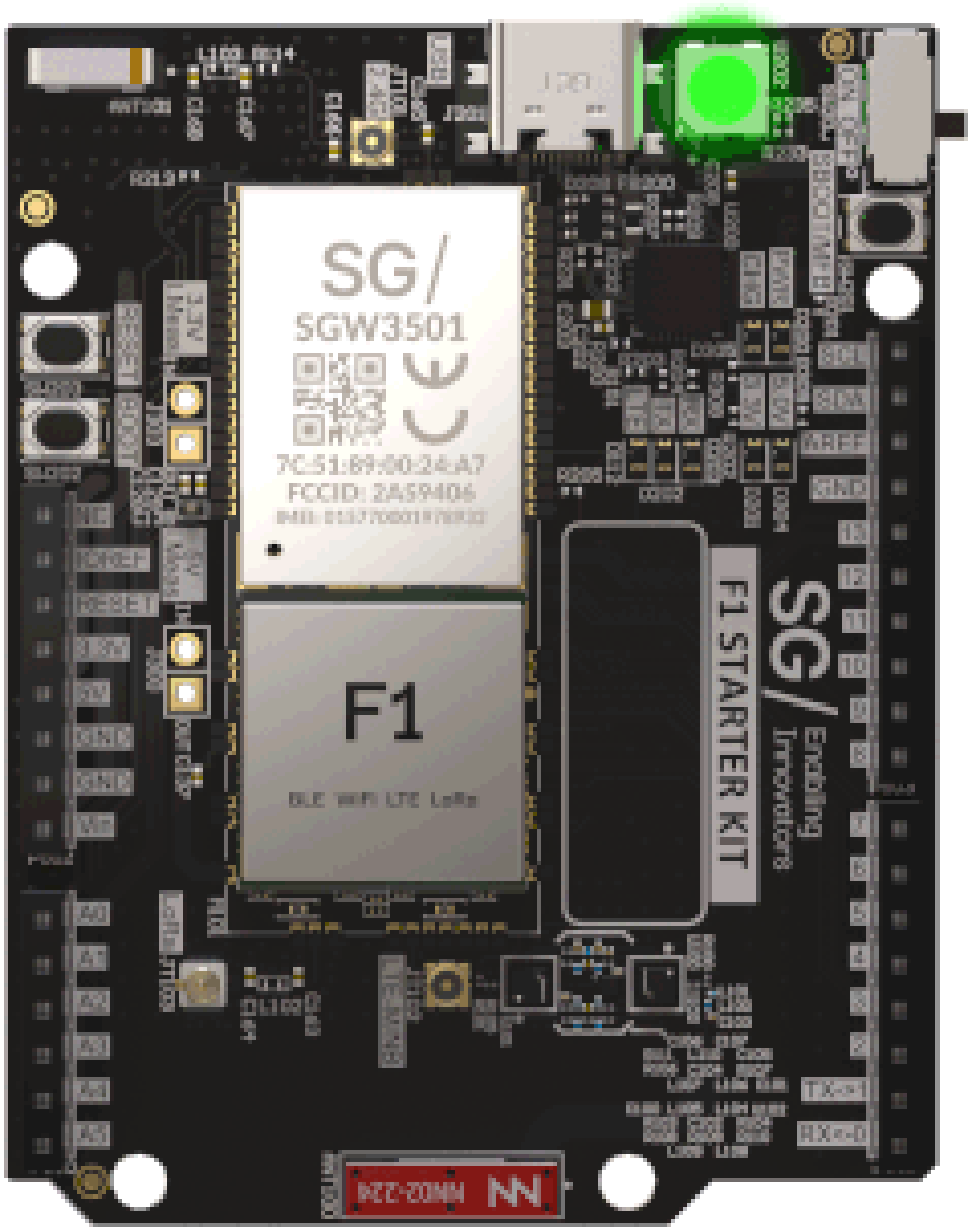
Where:

- `<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)
```

6.1.5 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
```

6.2 Hardware Tutorials

These tutorials cover interfacing with hardware peripherals on the F1 module.

6.2.1 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
```

6.2.2 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
```

6.2.3 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)
```

6.3 Network Tutorials

These tutorials cover wireless network connectivity on the F1 module.

6.3.1 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:

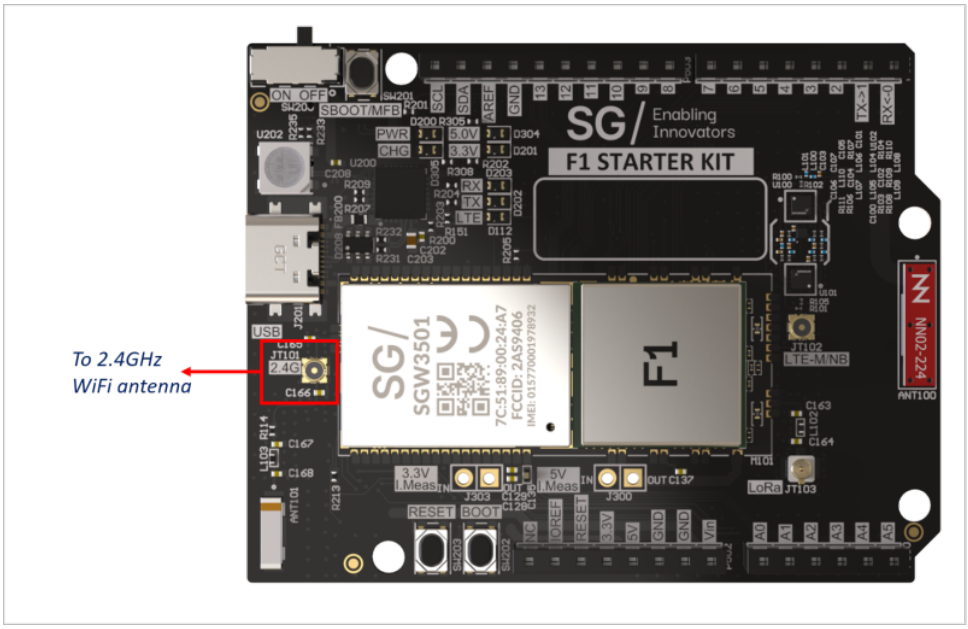
```
import network

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.

6.3.2 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.

6.3.3 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.

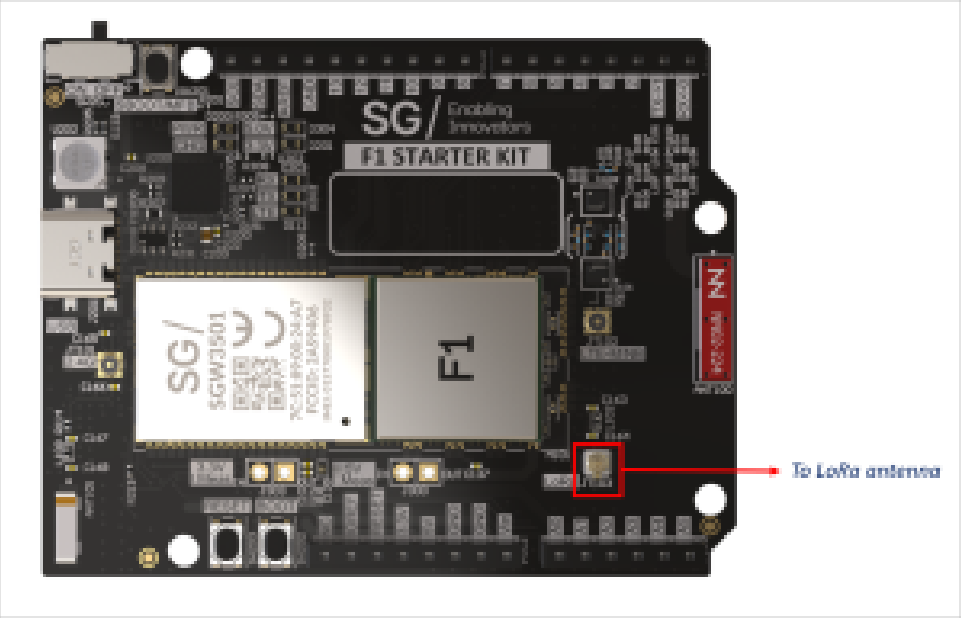


Fig. 1: F1 Starter Kit

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 ------------------------------------------------------------
↪-- #
```

6.3.4 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*

General remarks

To check the current modem firmware, you can use the following commands:

```
from LTE import LTE
lte = LTE()
print(lte.send_at_cmd('ATI1'))
```

To check whether the LTE connection has been established, you can use the following command:

```
from LTE import LTE
lte = LTE()
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:

```
from LTE import LTE
lte = LTE()
print(lte.isconnected())  # if True is returned, LTE connection is␣
↪established
```

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:

```
from LTE import LTE
lte = LTE()
lte_mode = lte.mode()  # Return current mode
if lte_mode == LTE.CATM1:
    print('Modem is in CAT-M1 mode!')
if lte_mode == LTE.NBIOT:
    print('Modem is in NB-IoT mode!')

# The below will automatically reset the modem
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:

```
from LTE import LTE
import time
lte = LTE()
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
while not lte.isattached():
    time.sleep(1)
lte.connect()
while not lte.isconnected():
    time.sleep(1)
```

CHAPTER

SEVEN

FIRMWARE & API REFERENCE

This chapter describes modules (function and class libraries) that are built into MicroPython. For standard modules implemented in MicroPython, refer to the MicroPython documentation.

7.1 SG Modules

These modules are specific to the SG devices and may have slightly different implementations to other variations of MicroPython.

Note

This documentation in general aspires to describe all modules and functions/classes which are implemented in MicroPython. However, MicroPython is continuously evolving and some functions, classes or modules may not be available on the SG Wireless F1 module yet.

7.1.1 Ctrl Client API Documentation

Contents

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")

    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.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.

Signals

[Ctrl firmware version 1.3 or above] ctrl.send_field(pin_number, value,timestamp)

pin_number: The destination field pin number configured through Ctrl. page.

value: The value you want to send, it should be of the same data type as the field.

timestamp: This is optional, the received timestamp will be used by default

[Ctrl firmware version 1.0 or below] ctrl.send_signal(signal_number, value)

Send a signal to CTRL. Arguments are:

signal_number: The signal number in CTRL, can be any value between 0-254 (255 is reserved)

value: The value you want to send, this can be any type.

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())
```

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_signal(1, math.sin(i/10*math.pi))
        print('sent signal {}'.format(i))
        time.sleep(10)
```

Example 2:

This example sends data from multiple pre-provisioned SGW8130 BLE sensors to CTRL The transmission interval is 5 seconds when using WiFi else 300 seconds to preserve bandwidth Sensor values are queued and delivered evenly to avoid rate limiting restrictions

```
import bluetooth
import struct
import time
import sys
import micropython
from ubinascii import hexlify, unhexlify

_IRQ_SCAN_RESULT = micropython.const(5)
_IRQ_SCAN_DONE = micropython.const(6)

micropython.alloc_emergency_exception_buf(100)

MIN_DELAY = 5 if ctrl.get_network_type() == 0 else 300

SGW_MAC_PREFIX = '7C5189'

REGISTERD_DEVICES = {
        SGW_MAC_PREFIX + '004F53': 'd5dd1706-acd1-49ce-875e-e362ad96f430',
        SGW_MAC_PREFIX + '004577': '7948c093-dbba-496e-9367-082a0d244d86',
        SGW_MAC_PREFIX + '0056D5': '1bac313c-b661-4c5e-bbde-0506c564a251',
        SGW_MAC_PREFIX + '0058A1': '40dc1654-ae54-4fe7-9461-b6813c716a4c'
}
```

(continues on next page)

(continued from previous page)

```
PRINT_ON = True
CTRL_ON = True
DEBUG=9

SCAN_INTERVAL=5000
SCAN_WINDOW=5000

class ctrl_sender():
    def __init__(self):
        self.ble = bluetooth.BLE()
        self.ble.active(True)
        self.ble.irq(self.bt_irq)
        self.ble.gap_scan(0, SCAN_INTERVAL, SCAN_WINDOW)
        self.last_sent = {'last_sent' : time.time() - MIN_DELAY}
        for key in REGISTERD_DEVICES.values():
            self.last_sent.update({key: 0})
        self.advertising_data = bytes(96)
        self.rssi = bytes(1)
        self.add_type = bytes(1)
        self.send_data_ref = self.send_data

    def print_debug(self, lvl, msg):
        if lvl <= DEBUG:
            print(msg)

    def send_data(self, device_token):
        adv_data = self.advertising_data
        addr_type = self.addr_type
        rssi = self.rssi
        self.print_debug(5, self.last_sent)
        self.print_debug(2, 'adv_data: {}'.format(adv_data))
        if (int.from_bytes(adv_data[9:11],'little')==89):
            try:
                if not (adv_data[11:12]==b'v'):
                    self.process_signal(0, "RSSI", rssi, device_token)
                    self.process_signal(1, "Temperature", int.from_
↪bytes(adv_data[11:13],'little')/100, device_token)
                    self.process_signal(2, "Humidity", int.from_bytes(adv_
↪data[13:15],'little')/100, device_token)
                    self.process_signal(3, "Battery", int.from_bytes(adv_
↪data[15:16],'little'), device_token)
                    self.last_sent[device_token] = time.time()
                    self.last_sent['last_sent'] = time.time()
            except Exception as ex:
                sys.print_exception(ex)
        return None

    def bt_irq(self, event, data):
```

(continues on next page)

(continued from previous page)

```
        if event == _IRQ_SCAN_RESULT:
            try:
                addr_type, addr, adv_type, rssi, adv_data = data
                hr_addr = hexlify(addr).decode('UTF-8').upper()
                if SGW_MAC_PREFIX in hr_addr:
                    self.print_debug(3, hr_addr)
                    device_token = REGISTERD_DEVICES.get(hr_addr)
                    self.print_debug(1, "mac={} device_token={}".format(hr_
↪addr, device_token))
                    if (adv_type==0) and (device_token is not None):
                        last_sent_global = self.last_sent.get('last_sent')
                        last_sent_device = self.last_sent.get(device_token)
                        self.print_debug(5, 'time.time()={} - last_sent_
↪global={} > MIN_DELAY={} = {}'.format(time.time(), last_sent_global, MIN_
↪DELAY, time.time() - last_sent_global > MIN_DELAY))
                        self.print_debug(5, 'time.time()={} - last_sent_
↪device={} > MIN_DELAY={} * len(REGISTERD_DEVICES)={} ={}'.format(time.
↪time(), last_sent_device, MIN_DELAY, len(REGISTERD_DEVICES), time.time() -
↪ last_sent_device > MIN_DELAY * len(REGISTERD_DEVICES)))
                        if (time.time() - last_sent_global > MIN_DELAY) and␣
↪(time.time() - last_sent_device > MIN_DELAY * len(REGISTERD_DEVICES)):
                            self.advertising_data = bytes(adv_data)
                            self.addr_type = addr_type
                            self.rssi = rssi
                            micropython.schedule(self.send_data_ref, device_
↪token)
            except Exception as ex:
                sys.print_exception(ex)
        elif event == _IRQ_SCAN_DONE:
            self.ble.gap_scan(0, SCAN_INTERVAL, SCAN_WINDOW)
        else:
            self.print_debug(1, "BLE event of type: {}".format(event))

    def process_signal(self, sig_nr, name, msg, device_token):
        if PRINT_ON:
            print('{}[{}]: {}'.format(name, sig_nr, msg))
        if CTRL_ON and device_token is not None:
            ctrl.send_signal(sig_nr, msg, device_token=device_token)

sender = ctrl_sender()
```

7.1.2 LoRa API Documentation

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␣
```

(continues on next page)

(continued from previous page)

```
↪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()                  # displays 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
```

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
```

LoRa Events and Callbacks

LoRa events and callback system are explained in *LoRa Callback System*.

LoRa RAW APIs

LoRa RAW mode API documentation can be found in *LoRa RAW API Documentation*.

LoRa WAN APIs

LoRa WAN mode API documentation can be found in *LoRa WAN API Documentation*.

7.1.3 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 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

To give the device its credentials, it takes the following arguments:

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

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

REMARK 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

# 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'),
```

(continues on next page)

(continued from previous page)

```
    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

Mandator operation to let the device join the network and be able to TX/RX with the lora-WAN server

Example:

```
import lora
import time

# start join procedure
lora.join()

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

Sending data

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
config	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
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 over it

Example:

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

lora.send('any', port=5)     # ignored because port 5 is not opened
lora.send('any', 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('any', 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 (default any)

'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 get_evt_str(evt):
    if evt != None:
        if evt == lora._event.EVENT_TX_DONE:
            return 'EVENT_TX_DONE'
        elif evt == lora._event.EVENT_TX_TIMEOUT:
            return 'EVENT_TX_TIMEOUT'
        elif evt == lora._event.EVENT_TX_FAILED:
            return 'EVENT_TX_FAILED'
        elif evt == lora._event.EVENT_TX_CONFIRM:
            return 'EVENT_TX_CONFIRM'
        elif evt == lora._event.EVENT_RX_DONE:
            return 'EVENT_RX_DONE'
        elif evt == lora._event.EVENT_RX_TIMEOUT:
            return 'EVENT_RX_TIMEOUT'
        elif evt == lora._event.EVENT_RX_FAIL:
            return 'EVENT_RX_FAIL'
    return '--UNKNOWN-EVENT--'

def cb_on_any(event, evt_data):
    print('(  cb_on_any   )--> event: ' + get_evt_str(event))
    if evt_data != None:
        print(evt_data)
    pass

def cb_on_port_1(event, evt_data):
    print('( cb_on_port_1 )--> event: ' + get_evt_str(event))
    if evt_data != None:
        print(evt_data)
    pass

port_1_triggers = lora._event.EVENT_RX_DONE | lora._event.EVENT_TX_DONE
lora_unittest.test_callback_set(handler=cb_on_port_1, trigger=port_1_
↪triggers, port=1)
lora_unittest.test_callback_set(handler=cb_on_any)
lora_unittest.test_gen_all_callbacks()
```

Duty cycle operations

The device is normally working in class-A and the device shall follow a duty-cycle to do 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
```

7.1.4 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
        region          : EU-868
        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:

```
::=
       "lora.radio_params("    ")"

::=
       ","
       |
       | ""

::=
   reset_all   "="             ; reset to factory settings
   | region      "="           ; change the region
   | frequency   "="       ; desired freq in Hz
   | freq_khz    "="      ; desired freq in KHz
   | freq_mhz    "="      ; desired freq in MHz
   | tx_power    "="       ; desired tx power
   | sf          "="               ; spreading factor
   | coding_rate "="               ; coding rate
   | preamble    "="          ; preamble length
   | bandwidth   "="               ; band-width
   | tx_iq       "="             ; inverted IQ feature

::= "True" | "False"

::= "lora._region."
::=
   "REGION_AS923" | "REGION_AU915" | "REGION_EU868" | "REGION_IN865" |
↪"REGION_KR920"
   | "REGION_RU864" | "REGION_US915"

::=
::=
::=

::= "7" | "8" | "9" | "10" | "11" | "12"

::= "lora._bw."
```

(continues on next page)

(continued from previous page)

```
::= "BW_125KHZ" | "BW_250KHZ" | "BW_500KHZ"

::= "lora._cr."
::= "CODING_4_5" | "CODING_4_6" | "CODING_4_7" | "CODING_4_8"
```

Examples:

```
# 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

# 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

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

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

>>> lora._region.    # press   to see the following list
REGION_AS923    REGION_AU915    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, 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 get_event_str(event, bytes):
    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_TX_CONFIRM:
        return 'EVENT_TX_CONFIRM'
    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'
    pass

def lora_callback(event, evt_data):
    print('lora event [ {} ] --> data: {}'
        .format(get_event_str(event), evt_data))

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

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 = 868000000,          # test frequency 868 MHz
    timeout = 20000)                # timeout for the tx continuous wave 20␣
↪sec

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

7.1.5 LoRa Callback System

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_TIMEOU	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_TIMEOU	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 continuout receiving mode and new data received and present to be delivered to the user.

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.

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 =
    [, trigger = ]
    [, port = ]
)
```

The parameters description is as follows:

handler This is the real micropython function to be called when a LoRa event occurs. The full handler description is in the following typical handler:

```
def lora_generic_callback(event, evt_data):
  # --- lora raw case
  if lora.mode() == lora._mode.RAW:

      if event == lora._event.EVENT_RX_DONE:
          # evt_data is a byte array object containing the received data
          print('received data: {}'.format(evt_data))
          pass
      elif event == lora._event.EVENT_RX_FAIL:     # evt_data is None
          pass
      elif event == lora._event.EVENT_RX_TIMEOUT:  # evt_data is None
          pass
      elif event == lora._event.EVENT_TX_DONE:     # evt_data is None
          pass
      elif event == lora._event.EVENT_TX_CONFIRM:
          # unexpected event in lora-raw
          pass
      elif event == lora._event.EVENT_TX_FAILED:   # evt_data is None
          pass
      elif event == lora._event.EVENT_TX_TIMEOUT:  # evt_data is None
          pass
      else:
          print('error: unknown error')

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

      if event == lora._event.EVENT_RX_DONE:
          # evt_data is a byte array object containing the received data
          print('received data: {}'.format(evt_data))
          pass
```

(continues on next page)

(continued from previous page)

```
      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:
          # evt_data is an integer holding the message id provided in tx req
          print("message ID {} has transmitted successfully".format(evt_
↪data))
          pass
      elif event == lora._event.EVENT_TX_CONFIRM:
          # evt_data is an integer holding the message id provided in tx req
          print("message ID {} has been confirmed".format(evt_data))
          pass
      elif event == lora._event.EVENT_TX_FAILED:
          # evt_data is an integer holding the message id provided in tx req
          print("message ID {} is not transmitted".format(evt_data))
          pass
      elif event == lora._event.EVENT_TX_TIMEOUT:
          # evt_data is an integer holding the message id provided in tx req
          print("message ID {} tx timeout".format(evt_data))
          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():
```

(continues on next page)

(continued from previous page)

```
        if v == __const:
            return k
    return 'unknown'

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

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

def lora_raw_callback_timeout(event, event_data):
    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
    )

# 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 --
```

(continues on next page)

(continued from previous page)

```
#   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(event, event_data):
    print('lora-wan event: {} with data: {}'.format(
        get_class_const_name(lora._event, event), event_data))
    pass

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

def lora_wan_callback_port_1_tx_confirm(event, event_data):
    if event != lora._event.EVENT_TX_CONFIRM:
        print("unexpected event in this callback")
        return
    print('lora-wan port 1 tx confirm on message id : {}'.format(event_
↪data))
    pass

def lora_wan_callback_port_2_timeout(event, event_data):
    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)
```

(continues on next page)

(continued from previous page)

```
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.

7.1.6 LTE

Constructors

class LTE.LTE(...)

Create and configure a LTE object. See __init__ for params of configuration.

```
from LTE import LTE
lte = LTE()
```

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)

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.

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.pppsuspend() and lte.ppresume 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

Constants

LTE.CATM1 : For use in CATM1 mode

LTE.NBIOT : For use in NBIOT mode

7.1.7 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 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.heartbeat() example:

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() to start the heartbeat blinking service. it has three signature as follows:

rgbled.heartbeat()

rgbled.heartbeat(<enable>)

rgbled.heartbeat(<color>, <cycle-time>, <blink-percentage>)

The description of each signature is as follows:

Check the current status of the heartbeat service and returns True or False.

To enable/disable the service

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
```

(continues on next page)

(continued from previous page)

```
       # 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()

It provides 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>, ... ]

<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

```
# 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
```

Here is the expected result:

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 )
rgbled.color( rgbled._color.CYAN )
rgbled.color( rgbled._color.WHITE )
```

CHAPTER

EIGHT

CONTACT

Email: info@sgwireless.com
Website: https://sgwireless.com/
LinkedIn: https://www.linkedin.com/company/sgwireless/

CHAPTER

NINE

LICENSE