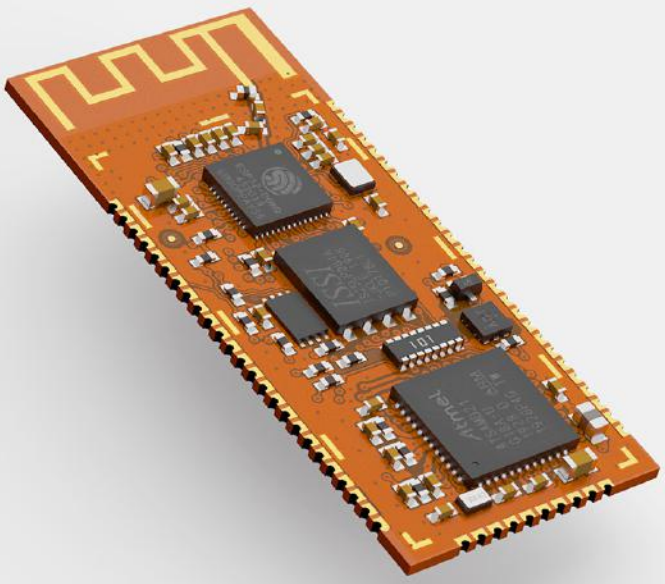
Briki MBC-WB User Manual

Wi-Fi & Bluetooth Device



Revision History

Revision number	Revision date	Summary of changes	Authors
1.2	07/04/2021	Pinout correction New pinout pictures	D.Trimarchi
1.1	20/01/2021	Key features, Security, Module geometry and PCB land pattern	D.Trimarchi
1.0	11/11/2020	Initial version	D.Trimarchi C. Ruggeri

Content Index

Figures Index

Tables Index

Tips & Tricks

Notes

Warning messages

```
Code snippets
```

Getting Started

The MBC-WB SoM is part of Briki MBC family of devices that share the same pinout and form factor regardless of each module's characteristics, such as MCU selection, peripherals, or field of application. This compact and certified SoM is the ideal solution for designers who want a unique device for their IoT or IIoT projects with Wi-Fi & BT/BLE connectivity plus a dedicated control MCU.

It features (depending on module versions):

— an ATSAMD21G18A ARM® Cortex®-M0+ MCU with 256 KB Flash, 32 KB of SRAM, operating at a maximum frequency of 48 MHz and reaching 2.46 CoreMark/MHz (MBC-WB01);

— an ESP32-D0WD dual-core Tensilica Xtensa LX6 with 448 KB ROM, 520 KB SRAM, plus a QSPI interface for an external flash and running at a maximum frequency of 240 MHz with integrated Wi-Fi and dual-mode Bluetooth;

— a CryptoAuth ECC608A chip for the module security;

— a QSPI 64-Mbit or 128-Mbit flash (partially used by ESP32 and part as a user free storage space).

The choice to put two of the most used microcontrollers on the market onto the same board has been driven by the need to overcome their respective limitations, offering to the customer the most versatile and complete SoM.

Both the MCUs are particularly good devices with different and complementary features. For example: flexible communication interfaces, USB with embedded host and device function, good analog and timing performances for the SAMD21, fast dual-core architecture, wireless communication, great computational power for the ESP32. The MBC-WB takes all their features and makes them available in a tiny 38x16mm SoM with 62 pins.

A complete module pinout, with each pin functions description, is available here, while the block diagram of the module is available here.

Two separate power supplies are exposed on the module pinout to allow the maximum flexibility and control over its power supply requirements. Detailed information about power supply voltages and current consumptions can be found in the Electrical characteristics section of this manual.

Detailed information about the supported power and sleep modes are available in the Power modes section.

The USB interface is available on the module pinout. By default, it is configured in device mode with two separate USB-CDC interfaces, one for the SAMD21 and one for the ESP32. For more information, read the related USB Interface section.

Both MCUs are fully user accessible (both in programming and debugging) to allow a full configuration/customization for each design's needs. The SAMD21 can be reprogrammed in-system through the SWD interface exposed on the module pinout. For non-intrusive on-chip debug of application code, the same interface can be used. The ESP32 can be reprogrammed through the classic UART interface or through the JTAG interface, which is also useful for on-chip debugging and is exposed on the module pinout.

Each chip can update its own or the counterpart's firmware (even in a secured environment, thanks to the crypto chip supplied with the board). This leads to two more ways to reprogram the MBC-WB:

— through the USB bootloader in the SAMD21. For more information about how the SAMD21's bootloader works, click here

— through the Wi-Fi based OTA interface offered by the ESP32. For more information about how the ESP32's OTA works, click here

Since the firmware of both MCUs can be customized, the user has the power to decide which of the two should act as main or companion and what task each chip should perform. This allows the designer to achieve a perfect balance between power consumption, functionalities implemented and overall performance. Even a "liquid" configuration can be achieved, where each MCU works based on its own strengths, exchanging data and messages thanks to the shared communication interfaces.

Two communication interfaces, plus some synchronizing signals, are shared between the two MCUs: one SPI and one UART, suitable for data exchange and firmware update. All the other peripherals and GPIOs are available for the user and exposed on the module pinout. To better understand how the two communication interfaces works, please check the Communication section of this manual.

Every MBC features a special ID line that can be configured to output to a carrier board the SoM's characteristics (MCUs mounted on the SoM, operating voltages, power consumptions, features enabled, etc.) during the board boot.

Key features

Features of the ESP32:

- Processors
  - CPU: Xtensa dual-core (or single-core) 32-bit LX6 microprocessor, operating at 240 MHz and performing at up to 600 DMIPS
  - Ultra low power (ULP) co-processor
  - Memory: 520 KiB SRAM

- Wireless connectivity
  - Wi-Fi: 802.11 b/g/n
  - Bluetooth: v4.2 BR/EDR and BLE

- Peripherals
  - 12-bit SAR ADC up to 18 channels
  - 2 × 8-bit DACs
  - 10 × touch sensors (capacitive sensing GPIOs)
  - Temperature sensor
  - 4 × SPI
  - 2 × I²S interfaces
  - 2 × I²C interfaces
  - 3 × UART
  - SD/SDIO/CE-ATA/MMC/eMMC host controller
  - SDIO/SPI slave controller
  - Ethernet MAC interface with dedicated DMA and IEEE 1588 Precision Time Protocol support
  - CAN bus 2.0
  - Infrared remote controller (TX/RX, up to 8 channels)
  - Motor PWM
  - LED PWM (up to 16 channels)
  - Hall effect sensor
  - Ultra low power analog pre-amplifier

- Security
  - IEEE 802.11 standard security features all supported, including WFA, WPA/WPA2 and WAPI
  - Secure boot
  - Flash encryption
  - 1024-bit OTP, up to 768-bit for customers
  - Cryptographic hardware acceleration: AES, SHA-2, RSA, elliptic curve cryptography (ECC), random number generator (RNG)

- Power management
  - Internal low-dropout regulator
  - Individual power domain for RTC
  - 5uA deep sleep current
  - Wake up from GPIO interrupt, timer, ADC measurements, capacitive touch sensor interrupt

Features of the ATSAMD21:

- Processor
  - ARM©, Cortex-M0+ CPU running at up to 48MHz
  - Single-cycle hardware multiplier
  - Micro Trace Buffer

- Memories
  - 256KB in-system self-programmable Flash
  - 32KB SRAM Memory

- System
  - Power-on reset (POR) and brown-out detection (BOD)
  - Internal and external clock options with 48MHz Digital Frequency Locked Loop (DFLL48M) and 48MHz to 96MHz fractional
  - External Interrupt Controller (EIC), 16 external interrupts, one non-maskable interrupt
  - Two-pin Serial Wire Debug (SWD) programming, test and debugging interface
  - Low Power
  - Idle and standby sleep modes
  - SleepWalking peripherals

- Peripherals
  - 12-channel Direct Memory Access Controller (DMAC)
  - 12-channel Event System
  - Up to five 16-bit Timer/Counters (TC), configurable as either:
  - One 16-bit TC with compare/capture channels
  - One 8-bit TC with compare/capture channels
  - One 32-bit TC with compare/capture channels, by using two TCs
  - Three 24-bit Timer/Counters for Control (TCC), with extended functions:
    - Up to four compare channels with optional complementary output
    - Generation of synchronized pulse width modulation (PWM) pattern across port pins,
    - Deterministic fault protection, fast decay and configurable dead-time between complementary output
    - Dithering that increase resolution with up to 5 bit and reduce quantization error
  - 32-bit Real Time Counter (RTC) with clock/calendar function
  - Watchdog Timer (WDT)
  - CRC-32 generator
  - One full-speed (12Mbps) Universal Serial Bus (USB) 2.0 interface with Device and embedded Host and 8 endpoints
  - Six Serial Communication Interfaces (SERCOM), each configurable to operate as either:
    - USART with full-duplex and single-wire half-duplex configuration
    - I2C Bus up to 3.4MHz
    - SMBUS/PMBUS
    - SPI
    - LIN slave
  - 12-bit, 350 ksps Analog-to-Digital Converter (ADC) with up to 14 channels
    - Differential and single-ended input
    - 1/2x to 16x programmable gain stage
    - Automatic offset and gain error compensation
    - Oversampling and decimation in hardware to support 13-, 14-, 15- or 16-bit resolution
  - 10-bit, 350ksps Digital-to-Analog Converter (DAC)
  - Two Analog Comparators (AC) with window compare function
  - Peripheral Touch Controller (PTC) with 256-channel capacitive touch and proximity sensing I/O
  - 38 GPIO pins

Features of the ATECC608A:

- Cloud authentication for AWS IoT
- Cloud authentication for Google Cloud IoT Core
- Secure Boot implementation with an ATSAMD21 Cortex-M0+
- Cryptographic coprocessor with secure hardware-based key storage
  - Protected storage for up to 16 Keys, certificates or data
- Hardware support for asymmetric sign, verify, key agreement – ECDSA: FIPS186-3 Elliptic Curve Digital Signature
- ECDH: FIPS SP800-56A Elliptic Curve Diffie-Hellman
- NIST standard P256 elliptic curve support
- Hardware support for symmetric algorithms
  - SHA-256 & HMAC hash including off-chip context save/restore
  - AES-128: encrypt/decrypt, galois field multiply for GCM
- Networking key management support
  - Turnkey PRF/HKDF calculation for TLS 1.2 & 1.3
  - Ephemeral key generation and key agreement in SRAM – Small message encryption with keys entirely protected
- Secure boot support
  - Full ECDSA code signature validation, optional stored digest/signature – optional communication key disablement prior to secure boot
  - Encryption/Authentication for messages to prevent on-board attacks
- Internal high-quality FIPS 800-90 A/B/C Random Number Generator (RNG)
- Two high-endurance monotonic counters
- Guaranteed unique 72-bit serial number
- Two interface options available
  - High-speed single pin interface with One GPIO pin
  - 1MHz Standard I2C interface
- 1.8V to 5.5V IO levels, 2.0V to 5.5V supply voltage
- <150nA Sleep current

Security

A key point of IoT and IIoT is security. Since many industrial companies are extensively implementing the Internet of Things at the edge of their networks and increasing the capabilities of the network itself, connecting many devices presents a huge security threat. This big number of connected devices offers a much larger surface prone to cyber-attack than the IT space where, by comparison, the volumes of data are lower and its exchanging can be more precisely controlled.

In the industrial sector, huge amounts of data are being processed by physical devices at the edge (through their firmware), sent back to the cloud for further analysis and used by different applications or, after processing, by the devices themselves to control the processes or the plant environment. Attackers can exploit these devices and their software to subvert and compromise the hardware itself. Significant numbers of IoT devices are not being used with security in mind, so every single device and sensor in the IoT represent a potential risk and an easy entry point to the network.

The MBC-WB01 features a double security layer.

The first directly inside the ESP32, that, along with TLS v1.2, implements secure boot, flash content encryption and cryptographic hardware acceleration (with AES, SHA-2, RSA, elliptic curve cryptography (ECC) and random number generator (RNG) support).

The secure boot support ensures that when the ESP32 executes any software from flash, that software is trusted and signed by a known entity. If even a single bit in the software bootloader and application firmware is modified, the firmware is not trusted, and the device will refuse to execute this untrusted code. This is achieved by building a chain of trust from the hardware, to the software bootloader, to the application firmware.

The flash encryption support ensures that any application firmware, that is stored in the flash of the ESP32, stays encrypted. This allows manufacturers to ship encrypted firmware in their devices.

The second is the Microchip ATECC608A that integrates ECDH (Elliptic Curve Diffie Hellman) security protocol - an ultra-secure method to provide key agreement for encryption/decryption - along with ECDSA (Elliptic Curve Digital Signature Algorithm) sign-verify authentication.

It also offers an integrated AES hardware accelerator extending the secure boot features to both the MBC's MCU and supplying the full range of security such as confidentiality, data integrity, and authentication to MBC's system. The ATECC608A employs ultra-secure hardware-based cryptographic key storage and cryptographic countermeasures which eliminate potential backdoors linked to software weaknesses.

It can also perform Elliptic Curve Diffie Hellman Key Exchange which means that the part can securely store the asymmetric keys (private key) for a TLS (v. 1.3) exchange and deliver the master secret to the microcontroller for the symmetric portions of the protocol, simplifying the connection and the authentication to Azure, AWS, and Google cloud platforms.

Ordering information

MBC

(Example: MBC-WB01-BPN)

MBC	-	WB	01	-	B	P	N
Device family: Modular Brick Concept		**Features:** W: Wi-Fi B: BT/BLE	**HW revision:** 01-09: D21 11-19: L21 21-29: C21 31-39: D51		**Flash Size:** A: 16 Mbit B: 64 Mbit C: 128 Mbit	**Ant. Option:** 0: I-PEX P: PCB Antenna	**Enhanced Security:** N: no S: yes

Current configurations:

MBC-WB01-APN
MBC Light version

MBC-WB01-BPN
MBC Standard version

MBC-WB01-CPS
MBC Full version

DBC

(Example: DBC-01-3U)

DBC	-	01	-	3	U
Device family: Developer Carrier Board		**HW revision:** 01-09: Debugger base 11-19: Integrated debugger		**Operating Voltage:** 3: 3.3V 5: 5V	**Features:** U: USB C: CAN

When the MBC is mounted on the DBC: **DBC-01-3U + MBC-WB01-BPN**

ABC

(Example: ABC-01)

ABC	-	01
Device family: Advanced Carrier Board		**HW revision:**

When the MBC is mounted on the ABC: **ABC-01 + MBC-WB01-BPN**

Block Diagram

The following image shows the functional block diagram of the MBC-WB01 module.

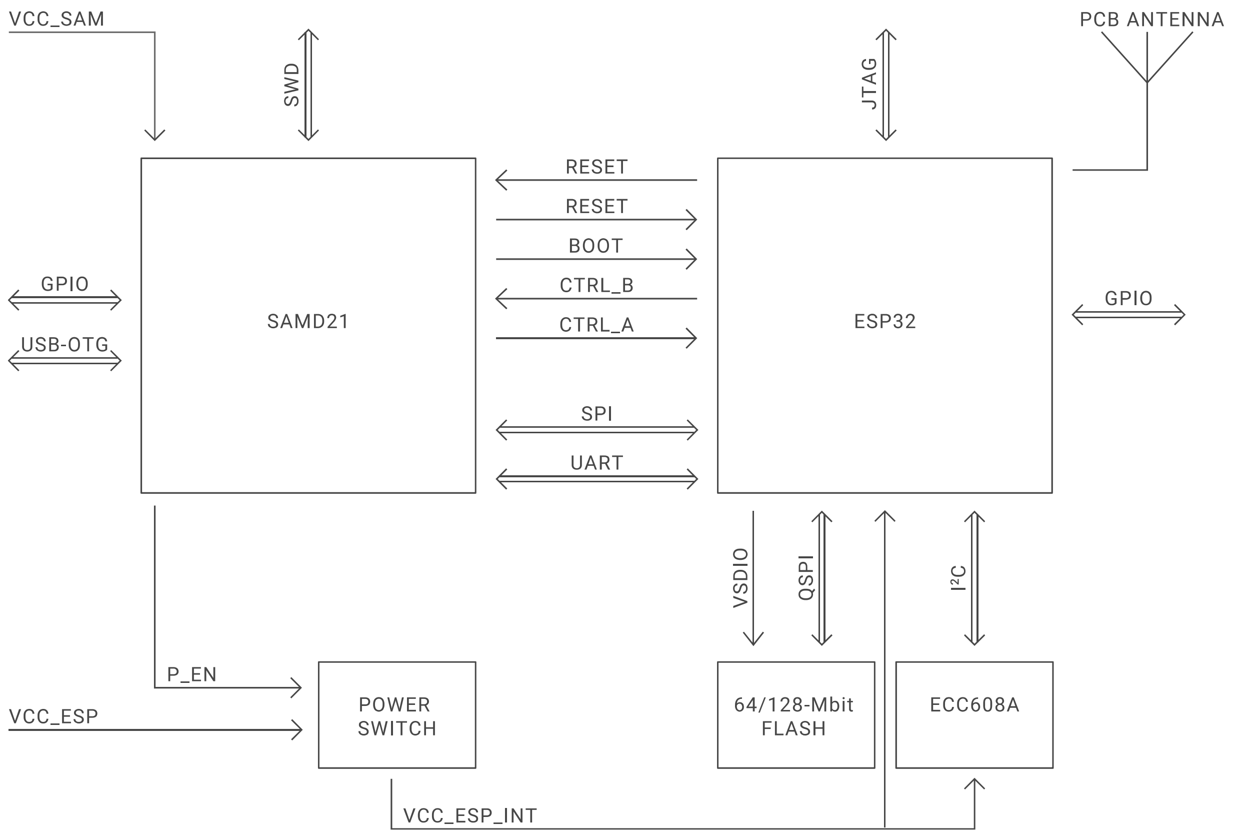


Figure 1 - Block Diagram

Pinout structure

The following image shows the pinout of the MBC-WB module.

The MBC standard has up to 62 pins: 14 of them dedicated to power supply, 2 dedicated to the USB interface, 1 for the board main reset, 1 for the ID, 1 for the optional external analog power supply reference, 14 can be both analog or digital GPIOs while the remaining 29 pins can be used as digital GPIOs.

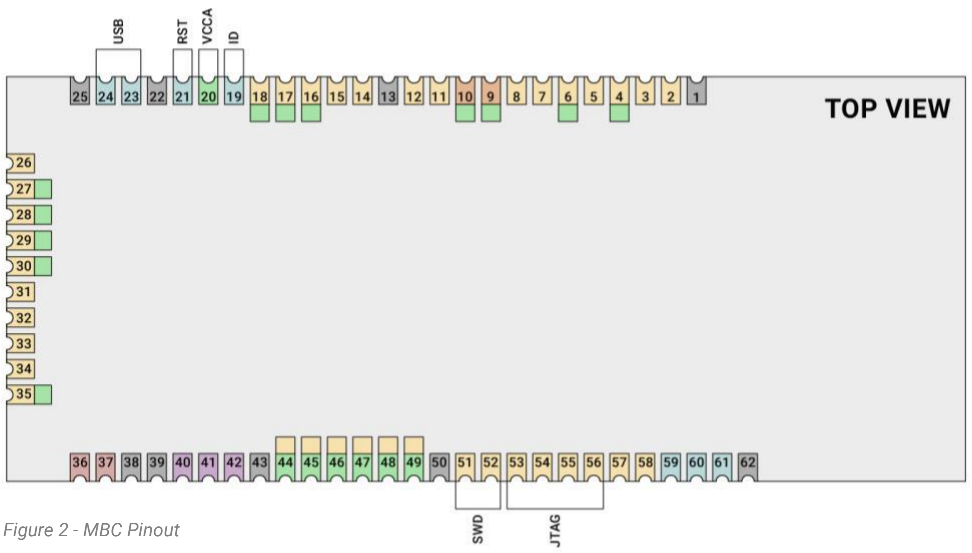


Figure 2 - MBC Pinout

- 🟥 SAMD21 power supply pin
- 🟪 ESP32 power supply pin
- ⬜ Ground pin
- 🟩 Analog Input pin (can also be used as digital GPIO)
- 🟨 Digital GPIO pin
- 🟧 Digital GPI pin
- 🟦 Special function pin - USB, SAMD21/ESP reset and ID

ESP32 Pinout Section

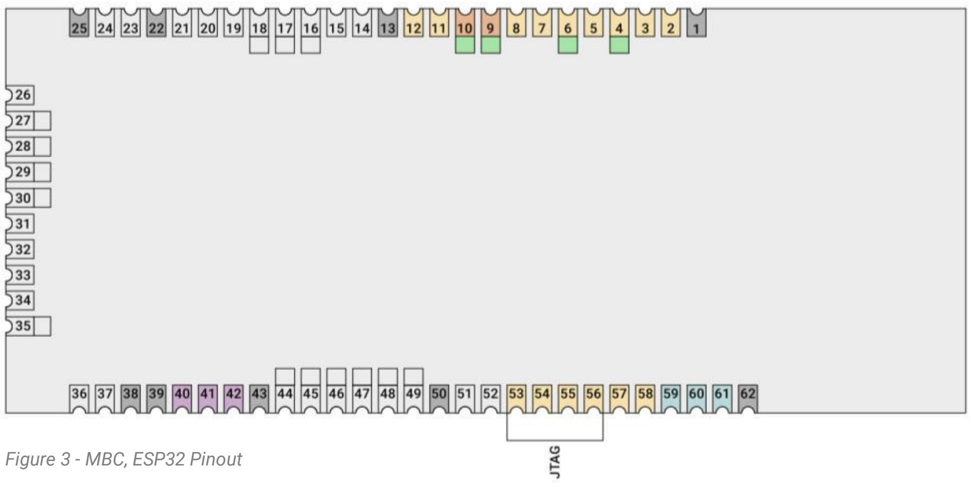


Figure 3 - MBC, ESP32 Pinout

Table 1 - ESP Pin description

pin #	Default Function	Alt. Functions / Notes			Dir	Voltage
		Analog	Digital	Special		
2	GPIO25	DAC_1, ADC2_CH8	GPIO25	RTC_GPIO6, EMAC_RXD0	I/O	3.3
3	GPIO27	ADC2_CH7	GPIO27	TOUCH7, RTC_GPIO17, EMAC_RX_DV	I/O	3.3
4	GPIO32	ADC1_CH4	GPIO32	32K_XP (32.768 kHz crystal oscillator input), TOUCH9, RTC_GPIO9	I/O	3.3
5	GPIO26	DAC_2, ADC2_CH9	GPIO26	RTC_GPIO7, EMAC_RXD1	I/O	3.3
6	GPIO33	ADC1_CH5	GPIO33	32K_XN (32.768 kHz crystal oscillator output), TOUCH8, RTC_GPIO8	I/O	3.3
7	U0TXD	-	GPIO1	CLK_OUT3, EMAC_RXD2	I/O	3.3
8	U0RXD	-	GPIO3	CLK_OUT2	I/O	3.3
9	GPI35	ADC1_CH7	GPI35	RTC_GPIO5	I	3.3
10	GPI34	ADC1_CH6	GPI34	RTC_GPIO4	I	3.3
11	GPIO4	ADC2_CH0	GPIO4	TOUCH0, RTC_GPIO10, HSPIHD, HS2_DATA1, SD_DATA1, EMAC_TX_ER	I/O	3.3
12	GPIO2	ADC2_CH2	GPIO2	TOUCH2, RTC_GPIO12, HSPIWP, HS2_DATA0, SD_DATA0	I/O	3.3
53	GPIO15	ADC2_CH3	GPIO15	TOUCH3, RTC_GPIO13, MTDO, HSPICS0, HS2_CMD, SD_CMD, EMAC_RXD3	I/O	3.3
54	GPIO13	ADC2_CH4	GPIO13	TOUCH4, RTC_GPIO14, MTCK, HSPID, HS2_DATA3, SD_DATA3, EMAC_RX_ER	I/O	3.3
55	GPIO12	ADC2_CH5	GPIO12	TOUCH5, RTC_GPIO15, MTDI, HSPIQ, HS2_DATA2, SD_DATA2, EMAC_TXD3	I/O	3.3
56	GPIO14	ADC2_CH6	GPIO14	TOUCH6, RTC_GPIO16, MTMS, HSPICLK, HS2_CLK, SD_CLK, EMAC_TXD2	I/O	3.3
57	GPIO16	-	GPIO16	HS1_DATA4, U2RXD, EMAC_CLK_OUT	I/O	3.3
58	GPIO17	-	GPIO17	HS1_DATA5, U2TXD, EMAC_CLK_OUT_180	I/O	3.3
59	ESP Power ON	Connected to PA28 of the SAMD21. Disable the control of this pin from the MBC's SAMD21 before driving externally. LOW = enable, HIGH = disable			I	3.3
60	ESP Reset	Connected to PA20 of the SAMD21. Disable the control of this pin from the MBC's SAMD21 before driving externally. LOW = reset, HIGH = run			I	3.3
61	ESP Boot	ADC2_CH1	GPIO0	TOUCH1, RTC_GPIO11, CLK_OUT1, EMAC_TX_CLK; Connected to PA15 of the SAMD21. Disable the control of this pin from the MBC's SAMD21 before driving externally; LOW = boot, HIGH = run	I/O	3.3

SAMD21 Pinout Section

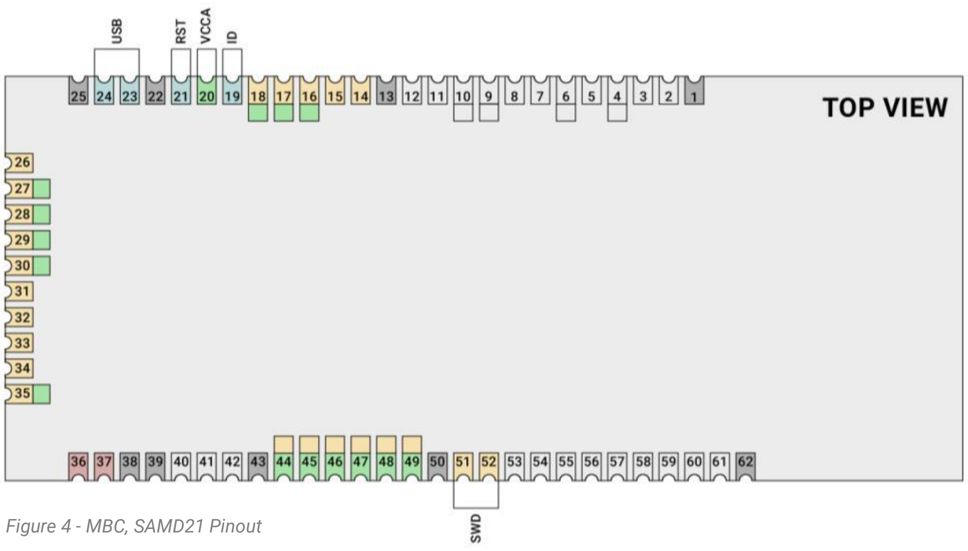


Figure 4 - MBC, SAMD21 Pinout

Table 2 - SAMD21 Pin Description

pin #	Default Function	Alt. Functions / Notes			Dir	Voltage
		Analog	Digital	Special		
14	PA23	-	PA23	EXTINT[7], PTC_X[11], SERCOM3/PAD[1], SERCOM5/PAD[1], TC4/WO[1], TCC0/WO[5], USB/SOF 1kHz, GCLK_IO(7)	I/O	3.3
15	PA14	-	PA14	EXTINT[14], SERCOM2/PAD[2], SERCOM4/PAD[2], TC3/WO[0], TCC0/WO[4], GCLK_IO[0]	I/O	3.3
16	PA10	AIN[18]	PA10	EXTINT[10], PTC_X[2], SERCOM0/PAD[2], SERCOM2/PAD[2], TCC1/WO[0], TCC0/WO[2], I2S/SCK[0], GCLK_IO[4]	I/O	3.3
17	PA09	AIN[17]	PA09	EXTINT[9], PTC_X[1], SERCOM0/PAD[1], SERCOM2/PAD[1], TCC0/WO[1], TCC1/WO[3], I2S/MCK[0]	I/O	3.3
18	PA08	AIN[16]	PA08	NMI, PTC_X[0], SERCOM0/PAD[0], SERCOM2/PAD[0], TCC0/WO[0], TCC1/WO[2], I2S/SD[1]	I/O	3.3
19	MBC's device ID	-	PA22	EXTINT[6], PTC_X[10] SERCOM3/PAD[0], SERCOM5/PAD[0], TC4/WO[0], TCC0/WO[4], GCLK_IO[6]	I/O	3.3
20	VCCA	Analog voltage reference for ADC/DAC			P	3.3
21	SAMD Reset	Connected with GPIO21 of the ESP32. Before driving externally ensure that GPIO21 is configured as open-drain. LOW = reset, HIGH = run			I	3.3
23	USB D+	-	PA25	EXTINT[13], SERCOM3/PAD[3], SERCOM5/PAD[3], TC5/WO[1], TCC1/WO[3]	I/O	3.3
24	USB D-	-	PA24	EXTINT[12], SERCOM3/PAD[2], SERCOM5/PAD[2], TC5/WO[0], TCC1/WO[2]	I/O	3.3
26	PA27	-	PA27	EXTINT[15], GCLK_IO[10]	I/O	3.3
27	PB02	AIN[10]	PB02	EXTINT[2], PTC_Y[8], SERCOM5/PAD[0], TC6/WO[0]	I/O	3.3
28	PB03	AIN[11]	PB03	EXTINT[3], PTC_Y[9], SERCOM5/PAD[1], TC6/WO[1]	I/O	3.3
29	PB08	AIN[2]	PB08	EXTINT[8], PTC_Y[14], SERCOM4/PAD[0], TC4/WO[0]	I/O	3.3
30	PB09	AIN[3]	PB09	EXTINT[9], PTC_Y[15], SERCOM4/PAD[1], TC4/WO[1]	I/O	3.3
31	PB11	-	PB11	EXTINT[11], SERCOM4/PAD[3], TC5/WO[1], TCC0/WO[5], I2S/SCK[1], GCLK_IO[5]	I/O	3.3

pin #	Default Function	Alt. Functions / Notes			Dir	Voltage
		Analog	Digital	Special		
32	PB10	-	PB10	EXTINT[10], SERCOM4/PAD[2], TC5/WO[0], TCC0/WO[4], I2S/MCK[1], GCLK_IO[4]	I/O	3.3
33	PA12	AC/CMP[0]	PA12	EXTINT[12], SERCOM2/PAD[0], SERCOM4/PAD[0], TCC2/WO[0], TCC0/WO[6]	I/O	3.3
34	PA13	AC/CMP[1]	PA13	EXTINT[13], SERCOM2/PAD[1], SERCOM4/PAD[1], TCC2/WO[1], TCC0/WO[7]	I/O	3.3
35	PA11	AIN[19]	PA11	EXTINT[11], PTC_X[3], SERCOM0/PAD[3], SERCOM2/PAD[3], TCC1/WO[1], TCC0/WO[3], I2S/FS[0], GCLK_IO[5]	I/O	3.3
44	AIN5	AIN[7], AC_AIN[3]	PA07	EXTINT[7], PTC_Y[5], SERCOM0/PAD[3], TCC1/WO[1], I2S/SD[0]	I/O	3.3
45	AIN4	AIN[6], AC_AIN[2],	PA06	EXTINT[6], PTC_Y[4], SERCOM0/PAD[2], TCC1/WO[0]	I/O	3.3
46	AIN3	AIN[5], AC_AIN[1]	PA05	EXTINT[5], PTC_Y[3], SERCOM0/PAD[1], TCC0/WO[1]	I/O	3.3
47	AIN2	ADC/VREFB, AIN[4], AC_AIN[0]	PA04	EXTINT[4], PTC_Y[2], SERCOM0/PAD[0], TCC0/WO[0]	I/O	3.3
48	AIN1	ADC/VREFA, DAC/VREFA, AIN[1]	PA03	EXTINT[3], PTC_Y[1]	I/O	3.3
49	AIN0	AIN[0], DAC_VOUT	PA02	EXTINT[2], PTC_Y[0]	I/O	3.3
51	SWDIO	-	PA31	EXTINT[11], SERCOM1/PAD[3], TCC1/WO[1]	I/O	3.3
52	SWDCLK	-	PA30	EXTINT[10], SERCOM1/PAD[2], TCC1/WO[0], GCLK_IO[0]	I/O	3.3

Boards pinout reference

MBC-WB

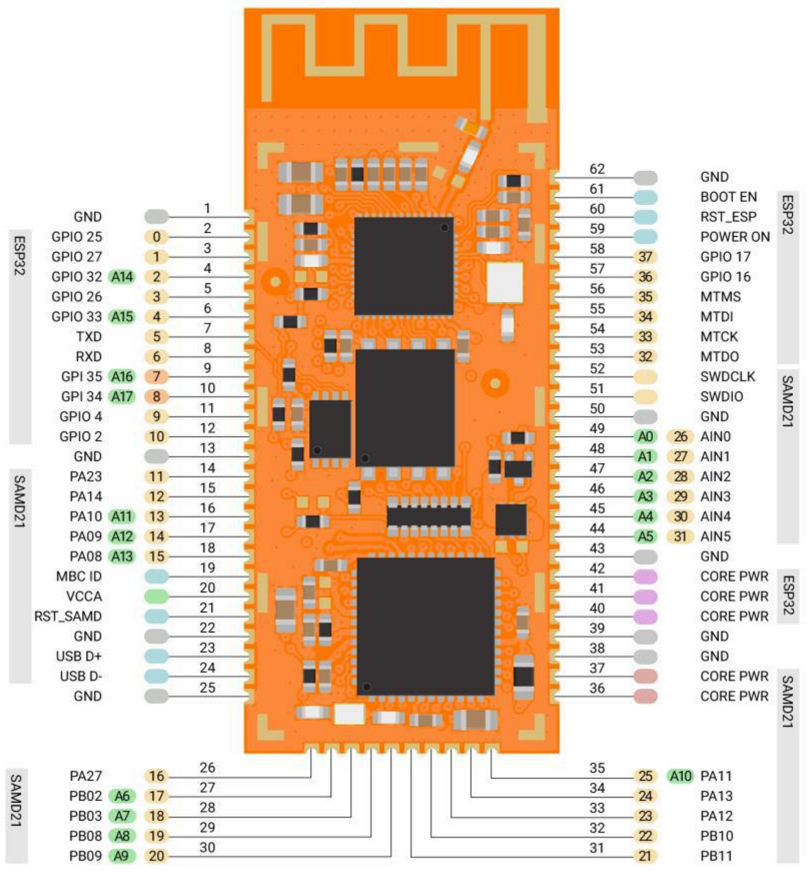


Figure 5 - MBC Pinout

1	Arduino® firmware pins
🟥	SAMD21 power supply pin
🟪	ESP32 power supply pin
⬜	Ground pin
🟩	Analog Input pin (can also be used as digital GPIO)
🟨	Digital GPIO pin
🟧	Digital GPI pin
🟦	Special function pin - USB, SAMD21/ESP reset and ID

ABC

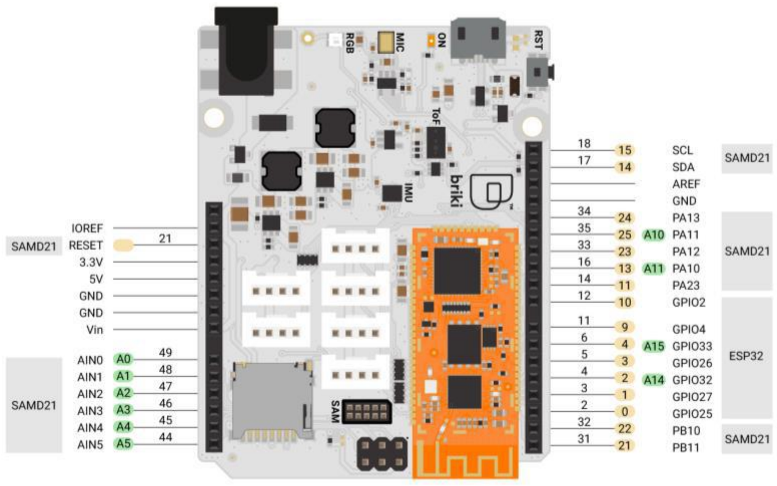
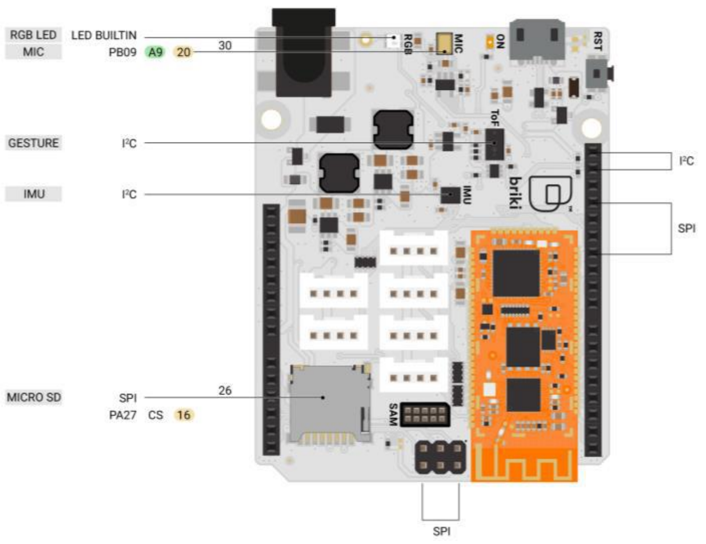


Figure 6 - ABC Front Pinout



1 Arduino® firmware pins

🟥 SAMD21 power supply pin
🟪 ESP32 power supply pin
⬜ Ground pin
🟩 Analog Input pin (can also be used as digital GPIO)
🟨 Digital GPIO pin
🟧 Digital GPI pin
🟦 Special function pin - USB, SAMD21/ESP reset and ID

Figure 7 - ABC Front HW Pinout

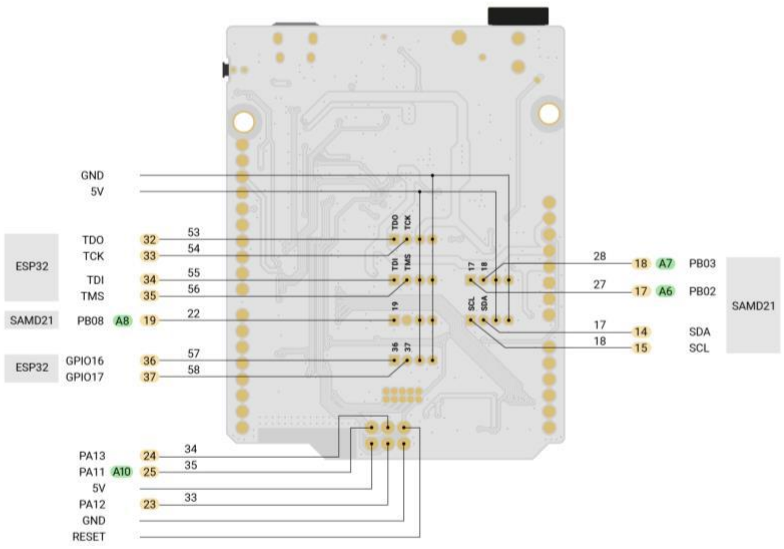


Figure 8 - ABC Back Pinout

DBC

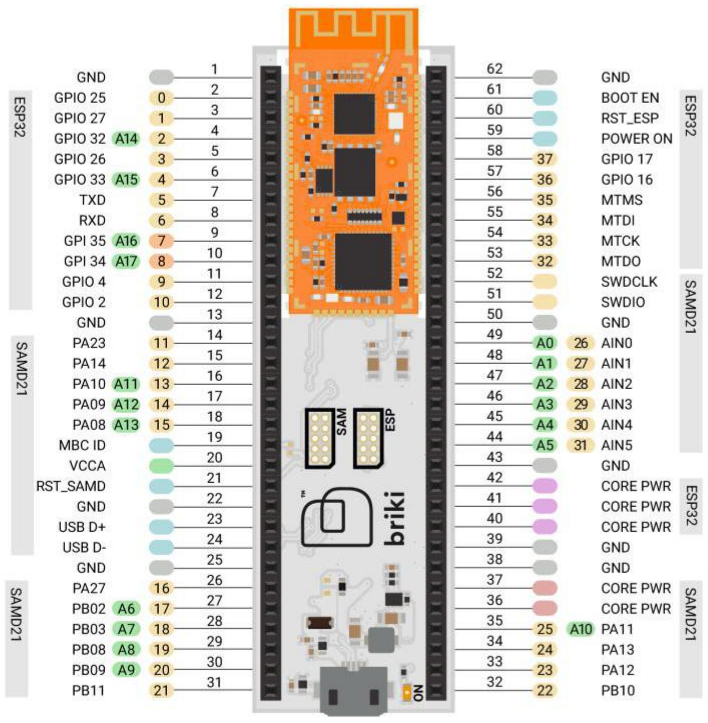


Figure 9 - DBC Pinout

1 Arduino® firmware pins

🟥 SAMD21 power supply pin
🟪 ESP32 power supply pin
⬜ Ground pin
🟩 Analog Input pin (can also be used as digital GPIO)
🟨 Digital GPIO pin
🟧 Digital GPI pin
🟦 Special function pin - USB, SAMD21/ESP reset and ID

Module geometry and PCB land pattern

The image below shows the MBC's physical dimensions, the pitch and pad's dimensions. A tolerance of about ±0.1mm should be considered.

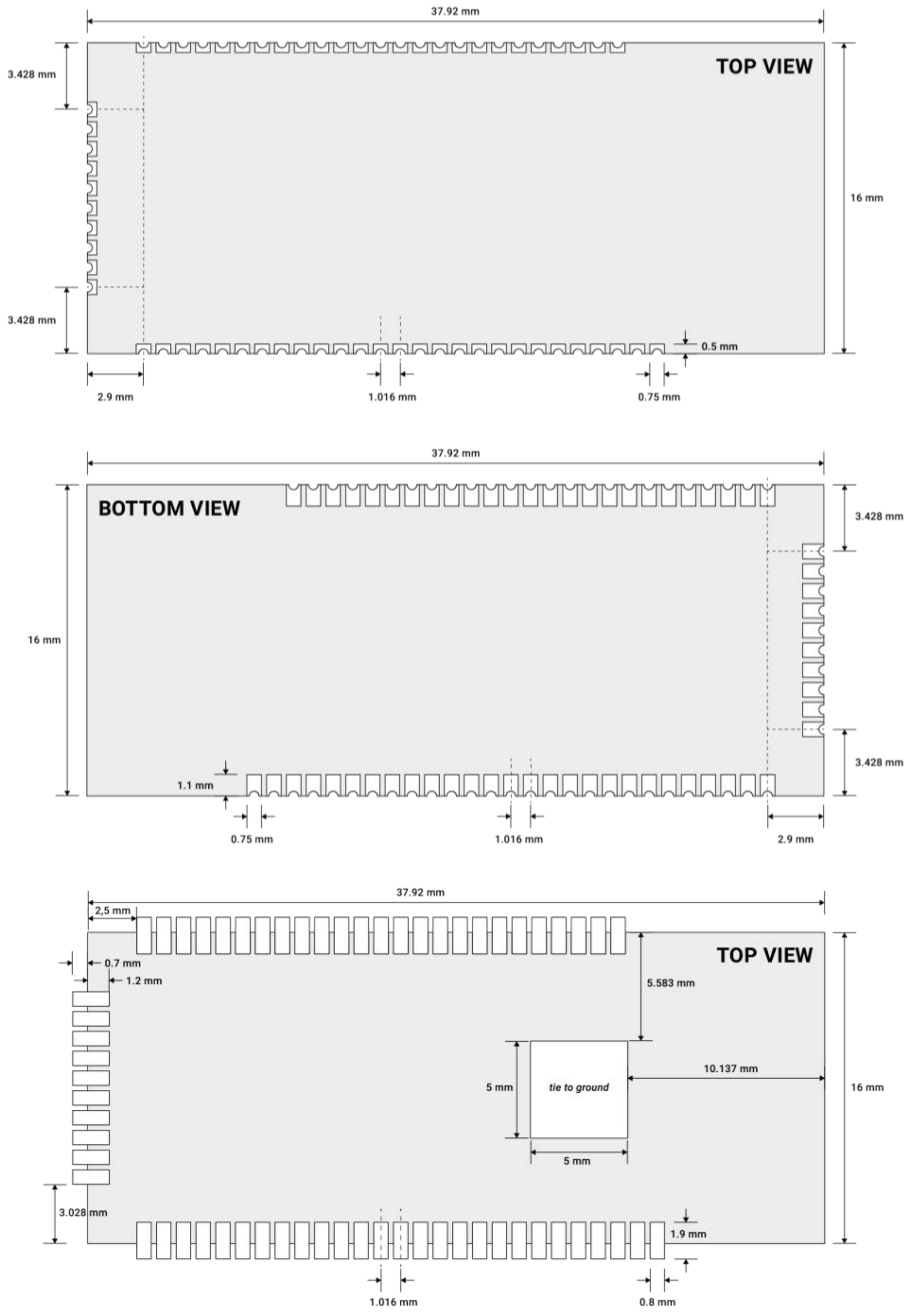
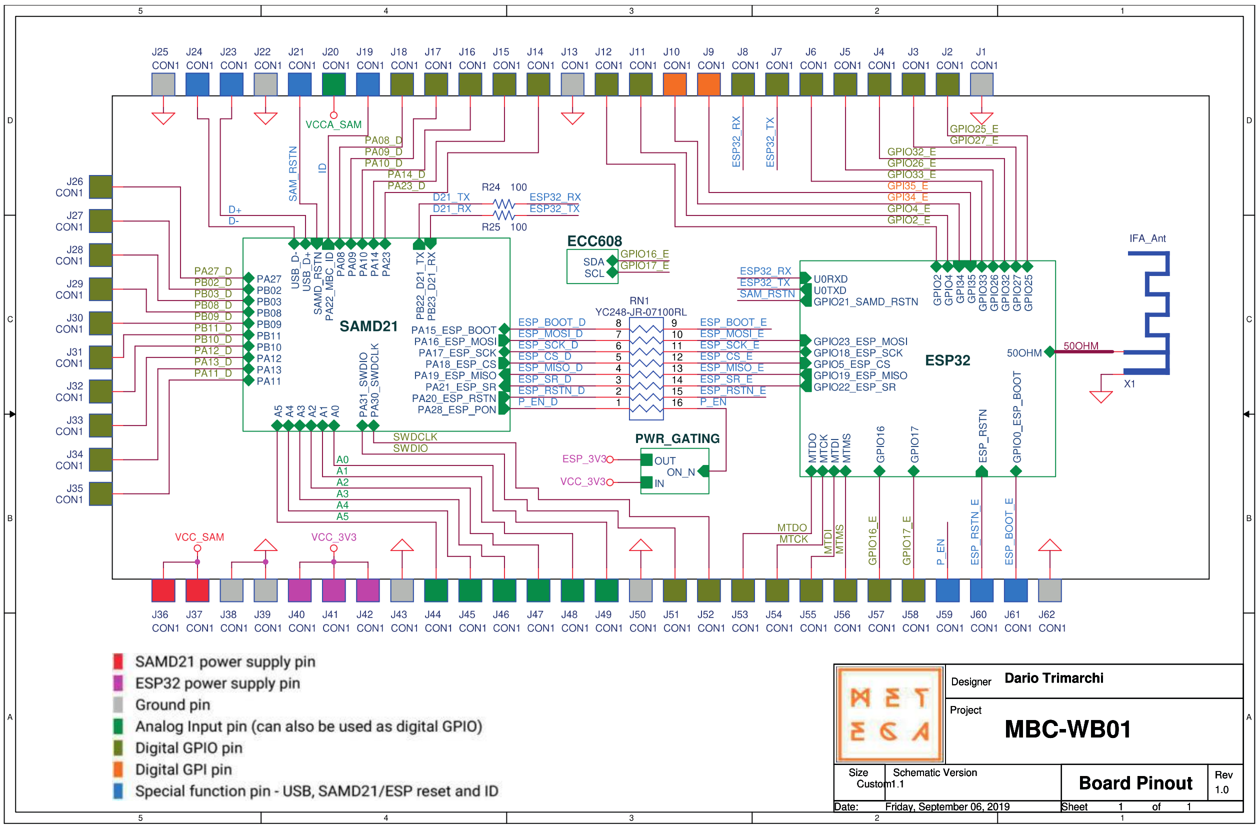
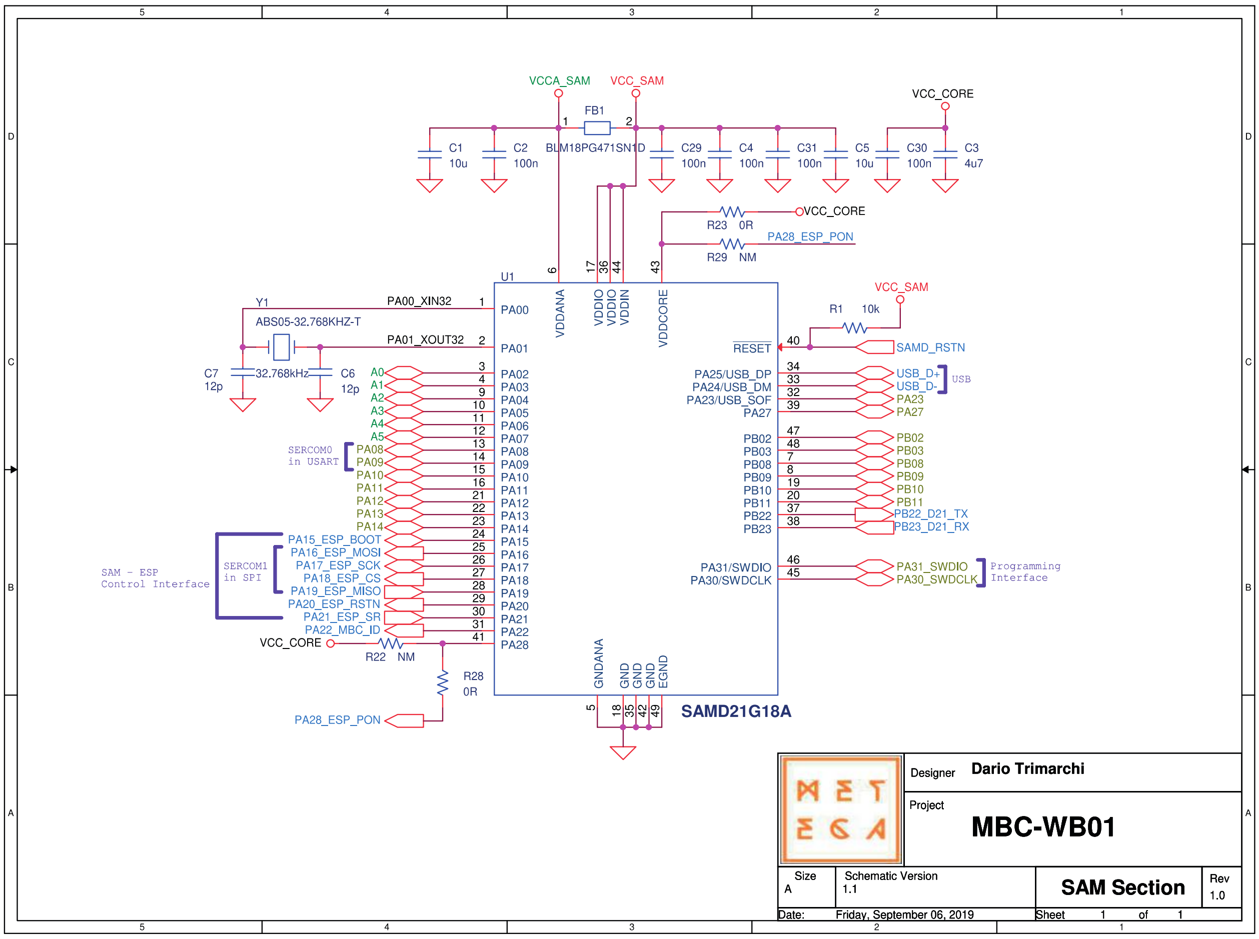


Figure 10 - MBC Dimensions and Land Pattern

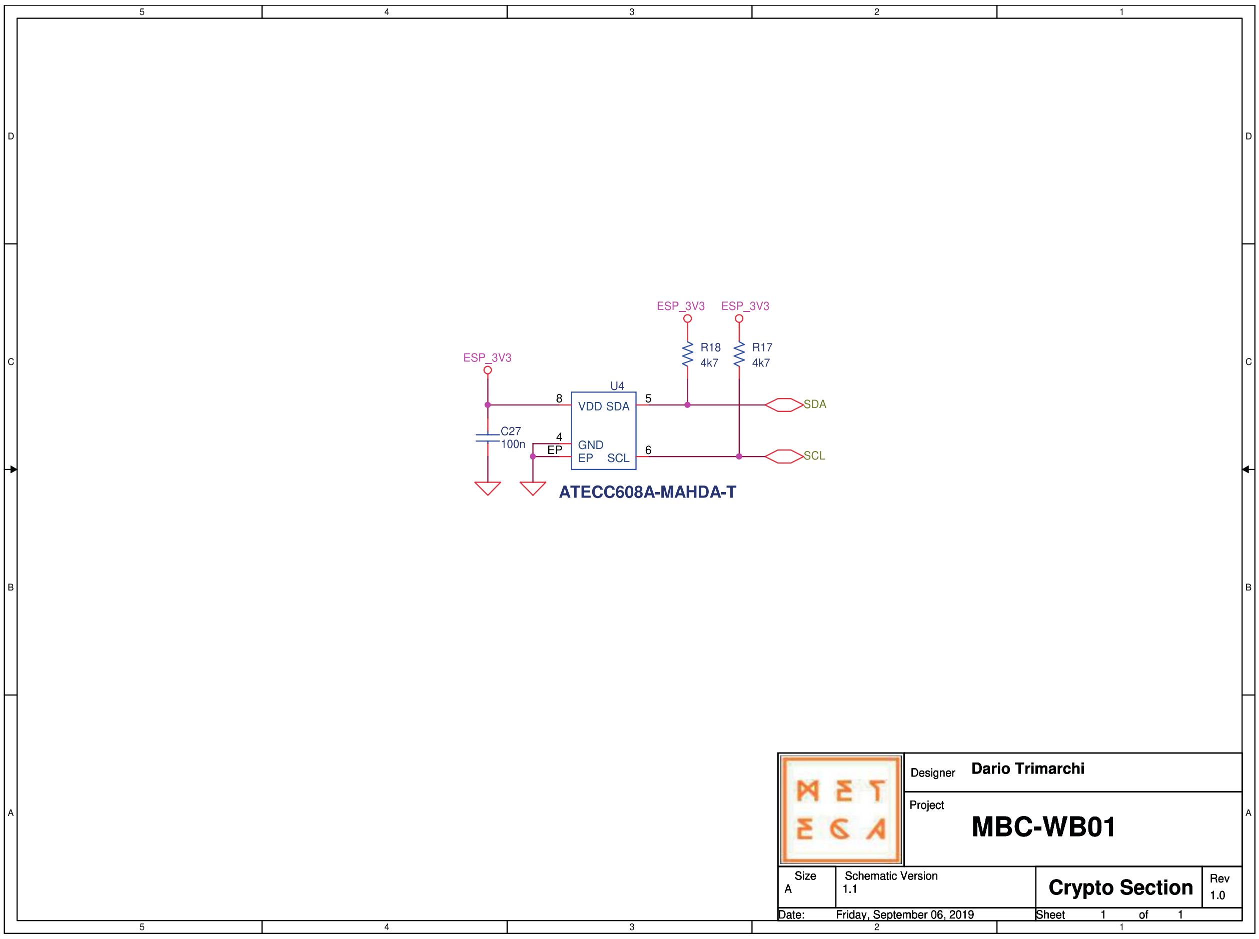
Schematic



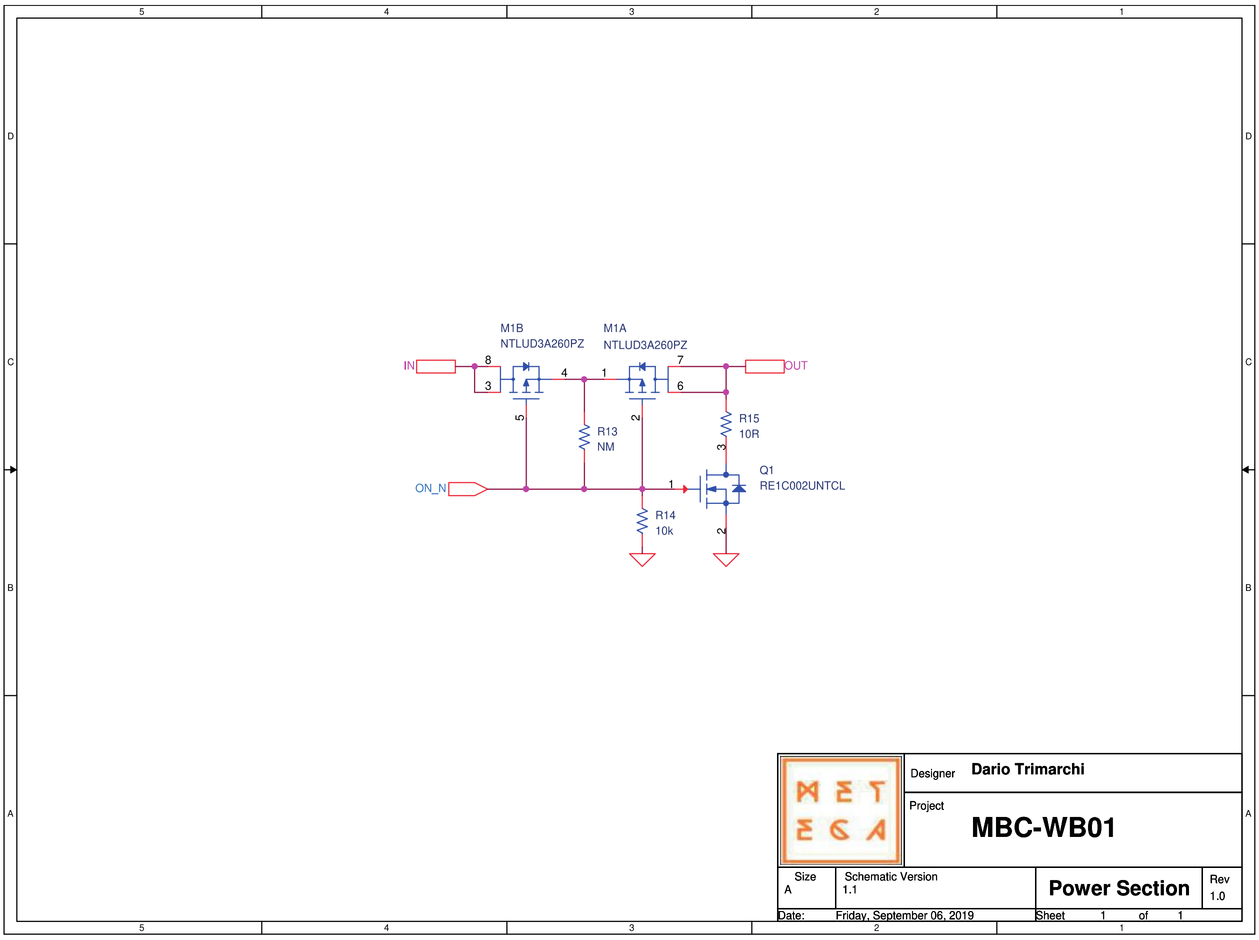
J1–J62 CON1
SAMD21
ESP32
ECC608
SDA
SCL
RN1
YC248-JR-07100RL
R24 100
R25 100
PWR_GATING
OUT
ON_N
IN
ESP_3V3
VCC_3V3
VCC_SAM
VCCA_SAM
IFA_Ant
50OHM
X1
🟥 SAMD21 power supply pin
🟪 ESP32 power supply pin
⬜ Ground pin
🟩 Analog Input pin (can also be used as digital GPIO)
🟩 Digital GPIO pin
🟧 Digital GPI pin
🟦 Special function pin - USB, SAMD21/ESP reset and ID
Designer Dario Trimarchi
Project MBC-WB01
Size Custom
Schematic Version 1.1
Board Pinout
Rev 1.0
Date: Friday, September 06, 2019
Sheet 1 of 1



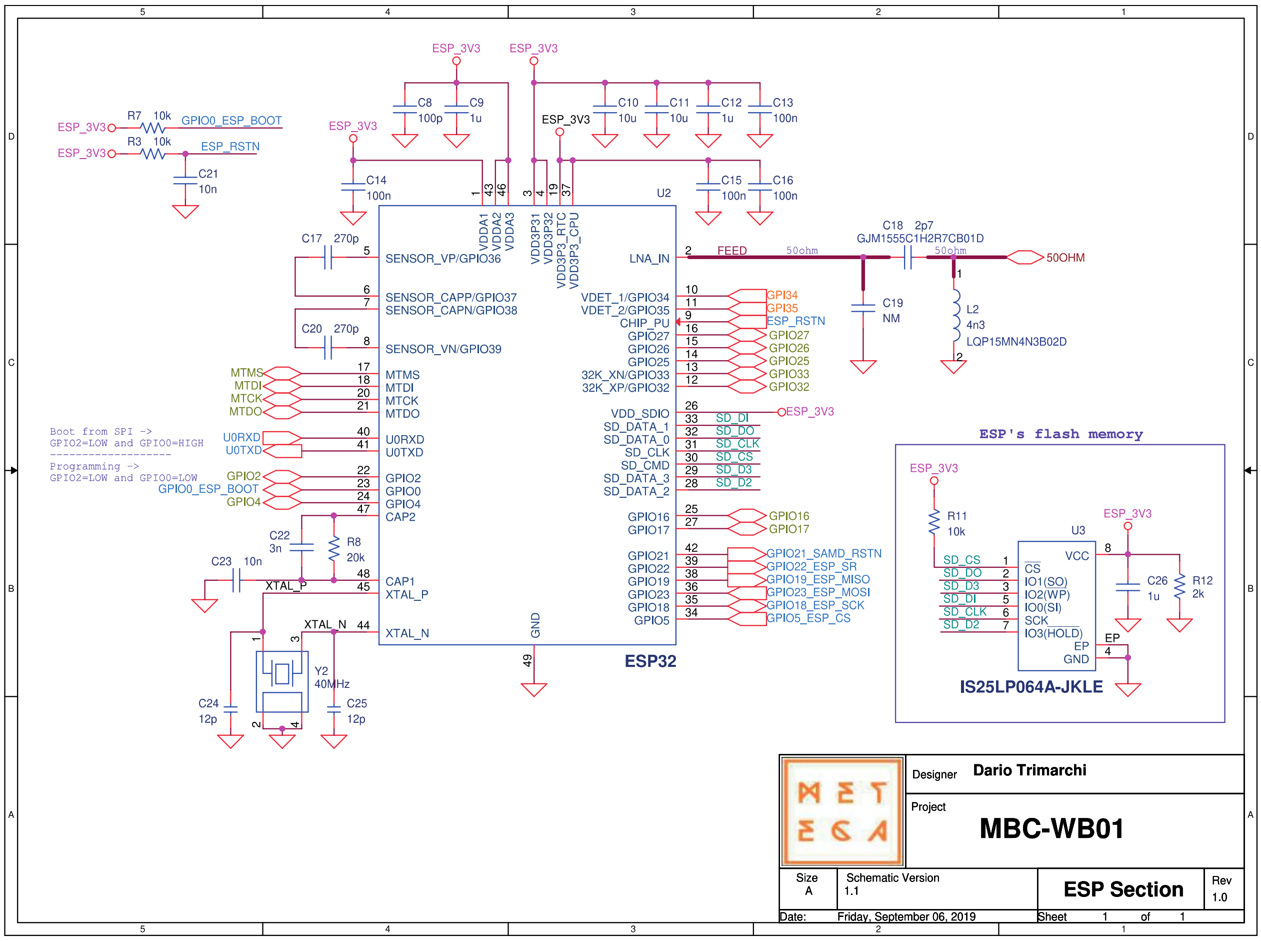
VCCA_SAM
VCC_SAM
VCC_CORE
FB1
BLM18PG471SN1D
C1 10u
C2 100n
C29 100n
C4 100n
C31 100n
C5 10u
C30 100n
C3 4u7
R23 0R
R29 NM
PA28_ESP_PON
U1
Y1
ABS05-32.768KHZ-T
32.768kHz
C7 12p
C6 12p
PA00_XIN32
PA01_XOUT32
R1 10k
RESET
SAMD_RSTN
USB_D+
USB_D-
USB
SERCOM0 in USART
SERCOM1 in SPI
SAM - ESP Control Interface
PA15_ESP_BOOT
PA16_ESP_MOSI
PA17_ESP_SCK
PA18_ESP_CS
PA19_ESP_MISO
PA20_ESP_RSTN
PA21_ESP_SR
PA22_MBC_ID
R22 NM
R28 0R
PB22_D21_TX
PB23_D21_RX
PA31_SWDIO
PA30_SWDCLK
Programming Interface
SAMD21G18A
VDDANA
VDDIO
VDDIN
VDDCORE
GNDANA
GND
EGND
Designer Dario Trimarchi
Project MBC-WB01
Size A
Schematic Version 1.1
SAM Section
Rev 1.0
Date: Friday, September 06, 2019
Sheet 1 of 1



ESP_3V3
ESP_3V3
ESP_3V3
R18
4k7
R17
4k7
U4
VDD SDA
GND
EP SCL
8
5
4
6
EP
C27
100n
SDA
SCL
ATECC608A-MAHDA-T
Designer Dario Trimarchi
Project
MBC-WB01
Size A
Schematic Version 1.1
Crypto Section
Rev 1.0
Date: Friday, September 06, 2019
Sheet 1 of 1



M1B
NTLUD3A260PZ
M1A
NTLUD3A260PZ
IN
OUT
8
3
5
4
1
2
7
6
R13
NM
R15
10R
Q1
RE1C002UNTCL
ON_N
R14
10k
Designer Dario Trimarchi
Project
MBC-WB01
Size A
Schematic Version 1.1
Power Section
Rev 1.0
Date: Friday, September 06, 2019
Sheet 1 of 1


ESP32
U2
IS25LP064A-JKLE
U3
ESP's flash memory
Boot from SPI -> GPIO2=LOW and GPIO0=HIGH
Programming -> GPIO2=LOW and GPIO0=LOW
ESP_3V3
GPIO0_ESP_BOOT
ESP_RSTN
Y2 40MHz
XTAL_P
XTAL_N
FEED
50ohm
50OHM
C18 2p7 GJM1555C1H2R7CB01D
L2 4n3 LQP15MN4N3B02D
C19 NM
GPIO21_SAMD_RSTN
GPIO22_ESP_SR
GPIO19_ESP_MISO
GPIO23_ESP_MOSI
GPIO18_ESP_SCK
GPIO5_ESP_CS
Designer Dario Trimarchi
Project MBC-WB01
Size A
Schematic Version 1.1
ESP Section
Rev 1.0
Date: Friday, September 06, 2019
Sheet 1 of 1

USB Interface

The Universal Serial Bus (USB) module of the SAMD21, complies with the Universal Serial Bus (USB) 2.1 specification, supporting both device and embedded host modes. It supports full (12Mbit/s) and low (1.5Mbit/s) speed communication and Link Power Management (LPM-L1) protocol. More information can be found in the section 32 of the device datasheet (pag. 788).

By default, the firmware implementation of the MBC-WB comes with a Dual CDC interface over the same physical USB peripheral: one for the SAMD21 itself and one for the ESP32 which lacks such a peripheral. For the latter, the SAMD21 acts as a USB-to-UART transceiver allowing a Host PC to access the ESP's UART interface, both for communication and programming. To modify this configuration, read the section SAMD21 CDC configuration.

On Windows, the USB driver discriminates between the two different interfaces embedded in the device descriptor of the MBC-WB. This leads to the enumeration of two separated USB ports: BRIKI MBC-WB (samd) for the SAMD21 and BRIKI MBC-WB (esp) for the ESP32.

In case a previous version of the driver has already been installed, the port names will not be displayed correctly. The suggested procedure is to connect the MBC-WB with an USB cable to the host PC, enter the Device Manager and, under the COM Port list, select the port belonging to the MBC. Now click on the right mouse button and select "Device uninstall" from the drop-down menu. Once the procedure has finished, disconnect the USB cable, launch the driver installer contained in the platform installation directory under the "driver" folder, and finally connect the board again. If the names are still not correct it is possible that the firmware on the target board is not updated to the last version. In that case, just simply upload a new firmware to the SAMD21 using one of the supported IDEs.

On other operating systems, such as Linux or MacOS, both ports show the same descriptor, and thus there is no way to distinguish them at a glance; usually the SAMD21 port is enumerated as first and the ESP32 port as second. So, the general rule to recognize which is which, is to look at the final number of the attached port. For example, under Linux, if the ports are enumerated as ttyACM0 and ttyACM1, it is likely they are referred to SAMD21 and ESP32 respectively, so the lower number is usually associated to the SAMD21.

A simple firmware that prints a different message on both the MCUs can be loaded to check how your system enumerated the two ports.

Regarding the firmware update via USB bootloader, it will be performed anyway, independently by which of the two port has been selected for. This means that the SAMD21 can be programmed even if the ESP32 port has been selected and vice versa. To better understand how the firmware update process works over the USB interface, please read the USB bootloader section.

SAMD21 USB-to-serial transceiver function

As already explained, in the MBC-WB, the SAMD21 forwards messages from the ESP32 to USB and vice versa. By default, it performs this job in the background, along with the handling of the other messages exchanged with the companion MCU to enable the GPIO virtualization and the Communication functions.

Since the SAMD21 comes with a single-thread firmware implementation (there is no RTOS that allows multi-threading as for the ESP32), the execution time of the code loop is affected by the execution time of each task it embeds. Thus, if a task requires a long time to be executed, this will have a negative impact on the overall system performances and consequently also on those tasks that runs in background.

To speed up the execution of all background tasks, the function communication.handleCommEvents() can be added in those functions that require a long time to end. This function will check for messages to be forwarded to/from USB as well as any other communication event that needs to be processed.

When the double USB-CDC interface is enabled, the communication between the two MCUs is performed via the shared SPI interface. If the shared UART interface must be used instead, for internal communication or any other reason, the secondary CDC interface can be disabled.

The board will then just show the SAMD21's USB-CDC interface. To disable the secondary CDC interface, please read the section SAMD21 CDC configuration.

Since the SAMD21 CDC is enabled in any case, with single or dual CDC interface, programming can still be performed on both MCUs.

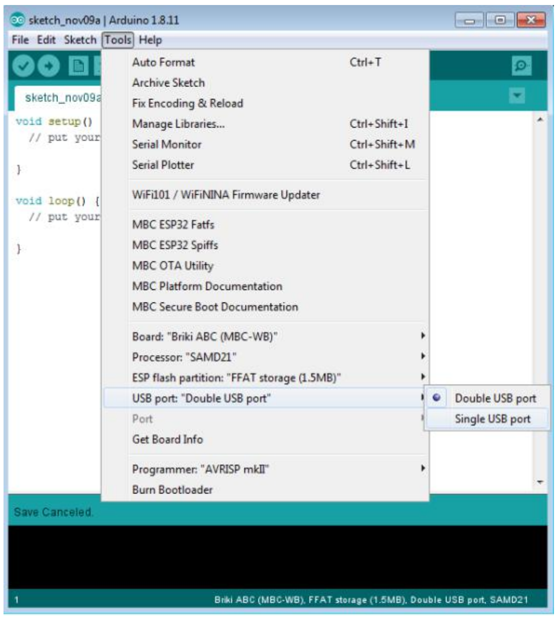
In case a single CDC interface is enable, the ESP32's serial interface can be mirrored on the SAMD21 USB interface using the UartBridge example supplied.

SAMD21 CDC configuration

As previously specified, the MBC-WB comes with a Dual-CDC implementation over the USB interface. This enables the direct communication between the ESP32 serial port and a Host PC, granting the typical ESP32 development boards experience.

The secondary CDC interface can be disabled in the following way:

— in the Arduino IDE: there is a dedicated menu under *Tools > USB Port: Single USB Port*


sketch_nov09a | Arduino 1.8.11
File Edit Sketch Tools Help
sketch_nov09a
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
MBC ESP32 Fatfs
MBC ESP32 Spiffs
MBC OTA Utility
MBC Platform Documentation
MBC Secure Boot Documentation
Port
Get Board Info
Burn Bootloader
Double USB port
Single USB port
Save Canceled.

```
void setup() {
  // put your

}

void loop() {
  // put your

}
```

Board: "Briki ABC (MBC-WB)"
Processor: "SAMD21"
ESP flash partition: "FFAT storage (1.5MB)"
USB port: "Double USB port"
Programmer: "AVRISP mkII"

1 Briki ABC (MBC-WB), FFAT storage (1.5MB), Double USB port, SAMD21

Figure 11 - Arduino IDE USB Port

— In PlatformIO under Visual Studio Code, an entry must be added to the platformio.ini file in a SAMD21's project:

```
build_flags = -D SINGLE_CDC
```

Once the number of USB ports has been selected (from Arduino or PlatformIO), loading any sketch on the SAMD21 will make the change effective.

SAMD21 USB bootloader

The bootloader flashed in the SAMD21 is a custom implementation of the standard SAM-BA bootloader firmware and is stored in its flash memory from the address 0x0000 to 0x2000.

Table 3 - SAMD21 firmware partition table

The default firmware uploaded in athe MBC-WB's SAMD21 has the following Partition Table:

Name	Type	Offset	Size	Flags
boot	data	0x0000	0x2000	-
app0	data	0x2000	0x3e000	-

The USB interface embedded in the SAMD21 can be used to upgrade the firmware of both microcontrollers. The MBC-WB shows different USB-CDC interfaces depending on the operating condition: one (SAMD21) or two (SAMD21 + ESP32) while the device is executing the application code and a third and different one when the bootloader is invoked.

For more information about the USB-CDC interfaces showed by the MBC-WB during the application execution, click here.

To manage the firmware update process on the MBC-WB, the Host PC must use custom software responsible for the communication with the bootloader, named mbctool.

SAMD21 firmware update

The mbctool invokes the SAMD21 bootloader with a specified triggering event on the USB. The SAMD21 received this event resets itself and runs the bootloader waiting for the upload process to start. The mbctool then starts downloading chunks of the binary to the bootloader that writes them directly to the target memory addresses. Once the entire firmware is written, verification is performed and, if everything is ok, the bootloader jumps to entry point address of the application.

ESP32 firmware update

In this case, after the bootloader is invoked, the mbctool specifies that the target device for the upload process is the ESP32. The SAMD21 then starts acting as an USB-to-UART transceiver: using a couple of GPIOs it puts the ESP32 in boot mode and then, through one of its SERCOM (SERCOM5), it streams data from the USB, to the ESP32's UART0 and vice versa. Once the whole firmware is uploaded and verified, a hard reset is performed and both the SAMD21 and ESP32 start executing their applications.

If the firmware loaded in the SAMD21 is corrupted or stops its execution for some reason (due to a crash or a deadlock) the USB connection will be lost, and the method described above to start the bootloader will not work anymore.

In this case, it is possible to manually invoke the bootloader by shorting the SAMD21 reset pin to ground twice in a brief period (<1s). In this way the bootloader will start again, and the USB connection will be restored, allowing to upload a new firmware image.

When the update process is managed internally by the MBC-WB itself, the ESP32 UART0, the GPIO ESP Reset, ESP Boot, ESP Power ON and SAMD Reset, which are all exposed on the module's pinout, are internally controlled and must be left unmanaged by an external board during the whole process. In case the firmware update process must be performed by an external logic, please disable the control of this peripheral and all the involved GPIOs inside the MBC-WB firmware (both in SAMD21 and ESP32) before starting.

SAMD21 USB bootloader update

In some cases, even the bootloader firmware can be updated, for example when new features have been released or when a user-customized bootloader must be used.

A copy of the last updated bootloader binary is always available within the mbc-wb platform. It is located in the platform installation directory under bootloaders/briki_mbc-wb folder.

To update a fresh bootloader, an external SWD programmer (like J-Link or similar one) can be used. The external programmer grants full access to the MCU structure, both in programming and debugging and it's the preferred method for this kind of update. Please, check the module pinout to locate SAMD21 programming pins.

For all those cases in which an external programmer is not available, it is possible to use a special firmware capable of directly writing a new bootloader in the SAMD21 MCU. This program is available at the following address for the Arduino platform: https://github.com/Meteca/samd21BootReplacement-arduino, while at the following one for the PlatformIO platform: https://github.com/Meteca/samd21BootReplacement-pio.

It is composed by 2 files: *bootReplacement.ino/ bootReplacement.cpp* (depending on the platform used, it is the sketch itself) and *binaryToWrite.h*, that contains the binary of the new bootloader. This repository is always updated to the latest bootloader version.

When an update to the SAMD21 bootloader is needed, the repository content can be downloaded, the bootReplacement sketch opened and compiled with the preferred IDE and eventually loaded into the SAMD21 microcontroller.

After having uploaded the above-mentioned firmware, to execute the bootloader update process, a serial monitor must be used and all the instructions shown must be followed. Once the procedure has come to an end, a new firmware can be uploaded to the SAMD21 to overwrite the bootReplacement firmware.

Please, note that this procedure requires care and attention, especially when an external programmer is not used!

In case something catastrophic happens while updating the bootloader section (e.g. bad firmware, or erasing the flash), the SAMD21 bootloader will not be invoked after the reset; no USB interface will be showed and any successive updates will not be possible. In this case only an external programmer can be used to restore the bootloader.

mbctool

The mbctool is a command line executable written in Python. A copy of this tool can be downloaded for Windows, Linux and Mac. It is the only software that must be used to perform a firmware update on the MBC-WB, whatever OS or IDE is used. It is based on bossac and a customized version of esptool and its tasks are:

- to generate the triggering event to invoke the SAMD21 bootloader; this is performed by opening and closing the USB connection at a given baud rate and with a specific timing. Part of the code, that underlies the application running in the SAMD21, is always listening for this event on the USB interface.
- to establish the communication with the bootloader, specifying which is the target device for the upload process SAMD21 or ESP32.
- wait for the MBC to be ready and finally launch the proper application, respectively bossac for the SAMD21 or esptool for the ESP32, that will upload the firmware.

Therefore, a copy of bossac and esptool command line applications is placed in the same folder of mbctool. All these three executables are directly embedded in the mbctool executable.

SAMD21 programming

To load a binary on the SAMD21 using the Command Line Interface, the following command can be used:

```
mbctool -d <samd> -p <port> -u <firmware.bin>
```

where: -d specifies the target device (--device); -p specifies the USB port the board is connected to (--port); -u specifies the file to be loaded (--upload).

ESP32 programming

By specifying different offsets in the esptool call, more than one partition can be programmed at the same time. To keep this feature, mbctool passes to esptool all the command line parameters after -u option.

For example, to program the bootloader of the ESP32, the following command can be issued:

```
mbctool -d <esp> -p <port> -u <0x1000 bootloader.bin>
```

where: -d specifies the target device (--device); -p specifies the USB port the board is connected to (--port); -u specifies offset and file to be loaded (--upload).

To also load the application and the partition table together with the bootloader, the following command can be used:

```
mbctool -d <esp> -p <port> -u <0x1000 bootloader.bin 0x10000 application.bin
0x8000 partitions.bin>
```

The whole command after -u option will not be parsed by mbctool, it will instead be passed to the write command of esptool as it is, allowing the maximum flexibility. No reset option can be specified for the write command, the ESP32 will be hard reset after every programming.

For the ESP32 only, the upload speed can be specified (it defaults to 115200 baud) with -s option. For example, to load a binary image at 1500000 baud the following command can be issued:

```
mbctool -d <esp> -p <port> -s 1500000 -u <0x10000 firmware.bin>
```

ESP32 OTA firmware update

The USB is not the only upgrade interface available in MBC-WB, there is also a Wi-Fi based OTA (Over-The-Air) procedure that can be used. The update mechanism, which is based on Espressif OTA functionality, allows the MBC-WB to update its own firmware during the normal execution of an application. This method relies on the ESP32 and allows to program both the MCU in every supported BSS operative mode: SoftAP, STA or SoftAP+STA.

Regardless of the application the ESP32 is executing, the OTA functionality is always reachable. This is ensured by a dedicated thread running on the ESP32 core 0; it is started by the following call placed in the app_main() function:

```
xTaskCreateUniversal(OTATask, "OTATask", 10000, NULL, 1, NULL, 0);
```

The triggered thread raises an AP named *MBC-WB-XXXXXX* - where *XXXXXX* represents the final part of the MBC-WB ESP32's MAC address. Moving the OTA task to a dedicated thread, ensures that the updating process will always be available, both in case the main application freezes and even if the user application doesn't use the Wi-Fi connection (as long as the firmware uploaded in either the ESP32 and the SAMD21 does not reset the board during the update process).

ESP32 programming

To work properly, the OTA requires configuring the Partition Table in the ESP32 memory with at least two "OTA app slot" partitions (i.e. ota_0 and ota_1) and an "OTA Data Partition". The OTA operation functions write a new app firmware image to whichever OTA app slot is not currently being used for booting. Once the image is verified, the OTA Data partition is updated to specify that this image should be used for the next boot.

The default firmware uploaded in the MBC-WB's ESP32 has the following Partition Table:

Table 4 - ESP32 firmware partition table

Name	Type	SubType	Offset	Size	Flags
nvs	data	nvs	0x9000	0x5000	-
otadata	data	ota	0xe000	0x2000	-
app0	app	ota_0	0x10000	0x340000	-
app1	app	ota_1	0x350000	0x340000	-
ffat	data	fat	0x690000	0x170000	-

SAMD21 programming

The OTA firmware update on the SAMD21 relies on the ESP32 and is performed by a library responsible for the update process that makes use of the UART interface shared between the two chips. Before starting the upload, it issues a reset on SAMD21 invoking its bootloader, which in this case does not fetch the new firmware from the USB interface, but from this UART. Once the bootloader is ready, the library leaves to the OTA task alone the control over the UART interface, avoiding the resource sharing among the other tasks. Without blocking the execution of the other concurrent tasks running in ESP32, care must be taken to ensure that none of them performs a reset, blocking the uploading process and leaving the SAMD21 stuck in the bootloader with an incomplete firmware image.

Fortunately, since the SAMD21 update is always mediated by its bootloader, which is placed in a write protected memory area, a new update process can always be done by resetting the chip and starting again from the beginning.

When the update process is managed internally by the MBC-WB itself, the ESP32 UART0, the GPIO ESP Reset, ESP Boot, ESP Power ON and SAMD Reset, which are all exposed on the module's pinout, are internally controlled and must be left unmanaged by an external board during the whole process. In case the firmware update process must be performed by an external logic, please disable the control of this peripheral and all the involved GPIOs inside the MBC-WB firmware (both in SAMD21 and ESP32) before starting.

The tool used to manage the Wi-Fi OTA communication is called BrikiOTA and can be used to program both the MCU.

BrikiOTA

The BrikiOTA Tool is a GUI program, callable from an IDE, written in Python and can be obtained from the Briki website for Windows, Linux and Mac. It is the only software that must be used to perform an OTA firmware update on the MBC-WB, whatever OS or IDE is used.

To upload a fresh firmware just compiled for a given MCU, it is sufficient to invoke the BrikiOTA tool from the chosen IDE. The right MCU and the generated binary have already been selected automatically once the GUI is shown.

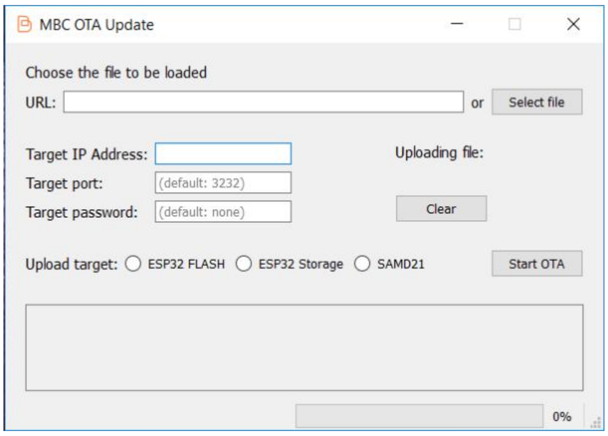


Figure 12 - MBC OTA GUI

Before starting the upload procedure, the user must ensure that the Host PC is connected to the same network the MBC-WB is connected to. The MBC's IP must be known (it must be inserted in the given field of the GUI) and must be reachable from the Host. In case the MBC-WB is not connected to any network yet, the Host PC can be connected to the MBC-WB's AP. In this case the default IP to enter in the GUI IP field is 192.168.240.1

Neither a port nor a password must be defined if the factory firmware is running on the MBC-WB. Finally, the "Start OTA" button can be pressed to start the uploading procedure.

To save power, it is possible to switch-off the default AP in the application sketch. To do so, the function `WiFi.mode()` can be called anywhere in the sketch. It is also possible to modify the AP's default configuration by using `setWiFiConfig()` function from `WiFi2Control` class.

```
#include "WiFi.h"
    void setup() {
    Wifi.mode(WIFI_MODE_NULL);
    }
    void loop(){}
```

Once the Wi-Fi is disabled, **OTA Update will no longer be available**. To enable it again, a fresh firmware without the call to the previous function must be uploaded to ESP32 over USB.

When the command line interface is to be preferred over a GUI program, for instance when a user wants to automate firmware update procedures, a modified version of the Espressif's tool espota can be used.

Espota is a command line tool written in python by Espressif to perform OTA update on the ESP32. This tool has been customized to upload a firmware on the SAMD21 too. The customized version of this tool is available in the *platform installation directory*, under the *tools* directory. The tool can program the ESP32 with the following command:

```
python espota.py -i <ESP_IP_Address> -p <ESP_OTA_PORT> -f <application.bin>
```

where: -i specifies the ESP32's IP address (–*ip* defaults to 192.168.240.1 when the default Access Point is used), -p specifies the OTA port the ESP is listening to (--*port* default is 3232) and -f is the binary file to be loaded (--*file*).

The tool can also program the SAMD21 by issuing the same command, adding the -*c* option:

```
python espota.py -i <ESP_IP_Address> -p <ESP_OTA_PORT> -c -f <application.bin>
```

where -c specifies that programming should be done on companion chip, SAMD21 (--*companion*).

If you want to update ESP32 storage partition, the command to be issued is similar to the previous one. This time you will need to use -*s* option:

```
python espota.py -i <ESP_IP_Address> -p <ESP_OTA_PORT> -s -f <storage.bin>
```

where -s option can be used to program the storage partition independently by the type of partition (SPIFFS or FAT).

In the Arduino IDE, the serial port monitor must be closed and re-opened to read the serial output when the SAMD21 has been updated via OTA.

Firmware platform

Installation prerequisites

To simplify the platform installation, Meteca provides an executable installer for all the main OSs. It contains the required drivers (only for the Windows version), the firmware platform as well as all the software tools needed for configuring and programming the MBC-WB.

Before launching the installer, one of the two supported IDEs must be installed: the Arduino IDE or Visual Studio Code with the PlatformIO plugin. The former is more suited for beginners while the latter is recommended as a more professional and complete IDE.

In case of any issue with the installer, or in case a manual installation is required, please check the manual installation section.

Arduino platform automatic installation

The Arduino IDE can be downloaded from the official website: https://www.arduino.cc/en/Main/Software.

Once the Arduino IDE has been installed, download the MBC-WB platform installer for the preferred OS:

Windows:	https://www.briki.org/download/resources/briki_arduino_installer.exe
Linux 64 bit:	https://www.briki.org/download/resources/briki_arduino_installer_linux64.tar.gz
Linux 32 bit:	https://www.briki.org/download/resources/briki_arduino_installer_linux32.tar.gz
MAC OSX:	https://www.briki.org/download/resources/briki_arduino_installer_osx.tar.gz

Before launching the installer, make sure that:

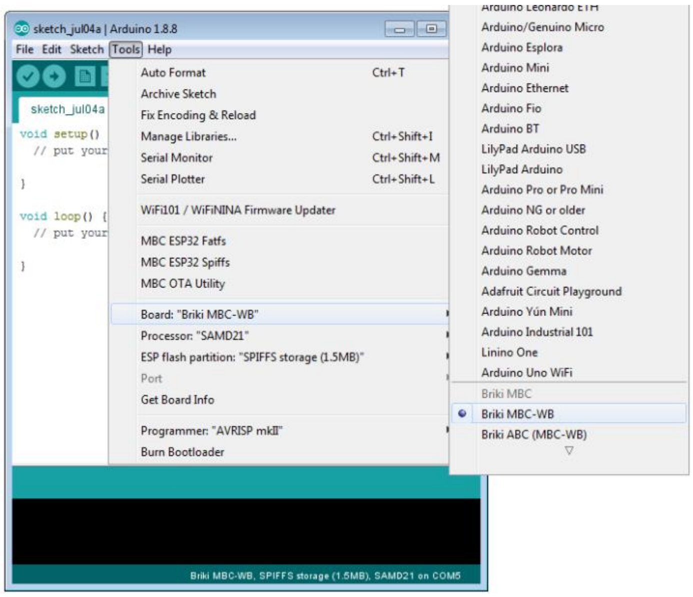
- the Arduino IDE is closed, or it will overwrite the settings the installer makes
- the Host PC has a stable connection to the internet since the installer will download the platform from the web
- on Linux and Mac systems the downloaded archive has been extracted before launching it

When ready, the installation can be started by double clicking on the installer executable.

Under Linux and Mac OS, in case of issues with the installer when launched using the UI, a terminal can be opened to drag and drop the file there. Pressing the enter key will then start the installation.

For Linux users: to complete the installation, serial port permission must be set. Please, read this reference or run arduino-linux-setup.sh script, that can be found in the Arduino IDE folder.

At the end of the installation procedure, the Arduino IDE can be opened to check if everything went well. Looking at Tools → Boards menu the Briki MBC-WB should be listed among the other boards.



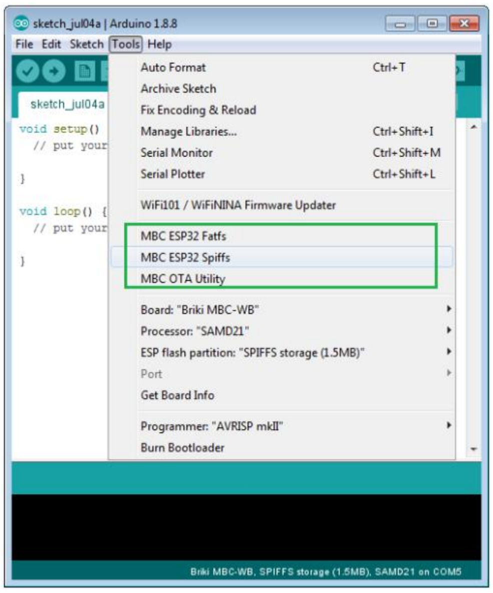
```
void setup() {
  // put your
}

void loop() {
  // put your
}
```

Figure 13 - Arduino IDE Board

To verify the correct configuration of the entire system (platform, drivers, mbctool etc.), the loading of a sketch on both the MCUs is recommended.

After selecting the correct microcontroller from the menu *Tools → Processor*, click on the compile and upload button to upload the current sketch to the given MCU.

The installer will also install additional tools (like the one for performing OTA updates). To check that tools are correctly installed, open the Tools menu:


sketch_jul04a | Arduino 1.8.8
File Edit Sketch Tools Help
sketch_jul04a
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Manage Libraries... Ctrl+Shift+I
Serial Monitor Ctrl+Shift+M
Serial Plotter Ctrl+Shift+L
WiFi101 / WiFiNINA Firmware Updater
MBC ESP32 Fatfs
MBC ESP32 Spiffs
MBC OTA Utility
Board: "Briki MBC-WB"
Processor: "SAMD21"
ESP flash partition: "SPIFFS storage (1.5MB)"
Port
Get Board Info
Programmer: "AVRISP mkII"
Burn Bootloader
Briki MBC-WB, SPIFFS storage (1.5MB), SAMD21 on COM5

```
void setup() {
  // put your
}

void loop() {
  // put your
}
```

Figure 14 - Arduino IDE additional tools

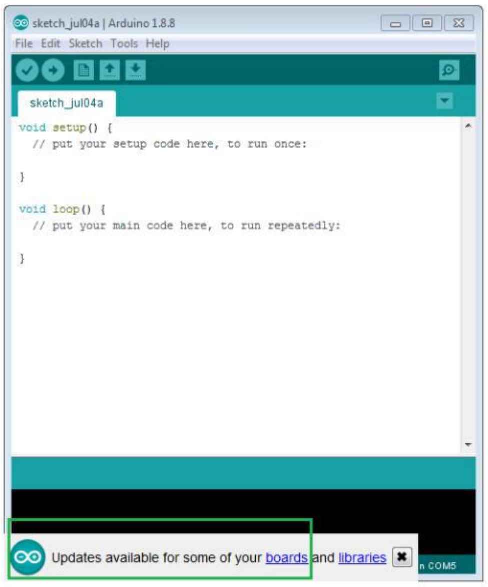
If the Arduino sketchbook folder has been changed from its default location, the installer will fail to install the additional tools and a manual process is required to install them.

The Windows installer also includes the USB drivers. Once an MBC board is connected to the USB, if its associated COM port is not listed under Tools- Port menu, this means that something probably went wrong during the installation process. In this event, please consult the troubleshooting section.

By selecting the MBC-WB board from the Tools menu, a wide range of examples will be available under *File → Examples → Examples for Briki MBC-WB*. Please note that the section *Examples for Briki MBC-WB* will only become available when Briki MBC-WB is the board selected under the Tools menu.

Arduino platform update

The Briki installer will automatically insert the Briki boards definition inside the Board Manager of the Arduino IDE. This means that, as soon as a new update is available, a notification will be displayed by the Arduino IDE itself.



```
void setup() {
  // put your setup code here, to run once:

}

void loop() {
  // put your main code here, to run repeatedly:

}
```

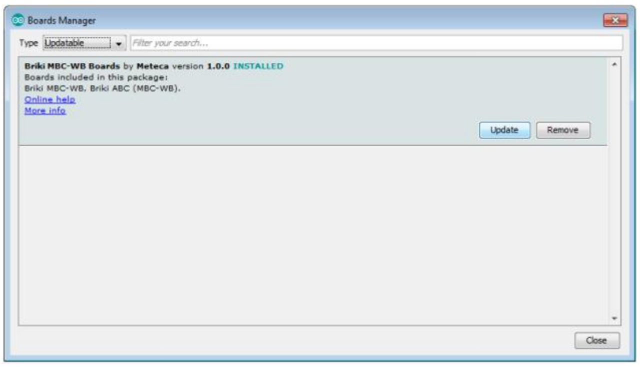
Updates available for some of your boards and libraries

Figure 15 - Arduino IDE platform update

When a new update becomes available a choice can be made: to update the platform

- by using the Arduino IDE Board Manager
- by using the installer used for the first installation.

In the Arduino IDE, it is only necessary to click on the “boards” link shown in the previous picture. The Boards Manager window will appear showing all the platforms that can be updated. By clicking on the Update button, the process will start.



Briki MBC-WB Boards by **Meteca** version **1.0.0** INSTALLED
Boards included in this package:
Briki MBC-WB, Briki ABC (MBC-WB).
Online help
More info

Figure 16 - Arduino IDE Boards Manager

Updating the platform with Boards Manager will not update the additional tools you can see in Tools menu. Updates to additional tools rarely happens but, if this is the case, a manual update is required.

To be sure that everything is up to date, including the firmware platform as well as all the tools, the update procedure must be done using the Briki installer supplied. It will always fetch the latest released version of the software package, automatically installing the tools and the platform in the right location, without the user's intervention.

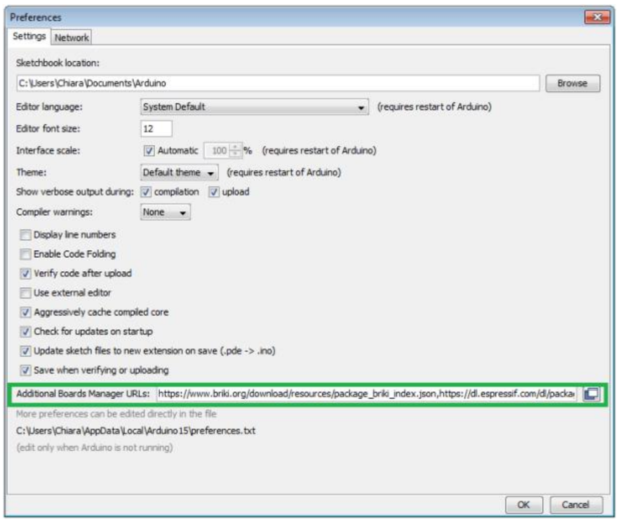
Please remember to close the Arduino IDE before running the installer.

Arduino platform manual installation

To manually install the firmware platform in the Arduino IDE, the first step is to add the Briki board definition inside the Boards Manager of the Arduino IDE.

To do this, from the Arduino IDE open *File → Preferences* menu (or *Arduino → Preferences* in Mac OSx) and paste the following links inside the Additional Board Manager URLs field:

https://www.briki.org/download/resources/package_briki_index.json
https://dl.espressif.com/dl/package_esp32_dev_index.json



Sketchbook location:
C:\Users\Chiara\Documents\Arduino
Editor language: System Default (requires restart of Arduino)
Editor font size: 12
Interface scale: ✓ Automatic 100 % (requires restart of Arduino)
Theme: Default theme (requires restart of Arduino)
Show verbose output during: ✓ compilation ✓ upload
Compiler warnings: None
☐ Display line numbers
☐ Enable Code Folding
✓ Verify code after upload
☐ Use external editor
✓ Aggressively cache compiled core
✓ Check for updates on startup
✓ Update sketch files to new extension on save (.pde -> .ino)
✓ Save when verifying or uploading
Additional Boards Manager URLs: https://www.briki.org/download/resources/package_briki_index.json,https://dl.espressif.com/dl/packa
More preferences can be edited directly in the file
C:\Users\Chiara\AppData\Local\Arduino15\preferences.txt
(edit only when Arduino is not running)

Figure 17 - Arduino IDE platform manual installation

Once the OK button has been pressed, all the changes will take effect. The second step is to open the Boards Manager in *Tools → Board → Boards Manager*. At this point it is necessary to search for the MBC-WB board by typing Briki or MBC in the search field or simply scrolling through the list of boards until the item Briki MBC-WB Boards by Meteca has been shown. Finally, the install button must be pressed to start the installation.

As soon as the Boards Manager completes the installation, it is possible to compile and upload sketches on the board. However, the additional tools, as well as the USB driver (in case a Windows system is in use), still need to be installed. Both are located inside the platform installation directory (its location may vary based on the OS used).

On Windows, the platform resides under the hidden folder AppData located under User folder. The display of hidden items must be enabled to browse in the platform installation directory.

On Windows, the location of the platform installation directory is the following:
C:\Users\CurrentUser\AppData\Local\Arduino15\packages\briki\hardware\mbc-wb\1.x.x\

On Linux, the platform installation directory is inside the hidden folder .arduino15 located under home directory.
The final location of the platform is the following: *~/.arduino15/packages/briki/hardware/mbc-wb/1.x.x/*

On Mac OS, the platform installation directory is placed inside the hidden folder Library in the home folder.
The final location in this case is the following:
/Users/CurrentUser/Library/Arduino15/packages/briki/hardware/mbc-wb/1.x.x/

Additional tools installation

Using the Briki executable installer to perform both installation and update, tools are installed in addition to the firmware platform. Using the Boards Manager instead, the installation must be performed manually following the operations described below.

The additional tools supplied with the platform extend the capability to interact with the MBC-WB board beyond the standard Arduino experience. The tools currently available can be used to perform OTA updates on both MCUs and to update the storage partition of the ESP32 MCU (SPIFFS or FAT).

If the Arduino IDE is open, close it. Then locate the platform installation directory. Inside this folder, look for the "tools" folder and within this a folder called "extensions". To proceed with the installation, locate the Arduino sketchbook folder. The default folder location is Documents/Arduino for Windows and Mac OS and ~/Arduino for Linux. Inside the Arduino sketchbook folder, create a new folder named "tools" (if you already have a tools folder inside the sketchbook directory you can skip this step). The final path should look like the following:

Documents/Arduino/tools/

Finally, just copy the contents of the "extensions" folder and paste it to the tools folder. In the Arduino IDE, all the tools installed should appear under the Tool menu.

Drivers installation

Using Linux or Mac, no driver installation is required to start using an MBC-WB board. On the other hand, Windows requires the installation of the proper USB drivers to make the Host PC interact with the board through the USB connection. Drivers can be found in the MBC-WB installation directory, under “drivers” folder. The path should look like the following:

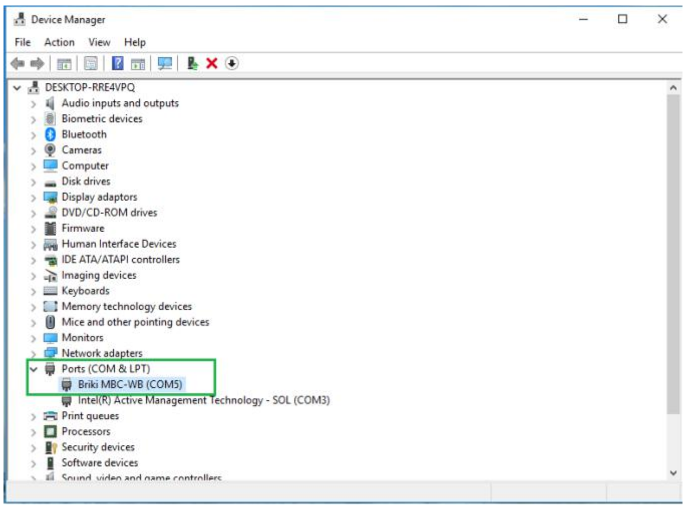
C:\Users\CurrentUser\AppData\Local\Arduino15\packages\briki\hardware\mbc-wb\x.x.x\drivers

Search this folder for the file *briki_drv_installer.exe* and execute it to start the driver installation. After the process is completed, the MBC-WB board can be connected to the Host PC, and the associated COM ports should be listed under the Device Manager. You can use one of these ports to interact with your board. The number of the available ports depends on the firmware loaded in the SAMD21 as depicted in this section.

To check the correct installation of drivers, it is recommended to try loading a sketch on the board (on both the SAMD21 and the ESP32)

Troubleshooting with the driver’s installation

If the sketch upload fails, the first thing to do is check if the drivers are installed correctly. Open the Windows Device Manager on the Host PC with the board connected via USB and check if a window like the following is prompted:



```
DESKTOP-RRE4VPQ
  Audio inputs and outputs
  Biometric devices
  Bluetooth
  Cameras
  Computer
  Disk drives
  Display adaptors
  DVD/CD-ROM drives
  Firmware
  Human Interface Devices
  IDE ATA/ATAPI controllers
  Imaging devices
  Keyboards
  Memory technology devices
  Mice and other pointing devices
  Monitors
  Network adapters
  Ports (COM & LPT)
    Briki MBC-WB (COM5)
    Intel(R) Active Management Technology - SOL (COM3)
  Print queues
  Processors
  Security devices
  Software devices
  Sound, video and game controllers
```

Figure 18 - Windows driver's troubleshooting

To invoke the Windows Device Manager on Windows 10:

- press W + X and select Device Manager
- from a command prompt write devmgmt.msc and press enter
- click the start button on the Taskbar and search for devmgmt.msc

In case the Briki MBC-WB board is not listed under the Ports (COM & LPT) – like in the following picture, it probably means the drivers install procedure went wrong.

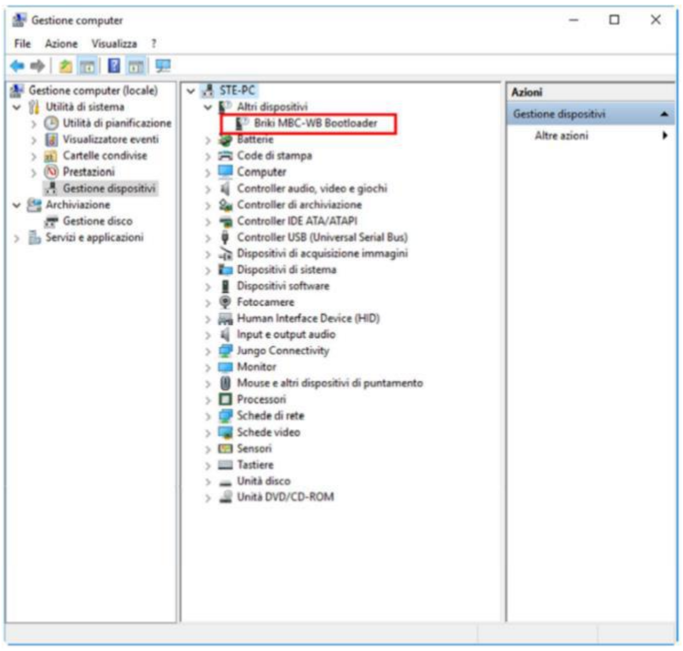


Figure 19 - Windows driver's Installation

In this case, try to manually install the drivers to fix the problem. From the Device Manager window, open Briki MBC-WB device properties (right click on Briki MBC-WB and then select Properties). From the popup window that will appear, select the *Driver* tab and then click on the *Update Driver*. When prompted, select the *Browse my computer for driver software* option. Then search for the driver's folder of MBC-WB platform (the same as the one where briki_drv_installer.exe is located) to start the installation.

At the end of the installation procedure, try to unplug and re-plug in the board on the Host PC and check if a new COM Port has now been correctly enumerated.

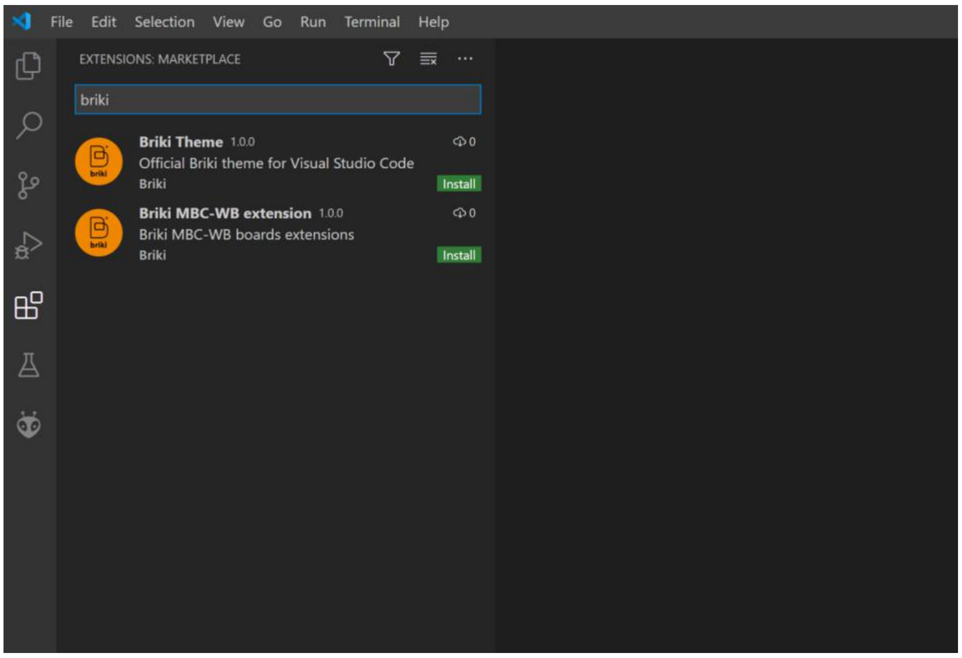
Visual Studio Code Installation

Visual Studio Code can be downloaded from the official website: https://code.visualstudio.com/download while PlatformIO can be downloaded following the official instructions available at the following URL:
https://platformio.org/install/ide?install=vscode

Visual Studio Code Tools installation

As for Arduino IDE, a set of additional tools have been created to give access to features like OTA firmware update, ESP flash partitioning, documentation and especially for Visual Studio Code a custom Theme designed to match the look and feel of Briki boards.

To install the additional tools, open the Visual Studio Marketplace and search for "Briki" in the search bar.

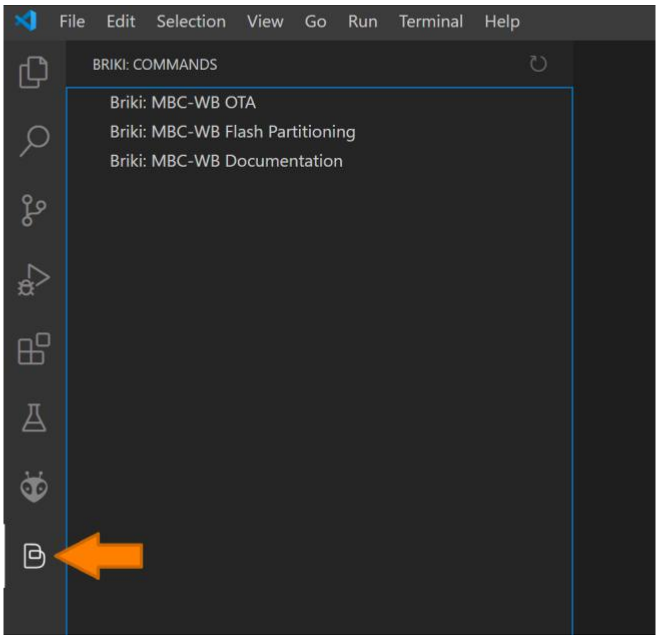


Briki Theme 1.0.0
Official Briki theme for Visual Studio Code
Briki

Briki MBC-WB extension 1.0.0
Briki MBC-WB boards extensions
Briki

Figure 20 - Visual Studio Code Extension

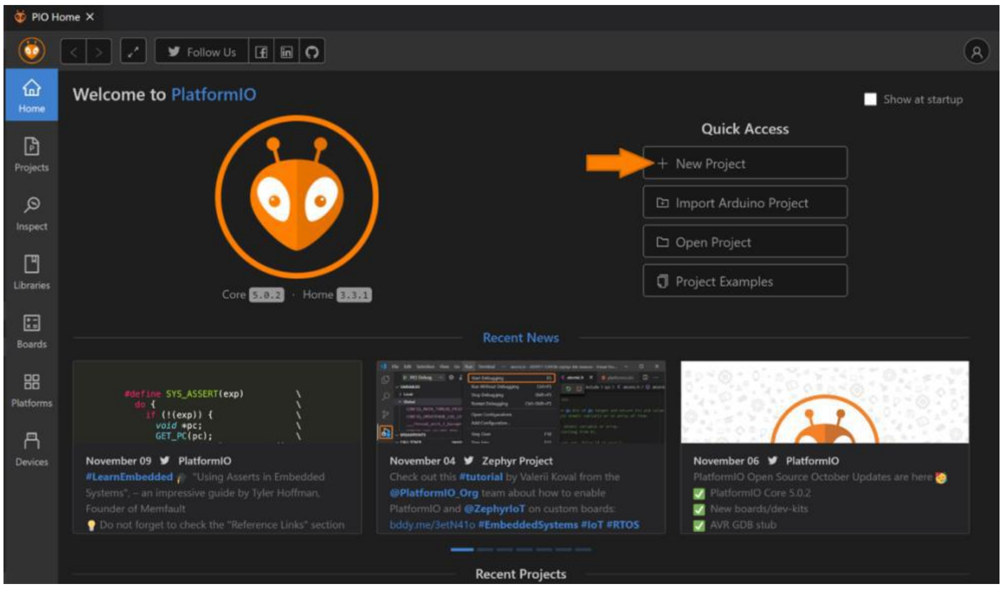
Then selecting "Briki MBC-WB extension" and pressing the install button, the installation will be performed. Once finished, the Briki button on the left menu bar can be pressed to select and use the extension.



Briki: MBC-WB OTA
Briki: MBC-WB Flash Partitioning
Briki: MBC-WB Documentation

Figure 21 - Visual Studio Code Briki extension

Once PlatformIO and the Briki extension for VSCode have been installed, to start using Briki boards just click on the PlatformIO button on the left menu bar and create a new project.



Welcome to PlatformIO

November 09 PlatformIO
#LearnEmbedded "Using Asserts in Embedded Systems", – an impressive guide by Tyler Hoffman, Founder of Memfault
Do not forget to check the "Reference Links" section

November 04 Zephyr Project
Check out this #tutorial by Valerii Koval from the @PlatformIO_Org team about how to enable PlatformIO and @ZephyrIoT on custom boards: bddy.me/3etN41o #EmbeddedSystems #IoT #RTOS

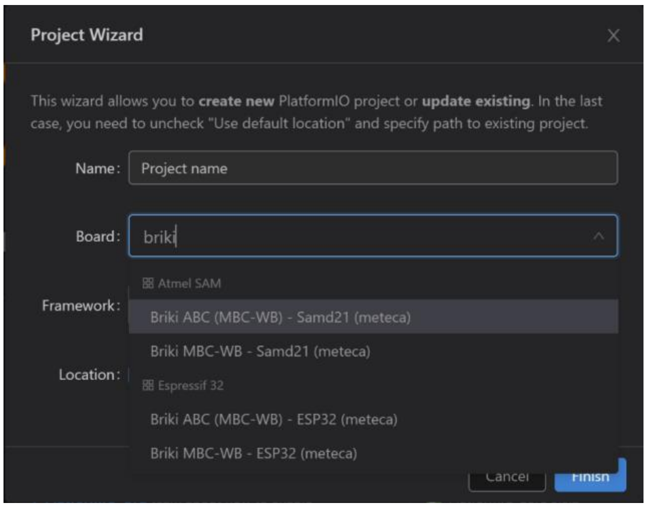
November 06 PlatformIO
PlatformIO Open Source October Updates are here
PlatformIO Core 5.0.2
New boards/dev-kits
AVR GDB stub

Figure 22 - Visual Studio Code new project

In case an existing Arduino project already exists, it can be imported using the dedicated "Import Arduino Project" button.

Once the "New project" button has been clicked, a new form will be shown. To select one of the Briki boards, start typing "Briki" in the Board field and a list of all the available Briki boards will be shown.

Please note that differently from the Arduino environment, PlatformIO binds a project to a specific MCU architecture, so the choice of using the SAMD21 or the ESP32 must be done in this Project Wizard Form and cannot be modified further nor changed just before compiling the project as for Arduino. The choices made here are valid for the whole project life.



Project Wizard

This wizard allows you to **create new** PlatformIO project or **update existing**. In the last case, you need to uncheck "Use default location" and specify path to existing project.

Name: Project name

Board: briki

Framework:

Location:

Atmel SAM
- Briki ABC (MBC-WB) - Samd21 (meteca)
- Briki MBC-WB - Samd21 (meteca)

Espressif 32
- Briki ABC (MBC-WB) - ESP32 (meteca)
- Briki MBC-WB - ESP32 (meteca)

Figure 23 - Visual Studio Code: project wizard

Once the form has been filled in every part, just click on the finish button to start working with the board.

Firmware structure

Both the MCUs of the MBC-WB feature a custom firmware derived by an Arduino implementation:

— ArduinoCore-samd released by Arduino for the SAMD21
— Arduino-esp32 released by Espressif for the ESP32

Starting from these cores, many custom modifications and improvements have been made to allow a better exploitation of the resources available in each microcontroller, to increase their performance, their ease of use and their versatility.

The first main difference from the firmware point of view between the two microcontrollers is in their structure: the ESP32 runs an RTOS implementation, which allows multiple parallel tasks, resource sharing and preemption, while the SAMD21 runs a single thread firmware on bare metal.

Although it is possible to run an RTOS on the SAMD21 or use the ESP32 with bare metal firmware, the current structure has been chosen to maximize the versatility and usability of the two chips in a cooperative way inside the MBC-WB. The ESP32 works at an operating frequency 5 times higher than the SAMD21 (240 MHz against 48 MHz) and this makes it perfect for running multiple parallel tasks alongside executing heavy computational threads. This unfortunately leads to a non-deterministic behaviour of the microcontroller, making it unsuitable for control application. However, the SAMD21, although not so fast, is the better of the two in terms of control, timing and analogue functions, particularly when working in a single-threaded implementation, ensuring precise, reliable and deterministic behaviour.

By exploiting the communication mechanism between the two chips, it is therefore possible to take advantage of the two different implementations on the same module at the same time.

The following pictures summarise the factory firmware structure for SAMD21 and ESP32.

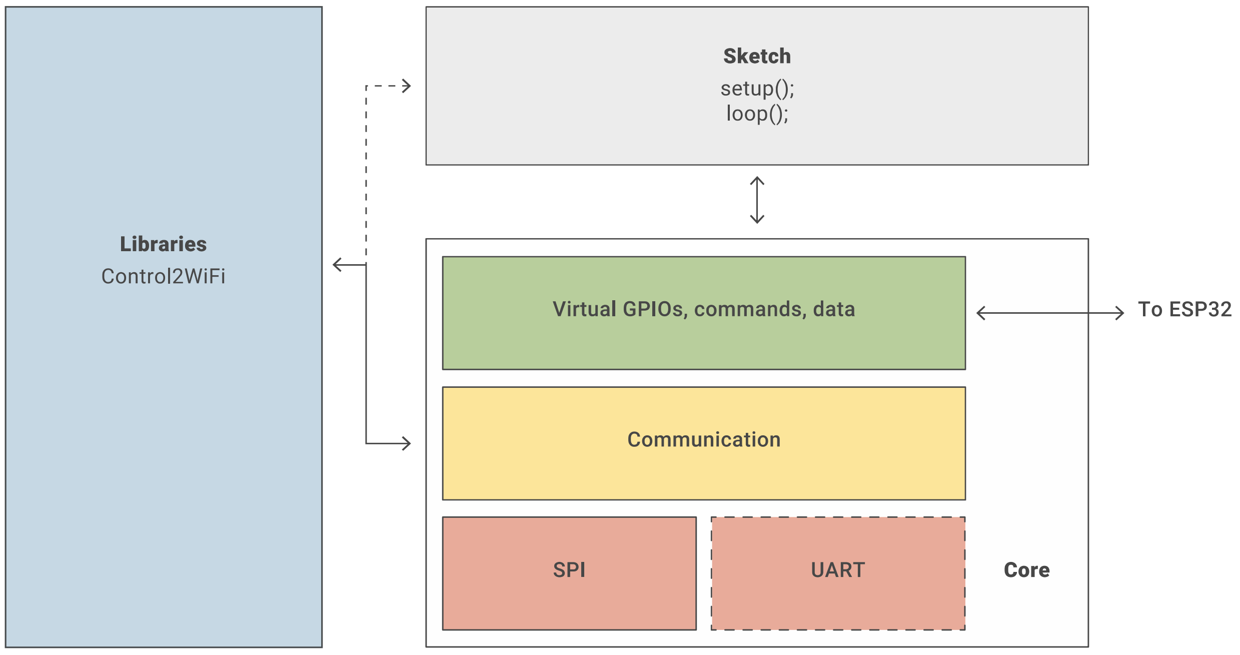


Figure 24 - SAMD21 Firmware structure

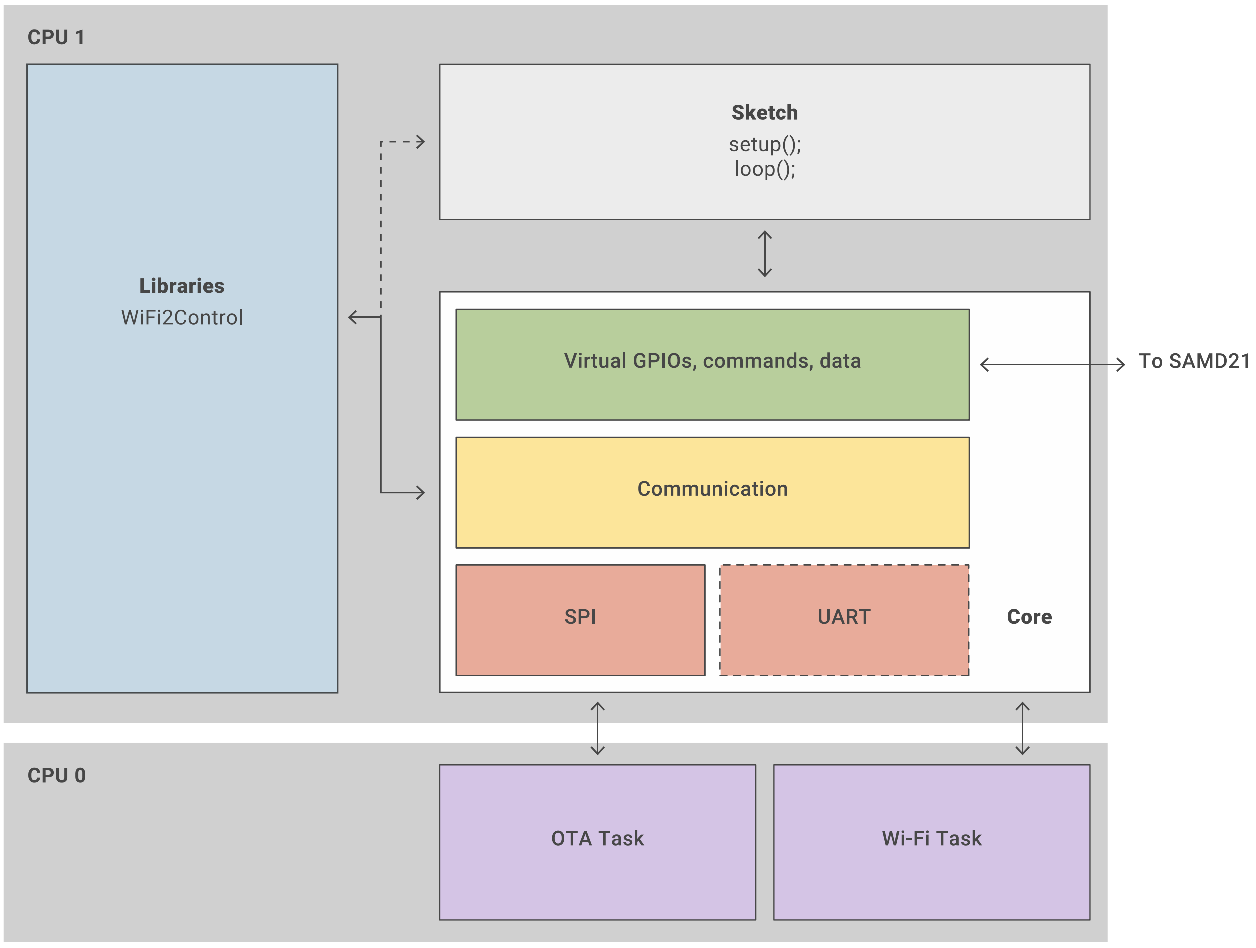


Figure 25 - ESP32 Firmware structure

Inter-chip communication

Aa shown by the pictures above, in both the MCUs alongside the typical Arduino `loop()`, a custom set of functions in charge of the communication and the synchronization process has been inserted between the two chips.

These functions belong to a custom communication library organised in a hierarchical way - high level APIs (e.g. Control2WiFi) exploit the same low level functions used to exchange generic data as described below. The library contains functions to manage data and commands exchange as well as data buffer initialization, and callbacks registration. The low-level communication driver is accessible from the mbc-wb platform by using the communication object.

Since the firmware structure of the two MCUs is different, the library behaves in a slightly different way in the two microcontrollers. Communication events in ESP32 are caught and parsed in a dedicated thread, this means that other complex tasks can be executed in parallel at the same time without the need to "manually" check for communication events in each thread: the scheduler will be responsible for executing the communication thread during its time-slot.

SAMD21, on the other hand, comes with a single-threaded firmware implementation that does not support multithreading as the ESP32 does. This entails the need to check for communication events at the end of each loop cycle. However, this kind of implementation has a side effect: if the `loop()` function requires a considerable amount of time to be executed, due to heavy computational tasks or long delays, some communication events may be lost.

To help the user mitigate this effect, which is closely related to his firmware implementation, a dedicated function has been defined for both the MCUs: the `communication.handleCommEvents()`.
This function is responsible for the communication events handling and when called upon, it will check for incoming communication requests from the companion chip and eventually process them.

In SAMD21, to keep the inter-chip communication bandwidth as high as possible, a call to the function `communication.handleCommEvents()` must be added in those functions that require a long time to come to an end.

The GPIO virtualisation function also relies on this library, this leads to the need to call the `communication.handleCommEvents()` when and where necessary to avoid undesired lags when controlling GPIOs.

As mentioned above, the low-level communication driver is accessible by using the communication object. A detailed description of the communication library can be found here. Among all the functions belonging to this library, three of them have been designed to grant the user a high degree of customisation in the communication behaviour. These functions allow the design of a custom communication protocol as well as the simple sending and receiving of generic data or custom command in both the directions.

Send commands

To send a command the *command ID* must be defined with the function `registerCommand()`. This function takes as input parameters: the new command ID to define, a callback function triggered by the command response issued by the companion chip - in which the response message is parsed - and a buffer in which to store this message.

```
bool communication::registerCommand (uint8_t cmdCode, ext_callback function,
uint8_t* buffer)
```

To ensure a correct communication protocol behaviour, the same command ID must also be defined in the companion chip, using the same function. Otherwise only the former will know the new command ID while the latter will not be able to recognize it as a valid command.

If the new command defined does not require a reply from the other chip, a NULL can be passed as second and third parameter. In this way, without the registration of the callback function, no response will be received.

Once the command is correctly registered through its command ID, it is possible to send it by calling the `sendCommand()` function. This function takes as parameters the command ID, a pointer to the data you want to send and the data length.

```
void communication::sendCommand (uint8_t cmdCode, uint8_t* data, uint32_t len)
```

The following example (written in Arduino language) has been designed to help the user to better understand how to use the above described commands. Let us suppose that SAMD21 has a sensor and a button connected to it and needs to send to the ESP32 their state so that the latter can send a status report via Wi-Fi. In this case 2 commands must be defined: one related to the sensor and the other for the button.

```
#define COMMAND_SENSOR 0x01
#define COMMAND_BUTTON 0x02
#define PIN_BUTTON 11
#define PIN_SENSOR A0
uint8_t commandData[50];
bool measure = true;
void setup() {
  pinMode (PIN_BUTTON, INPUT);
  pinMode (PIN_SENSOR, INPUT);
  // register command for communication
  communication.registerCommand (COMMAND_SENSOR, dataReceivedCallback,
commandData);
  communication.registerCommand (COMMAND_BUTTON, NULL, NULL); // just send data
}
void loop() {
  if (measure == true) {
    uint32_t value = analogRead (PIN_SENSOR);
    if (value < 500) {
      // sensor 1 is below a given treshold. let's send the mesaurement
      uint8_t message[30];
      sprintf ((char*)message, "sensor value: %d", value);
      communication.sendCommand (COMMAND_SENSOR, message, strlen
((char*)message));
    }
  }
  // read button value
  uint8_t buttonVal = digitalRead (PIN_BUTTON);
  if (buttonVal == 0) {
    // button pressed. let's signal the event
    communication.sendCommand (COMMAND_BUTTON, &buttonVal, 1);
  }
}
void dataReceivedCallback (uint32_t commandLen) {
  // check if companion wants us to stop / start measurements
  If (memcmp (commandData, "ON", 2) == 0) {
    // start measurements
    measure = true;
  }
  else if (memcmp (commandData, "OFF", 3) == 0) {
    // stop measurements
    measure = false;
  }
}
```

Send raw data

By defining different command IDs it is possible to have a structured communication between the two MCUs. However, it is also possible to exchange direct messages between the two chips even without defining a structured protocol. To allow this, the communication object has been designed to provide three separate functions: `sendGenericData()`, `availGenericData()` and `getGenericData()`. The `sendGenericData()` function takes a buffer and its length as input parameters and sends them directly to the companion chip.

```
void communication::sendGenericData (void* data, uint32_t len)
```

The receiving MCU can use the `availGenericData()` function to check data availability. If data is available, its size is returned to the caller.

```
int32_t communication::availGenericData (void)
```

Finally, if data are present, it is possible to use *getGenericData()* function to fetch them. The `getGenericData` function takes 2 input arguments: a buffer where data will be placed and an integer representing the number of bytes to read.

```
uint32_t communication::getGenericData (uint8_t* data, uint32_t len)
```

To speed up the communication between the two chips, the data is not transferred when the `getGenericData` function is called, but as soon as the system detects its availability. This means that the data is received at the same time a call to the `availGenericData` function is made. To do this, an internal buffer is dynamically created using the malloc function. Then, calling `getGenericData` simply copies the received data into the buffer passed as a parameter.

malloc is not a safe instruction in memory constrained devices such as microcontrollers. This is mainly due to the risk of running out of memory - if the malloc result is not carefully checked at each call - and to the memory fragmentation that occurs due to repeated memory allocation and deallocation.

To avoid the use of malloc, another function has been added to the communication library: `initExternalBuffer()`. This function registers an external buffer, which will be used to retrieve incoming data, defined within the user firmware instead of using a dynamically created buffer.

```
void communication::initExternalBuffer (void* extBuf)
```

Using the `initExternalBuffer` function, the `getGenericData` can be called without input parameters, since data are already available in the external buffer - a call to `getGenericData` is however needed to clear the available data flag.

Please note that using `initExternalBuffer` is suggested to avoid runtime error.

A simple usage example of these functions is the following:

```
char msg[] = "A generic message";
uint8_t buff[100];
void setup() {
  Serial.begin(115200);
  communication.initExternalBuffer (buff);
}
void loop() {
  communication.sendGenericData (msg, sizeof(msg));
  uint32_t avail = communication.availGenericData ();
  if(avail) {
    communication.getGenericData ();
    Serial.write (buff, avail);
  }
  delay (1000);
}
```

The same example can be uploaded in both the MCUs to see the message being exchanged.

Automatic Baud Rate detection

The Automatic Baud Rate detection (ABR or autobaud) refers to a special feature running on the SAMD21 that is capable of automatically configuring the UART interface shared with the ESP32 according to the UART settings of the ESP32 and, if the Dual-CDC is enabled, links this UART with the corresponding CDC.

The feature is based on the Inter-chip communication feature explained above. During the application firmware boot, a message containing the UART settings is sent from the ESP32 to the SAMD21 through the pre-defined communication interface (SPI or UART, depending on the choice made by the user). Once the message has been parsed, all the interfaces are configured accordingly, granting the Host PC access to the ESP32 via the USB interface offered by the SAMD21. For more details on how the SAMD21 acts as a UART-to-USB bridge, please read the related section.

GPIO virtualization

The MBC-WB has 37 pins that can be used as GPIO. Some of them are connected to the SAMD21, while the others to the ESP32. To simplify the user experience, a special feature has been introduced within the firmware platform provided: the GPIO virtualisation.

Based on the Inter-chip communication, this feature allows each MCU to control the pins that are physically connected to the other chip in an almost transparent way, which does not differ from controlling the pins of the former MCU itself. So, for the user it is totally transparent: moving the pin #1 on the MBC always has the same effect whether the command is issued from SAMD21 or ESP32.

Hence, to use this GPIO virtualisation system, all the common Arduino GPIO APIs can be called as usual (e.g. digitalWrite, analogWrite, etc.) and the two MCUs will be able to act on the right GPIO, regardless of which MCU issued the command.

Please, check how the MBC GPIOs are assigned to the two MCUs, looking to pinout reference section.

Hardware communication interfaces

The MBC-WB features two communication interfaces, one SPI and one UART, plus some control and synchronization signals shared among the two MCUs.

SPI interface

By default, the firmware is configured to exploit the SPI interface, shared among the ESP32 and the SAMD21, for command and data exchange between the two chips.

From the hardware point of view, the SPI interface is configured in a fashion where the SAMD21 acts as the Main, while the ESP32 is the Companion. The choice has been made to achieve the highest possible throughput. In fact, using the SAMD21 as main and the ESP32 as companion leads to a maximum achievable throughput of 540 kBps, during the transmission from SAMD21 to ESP32, while in the opposite direction it is 275 kBps.

To switch from the SPI to UART interface for command and data exchange, a different approach must be followed based on the IDE used:

- Arduino: a specific file (named "communication_channel.h" in cores\samd21\) inside the platform must be modified accordingly
- Visual Studio Code + PlatformIO: the line `build_flags = -DCOMM_CH_UART` must be added to the platformio.ini

To synchronize and allow the communication between the two chips, the signals described in the following table are used. The first column of the table describes the pin from the hardware point of view (i.e. the physical pin retrievable on the pinout of each MCU package), while the second column describes the pin from its firmware reference (i.e. the name the pin gets in the implementation of the firmware platform).

Table 5 - SAMD21-ESP32 shared SPI interface signals

Pin # (HW)	Pin # (FW)	Function	Notes	Direction (default)
GPIO0 (ESP32) **PA15 (SAMD21)**	46 (ESP32) 42 (SAMD21)	ESP32 boot control / ESP32 handshake	boot function is enabled only during ESP32's power up. Direction cannot be swapped	SAMD21 → ESP32 ESP32 → SAMD21
GPIO5 (ESP32) **PA18 (SAMD21)**	41 (ESP32) 45 (SAMD21)	SPI CS	by default, SAMD21 is the main, ESP32 the companion	SAMD21 → ESP32 ESP32 → SAMD21
GPIO18 (ESP32) **PA17 (SAMD21)**	40 (ESP32) 44 (SAMD21)	SPI SCLK		SAMD21 → ESP32
GPIO19 (ESP32) **PA19 (SAMD21)**	38 (ESP32) 46 (SAMD21)	SPI MISO		ESP32 → SAMD21
GPIO21	45 (ESP32)	SAMD21 reset control	direction cannot be swapped	ESP32 → SAMD21
GPIO22 (ESP32) **PA21 (SAMD21)**	42 (ESP32) 47 (SAMD21)	Companion Ready	by default, ESP32 uses this signal to trigger an interrupt in SAMD21	ESP32 → SAMD21
GPIO23 (ESP32) **PA16 (SAMD21)**	39 (ESP32) 43 (SAMD21)	SPI MOSI		SAMD21 → ESP32
PA20 (SAMD21)	41 (SAMD21)	ESP32 reset control	direction cannot be swapped	SAMD21 → ESP32
PA28 (SAMD21)	40 (SAMD21)	ESP32 power control	direction cannot be swapped	SAMD21 → ESP32

UART interface

Instead of using the SPI, it is also possible to use the UART interface. In this case the maximum throughput achievable is lower than the one reached by the SPI interface and its value is approximately 300 kBps.

Once enabled, UART communication interface replaces the default SPI, even maintaining some of the control signals previously used.

Table 6 - SAMD21-ESP32 shared UART interface signals

Pin # (HW)	Pin # (FW)	Function	Notes	Direction (default)
GPIO0 (ESP32) **PA15 (SAMD21)**	46 (ESP32) 42 (SAMD21)	ESP32 boot control / ESP32 handshake	boot function is enabled only during ESP32's power up. Direction cannot be swapped	SAMD21 → ESP32 ESP32 → SAMD21
GPIO1 (ESP32) **PB23 (SAMD21)**	5 (ESP32) 51 (SAMD21)	ESP32 UART TX / SAMD21 UART RX		ESP32 → SAMD21
GPIO3 (ESP32) **PB22 (SAMD21)**	6 (ESP32) 50 (SAMD21)	ESP32 UART RX / SAMD21 UART TX		SAMD21 → ESP32
-				
GPIO21	45 (ESP32)	SAMD21 reset control	direction cannot be swapped	ESP32 → SAMD21
GPIO22 (ESP32) **PA21 (SAMD21)**	42 (ESP32) 47 (SAMD21)	Companion Ready	by default, ESP32 uses this signal to trigger an interrupt in SAMD21	ESP32 → SAMD21
-				
PA20 (SAMD21)	41 (SAMD21)	ESP32 reset control	direction cannot be swapped	SAMD21 → ESP32
PA28 (SAMD21)	40 (SAMD21)	ESP32 power control	direction cannot be swapped	SAMD21 → ESP32

Advanced features

One of the advantages offered by the MBC-WB structure is the coexistence of two MCUs with complementary characteristics. Depending on the user's application it is preferable to choose one or the other microcontroller according to their respective strengths. As mentioned above, the ESP32 features several wireless interfaces, a dual-core architecture, as well as a great computational power with a FreeRTOS firmware implementation. On the other hand, the SAMD21 can achieve higher performances in A/D & D/A conversions, in timed application as well as in PWM generation, also thanks to its bare metal firmware implementation.

Arduino framework comes with a set of predefined APIs to manage analog and PWM pins. These APIs are generic for all the boards and can be quite limiting when a more specific and customized configuration is required. To overcome these limitations, a new set of APIs has been introduced in Briki framework. In this section all these Advanced features are described and explained.

Timers and PWM

SAMD21

Arduino APIs does not provide any function to set up timers at a specified frequency nor callback functions to call when the timeout occurs. This implies that users must write their own driver to interact with timers or use a third-party library. With an external library, the user has no control over which timer is used and whether it will interfere with other functions/peripheral or not during the PWM generation.

Timers in SAMD21 can be of two types: TC (Timer/Counter) and TCC (Timer/Counter for Control Applications). Both can be used for timing purposes or for PWM generation.

To grant a single point of access to all the timers, we defined some new APIs, directly embedded into our Arduino Core. These functions simplify the configuration and starting of a timer with a given frequency and allow the registration of a callback function to call when a timeout occurs. The biggest advantage related to these functions, relies on the *Automatic Resource Management*: if a timer is used by a PWM function, it will not be used to timing purposes.

To "create" a timer object and allocate all the needed resources, the following function can be used:

```
int8_t createTimer(uint32_t n_micros, void (*callback)());
```

where `n_micros` is the time interval expressed in microseconds, and *callback* is the function called whenever a timeout occurs. This callback function takes no argument and returns no value.

`createTimer` function looks for the first available timer, discarding those currently used with `analogWrite` function or affected by a previous call to `createTimer`. Once a free timer is found, it is configured and the `timer ID` (which identifies one of the 6 available timers of SAMD21) is returned as an integer. If there is no timer available, the function returns -1.

The following table highlights the correspondence between timer ID and SAMD21's hardware timer:

Table 7 - Timer HW/FW mapping

Timer ID	Hardware Timer
0	TCC0
1	TCC1
2	TCC2
3	TC3
4	TC4
5	TC5

A timer created with this function is not yet started. To start the timer, `startTimer` function must be called.

```
void startTimer(uint8_t timerId);
```

This function starts a previously configured timer. It takes only one parameter, `timerID` which is the value returned by the function `createTimer` corresponding to the instance of a given hardware timer.

A running timer can be stopped at any time by calling the `stopTimer` function. Even in this case the only parameter required is the timer ID. A timer stopped with this function is only paused. It can be resumed at any time by calling the `startTimer` function again.

```
void stopTimer(uint8_t timerId);
```

Finally, to stop and de-allocate the timer resources, `destroyTimer` can be called. Again, the only required parameter is the timer ID. Once the resources are freed, `timerId` stops being referred to any hardware timer instance.

```
void destroyTimer(uint8_t timerId);
```

To keep track of the remaining available timers, the function to call is `availableTimers`:

```
uint8_t availableTimers();
```

that returns the number of free timers that can still be used for timing purposes.

Thanks to the functions described above, the user can add timed events or functions to the application at design time without having to worry about resource sharing with PWM. Despite this, it is always possible that external libraries can interact with hardware timers or a specific timer must be allocated to other activities and must not be used by these functions. For this purpose, we have defined two utility functions to better manage the use of timers:

```
void allocateTimer(uint8_t timerId);
```

that allows the user to specify which timer must not be considered as available by the above-mentioned functions. The parameter is the timer associated ID: `timerId`. This ensures that no resource conflict will happen if some timer is used outside the standard Arduino APIs.

The function `isTimerUsed` can be called to check if the given timer instance, identified by the `timerId` parameter passed as argument, is currently in use or it is available. The function returns `true` if the timer is already in use, `false` otherwise.

```
bool isTimerUsed(uint8_t timerId);
```

Please, note that the PWM configuration has a higher priority than the simple timing configuration. This means that a timer already configured, up and running as a timing timer will have its configuration overwritten by a call to configure it as PWM timer, resulting in a change in its behviour. On the contrary, a timer configured as PWM cannot be reconfigured as a timing timer without a prior dispose of its allocated resources. `allocateTimer` function can be used to start the timer before the PWM pulse, by passing the timer instance your PWM is using, to ensure PWM will not interrupt it.

In the Arduino framework, the function assigned to PWM generation is `analogWrite`.

The `analogWrite` function takes two input parameters: the `pin` and the `duty-cycle` the PWM signal will have. The duty-cycle depends on the configured resolution. By default, it is configured to 8-bit; this means that the duty-cycle value can be expressed as an unsigned integer with a value ranging from 0 to 255. Resolution can also be changed by using `analogWriteResolution` function.

The following example shows how the `analogWriteResolution` can be used to change the PWM resolution from 8 bit to 10 bit and then how to call `analogWrite` function to generate a pulse with 50% of duty-cycle on pin #13:

```
analogWriteResolution(10);
analogWrite(13, 512);
```

In SAMD21 only those pins that are internally connected to a hardware timer can generate a PWM signal. This means that once a timer has been configured to generate a PWM signal, its output can only be routed to a set of predefined pins. A timer already configured to generate PWM pulses cannot be used for timing/control purposes. Please read the section above for information about resource sharing and how to manage them.

Using the Arduino framework, the MBC-WB can generate PWM pulses on the following SAMD21 pins:

#11, #12, #13, #14, #15, #18, #20, #21, #22, #23, #24 and #25.

To find the location of these pins, please refer to the pinout section of this documentation. For more information about how timers are assigned to pins, please refer to SAMD21 datasheet.

The table below reports the timer and the channel used by each pin to generate PWM pulses by using the function `analogWrite`:

Table 8 - AnalogWrite pin and timer/channel mapping

MBC-WB pin	Hardware Timer / Channel
11	`TC4 CH1`
12	`TC3 CH0`
13	`TCC1 CH1`
14	`TCC0 CH1`
15	`TCC0 CH0`
19	`TC4 CH0`
20	`TC4 CH1`
21	`TC5 CH1`
22	`TC5 CH0`
23	`TCC2 CH0`
24	`TCC2 CH1`
25	`TCC1 CH1`

The `analogWrite` function provides a simple way to generate PWM signals. However, by default in the classical Arduino implementation, it generates pulses with a fixed frequency of 732 Hz. The only parameter that can be changed is the duty-cycle. Usually to change the frequency the user must interact directly on SAMD21's registers.

To overcome this limit, a new function has been defined: `analogWriteFreq`. It works exactly as the `analogWrite` function, but beside the `pin` number and `duty-cycle`, it also takes a third parameter that specify the `frequency` the pulse should be generated at. This parameter is an unsigned integer used to specify the pulsing frequency from 1Hz up to the maximum allowed frequency (see PWM benchmarks for details).

The frequency set with this function is specific for each pin; this means that it is possible to set different frequencies for each pin. Once the frequency has been specified, the duty-cycle can be changed by calling the standard `analogWrite` function as always. This will not overwrite the previously set frequency. If, instead, the frequency must be changed, at any time it is sufficient to call again the `analogWriteFreq` with the new desired frequency value.

The following code example show how the `analogWriteFreq` function can be used with the classical “Arduino Fade” example used to fade in and out an LED’s light:

```
int led = 13;
int brightness = 0;
int fadeAmount = 5;
void setup() {
    pinMode(led, OUTPUT);
    analogWriteFreq(led, brightness, 1000); // initialize PWM on pin 13,
    initial duty-cycle 0 and frequency 1000Hz
}
void loop() {
    // set the brightness of pin 13:
    analogWrite(led, brightness);

    // change the brightness for next time through the loop:
    brightness = brightness + fadeAmount;

    // reverse the direction of the fading at the ends of the fade:
    if (brightness <= 0 || brightness >= 255) {
    fadeAmount = -fadeAmount;
    }

    // wait for 30 milliseconds to see the dimming effect
    delay(30);
}
```

Pins sharing the same hardware timer can have different duty cycle, but not different frequencies: changing the frequency for a pin means changing the frequency for all the pins connected to the same timer.

Benchmarks

The precision of the frequency used to generate PWM pulses changes accordingly to the type of timer used. In general, PWM generated with a TCC timer are more precise than those generated with a TC timer.

TCC timers can generate PWM signals at any frequency in the range **1 Hz to 16 MHz** and with a duty-cycle from 0 to 100%. They can also generate pulses under 1 Hz of frequency. Please, look at SAMD21 datasheet for more information about how to reach such frequencies.

TC timer can generate pulses with fixed frequency steps until 183 Hz. In case none of the available fixed frequencies is directly chosen by the user, the timer is automatically set with the nearest possible fixed frequency value. Beyond 183 Hz it can generate a PWM signal at any frequency in the range between **183 Hz and 10 MHz** and with a duty-cycle from 0 to 100%.

The currently supported TC frequencies are:

- 2,86 Hz
- 11,44 Hz
- 45,7 Hz
- 91,5 Hz
- any frequency between 183 Hz and 10 MHz

ESP32

Beside the above-mentioned `analogWrite` and `analogWriteFreq` functions, inherited by the Arduino Core implementation, the ESP32 also provides its own IDF functions to work with timers. Please, for more informations about the available functions read the related section of the official Espressif documentation.

In addition to this, timers used for timing purposes in ESP32 do not interfere with PWM generation, hence no special care must be taken to work with both the functionalities.

Despite its versatility, one characteristic must be considered when working with timers in ESP32: it runs a FreeRTOS kernel that allows to generate concurrent threads. This implies that the user code is constantly scheduled among all the concurrent threads. Scheduling, despite taking into account the level of priority that each task may have, can interfere with the application execution time leading to a loss of determinism.

To have an overview of timers in ESP32, please check the available examples in *File → Examples → Examples for Briki MBC-WB → ESP32 → Timer* or, by using Ticker library, in *File → Examples → Examples for Briki MBC-WB → Ticker*.

Even the ESP32 can obviously generate PWM pulses.

Using the Arduino framework, the MBC-WB can generate PWM pulses on the following ESP32 pins:

#0, #1, #2, #3, #4, #5, #6, #9, #10, #33, #34, #35, #36, #37.

To find the location of these pins, please refer to the pinout section of this documentation.

The ESP32 has a completely different way of generating PWM pulses than SAMD21, although the same `analogWrite` and `analogWriteFreq` functions available for the SAMD21 are also available for the ESP32 and have the same behavior and configuration process.

The main difference with SAMD21 is the maximum frequency the PWM can achieve. For ESP32 the maximum allowed frequency is **80 MHz / $2^{precision}$**, where precision can be any number in the range from 1 to 15. This means that with an 8-bit resolution, the maximum obtainable frequency is 312.5 kHz.

As can be noticed, the SAMD21 can reach higher frequencies than ESP32, despite a lower clock rate. Therefore, for high precision control application, the SAMD21 is the suggested microcontroller to work with.

Analog acquisition

The MBC-WB exposes 18 pins with analog capabilities: 14 of them (from A0 to A13) belong to SAMD21, whilst the remaining 4 (from A10 to A13) belongs to ESP32.

Arduino classical APIs only allow to call `analogRead` to start an A/D conversion or to select the ADC resolution by calling `analogReadResolution` function and passing to it the desired resolution (8, 10 or 12 bits). However, SAMD21 has a richer set of functionalities to perform A/D conversions. The following functions provide all the SAMD21's analog features to give the user a full control over the ADC peripheral.

All the following functions derive from the SAMD21's hardware features, thus they are not available for the ESP32.

The SAMD21 features a flexible input selection mux capable of both differential and single-ended measurements plus an optional gain stage suitable to increase the dynamic range. This stage can be controlled with `analogReadGain` function to set a gain ranging from 0.5x up to 16x. The default value is 0.5x.

```
void analogReadGain(eAnalogGain g);
```

The input parameter *g* specifies the dynamic range. Allowed values are:

Table 9 - analogReadGain

`AG_0_5X`	gain set to 0.5x
`AG_1X`	gain set to 1x
`AG_2X`	gain set to 2x
`AG_4X`	gain set to 4x
`AG_8X`	gain set to 8x
`AG_16X`	gain set to 16x
`AG_DEFAULT`	(same as AG_0_5X)

Inherent gain and offset errors affect the absolute accuracy of the ADC.

The offset error is defined as the deviation of the actual ADC transfer function from an ideal straight line at zero input voltage.

The gain error is defined as the deviation of the last output step's midpoint from the ideal straight line, after compensating for offset error.

The SAMD21 features the in-hardware offset ang gain compensation, without the need of specific code functions. To correct these two errors, the Digital Correction Logic must be enabled, the Offset value and Gain value must be stored respectively into the Offset Correction register and the Gain Correction register.

Once all is setup properly, the ADC hardware will automatically compensate these errors giving a result based on the formula: **Result = (Conversion value – Offset value) * Gain value**

To automatically correct these two errors at once the following function has been added to the Arduino Core; it can be called inside the `setup()` function:

```
void analogReadCorrection(int offset, uint16_t gain);
```

The correction will introduce a latency of 13 ADC clock cycles. In free running mode this latency is introduced on the first conversion only (its duration is always less than the propagation delay). In single conversion mode this latency is introduced for each conversion.

The `analogCalibrate` function fetches the `BIAS` and `LINEARITY` calibration values acquired during the production test and stored in the NVM Software Calibration Area and loads them into the ADC Calibration register to achieve the specified accuracy.

```
void analogCalibrate(void);
```

The accuracy of the ADC determines how close the actual digital output is to the theoretically expected digital output for a given analog input. In other words, the accuracy of the converter determines how many bits in the digital output code represent useful information about the input signal. The accuracy of the ADC is a function of its internal circuitry and noise from external sources connected to the ADC input.

The ADC is clocked by GCLK_ADC. This clock can be prescaled to enable conversion at lower clock rates with the `analogPrescaler` function:

```
void analogPrescaler(uint8_t val);
```

The input parameter represents the value the clock can be prescaled by. Allowed values are:

Table 10 - analogPrescaler

`ADC_CTRLB_PRESCALER_DIV4_Val`	Peripheral clock divided by 4
`ADC_CTRLB_PRESCALER_DIV8_Val`	Peripheral clock divided by 8
`ADC_CTRLB_PRESCALER_DIV16_Val`	Peripheral clock divided by 16
`ADC_CTRLB_PRESCALER_DIV32_Val`	Peripheral clock divided by 32
`ADC_CTRLB_PRESCALER_DIV64_Val`	Peripheral clock divided by 64
`ADC_CTRLB_PRESCALER_DIV128_Val`	Peripheral clock divided by 128
`ADC_CTRLB_PRESCALER_DIV256_Val`	Peripheral clock divided by 256
`ADC_CTRLB_PRESCALER_DIV512_Val`	Peripheral clock divided by 512

The `analogHWAveraging` function enables hardware signal averaging. Signal averaging is a signal processing technique applied in the time domain, intended to increase the strength of a signal relative to noise at the cost of a reduced sampling rate. By averaging a set of replicate measurements, the Signal-to-Noise Ratio (SNR) will be increased, ideally in proportion to the number of measurements; in particular the random noise components are reduced by a factor equal to the square root of the number of averaged cycles.

Many and different are the sources of noise that affect the SNR and degrade the conversion quality: the source impedance, the thermal noise generated by resistive elements, the other signals present on the PCB itself (in particular the digital ones) and many more. Signal averaging typically relies heavily on the assumption that the noise component of a signal is random, having zero mean, and being unrelated to the signal. Over time, the value of Gaussian distribution noise averages to zero, so by taking an average of several signal cycles, the random noise error can be removed.

Using an hardware approach to the signal averaging, instead of a software one, gives some advantages: it is faster than a dedicated code function and no code is needed to process the data acquired, it is simply required to properly setup the hardware function and get back filtered data without any intervention from the CPU, leaving more room for the execution of other function in the meanwhile.

```
void analogHWAveraging(eAnalogAccumuDepth acc, eAnalogDivFact div);
```

The function takes as input parameter the number of accumulation samples and the division factor. The first parameter, `eAnalogAccumuDepth`, takes care of all the accumulation values. Allowed values for this parameter are:

Table 11 - eAnalogAccumuDepth

`ACCUM_1_SAMPLE`	1 sample accum.	intermediate result precision = 12 bits	num. of automatic right shift = 0
`ACCUM_2_SAMPLE`	2 sample accum.	intermediate result precision = 13 bits	num. of automatic right shift = 0
`ACCUM_4_SAMPLE`	4 sample accum.	intermediate result precision = 14 bits	num. of automatic right shift = 0
`ACCUM_8_SAMPLE`	8 sample accum.	intermediate result precision = 15 bits	num. of automatic right shift = 0
`ACCUM_16_SAMPLE`	16 sample accum.	intermediate result precision = 16 bits	num. of automatic right shift = 0
`ACCUM_32_SAMPLE`	32 sample accum.	intermediate result precision = 17 bits	num. of automatic right shift = 1
`ACCUM_64_SAMPLE`	64 sample accum.	intermediate result precision = 18 bits	num. of automatic right shift = 2
`ACCUM_128_SAMPLE`	128 sample accum.	intermediate result precision = 19 bits	num. of automatic right shift = 3
`ACCUM_256_SAMPLE`	256 sample accum.	intermediate result precision = 20 bits	num. of automatic right shift = 4
`ACCUM_512_SAMPLE`	512 sample accum.	intermediate result precision = 21 bits	num. of automatic right shift = 5
`ACCUM_1024_SAMPLE`	1024 sample accum.	intermediate result precision = 22 bits	num. of automatic right shift = 6

The second parameter, `eAnalogDivFact`, takes care of all the right shift (division) factors. Allowed values for this parameter are:

Table 12 - eAnalogDivFact

`DIV_FACT_1`	0 right shift, equal to divide the result for 1
`DIV_FACT_2`	1 right shift, equal to divide the result for 2
`DIV_FACT_4`	2 right shift, equal to divide the result for 4
`DIV_FACT_8`	3 right shift, equal to divide the result for 8
`DIV_FACT_16`	4 right shift, equal to divide the result for 16

By combining these two parameters in the function `analogHWAveraging`, different output resolutions can be achieved: for example combining `ACCUM_8_SAMPLE` and `DIV_FACT_8` you will get an intermediate resolution of 15 bits due to accumulation and a final output resolution of 12 bits due to a right shift of 3 positions. Using the same `ACCUM_8_SAMPLE` with `DIV_FACT_2` this time, leads to an output resolution of 14 bits.

Be careful that `analogHWAveraging` automatically configure the ADC sampling rate to a preset value depending on the averaging value selected. If a different clock prescale factor must be chosen, please, call the `analogPrescaler` function right after this one.

To accurately convert an analog signal into its digital representation, the ADC must have some specific characteristics:

— a sampling rate high enough to provide enough samples to adequately represent the input signal. Based on the Nyquist-Shannon Sampling Theorem, the minimum sampling rate must be at least twice the frequency of the highest frequency component in the target signal (Nyquist Frequency).
— an adequate sampling resolution capable of faithfully recreate the amplitude variation of the input signal. If the signal is sampled at or above the Nyquist Frequency, post-processing techniques can be used to interpolate intermediate values and reconstruct the original input signal to within desired tolerances.

Since many MCUs usually do not have ADCs with a high number of bit (typical is 10-12 bits), there are alternative methods that can enhance digital sampling results with relatively simple post-processing. The Oversampling and Decimation is one of them.

This technique involves oversampling of the input signal so that a number of samples can be used to compute a virtual result with greater accuracy than a single real sample can provide. Oversampling simply refers to sampling the signal at a rate much higher than the Nyquist Frequency. The increased sampling rate does not directly improve ADC resolution, but by providing more samples, the input signal can be tracked more accurately by better utilizing the existing ADC dynamic range.

It should be clear that oversampling by itself improves the digital representation of the signal only down to the physical dynamic range limit (minimum step size) of the ADC. Increasing the sampling rate further without additional post-processing simply results in multiple samples of the same value during each step in the waveform, yielding no real improvement in the basic digital conversion. Once the oversampling is done, we can simply sum the samples during a given sampling interval to derive a value that represents the value of the input during that sampling interval.

For example, when we sum sixteen 12-bit values, the result is a 16-bit (decimated) result. In addition to yielding a more accurate approximation of the signal value during a given sampling interval, decimation also helps to improve the signal-to-noise ratio (SNR) of the input signal. By spreading the effects of random noise over multiple samples and computing a sum, decimation allows the noise to be partially cancelled from the result.

In SAMD21 the oversampling and decimate is directly done in hardware, without the need of any code function except the following one needed to properly configure the ADC registers. The ADC resolution can be increased from 12 bits up to 16 bits, for the cost of reduced effective sampling rate. To increase the resolution by n bits, 4^n samples must be accumulated. The result must then be right shifted by n bits. This right-shift is a combination of the automatic right-shift done by hardware itself and the value passed as input parameter to the function:

```
void analogOversamplingAndDecimate(eOversamplingVal ovs);
```

The input parameter, `eOversamplingVal`, sets the resulting bit depth for the AD conversion (from 13 up to 16 bits). Allowed values for this parameter are:

Table 13 - eOversamplingVal

`OVERSAMPLED_RES_13BIT`	→ number of samples to average = 4	number of automatic right shift = 0
`OVERSAMPLED_RES_14BIT`	→ number of samples to average = 16	number of automatic right shift = 0
`OVERSAMPLED_RES_15BIT`	→ number of samples to average = 64	number of automatic right shift = 2
`OVERSAMPLED_RES_16BIT`	→ number of samples to average = 256	number of automatic right shift = 4

Final conversion of ADC is now at 13, 14, 15 or 16 bits wide.

Power modes

The MBC-WB features a flexible power management that allows the user to individually control the power source of each MCU thanks to the exposed pins on the module pinout. The picture below shows the location of the power pins on the pinout: pin #36 and #37 are the SAMD21's power pins, while pin #40, #41 and #42 are the ESP32's power pins. The pins colored in grey are ground connections.

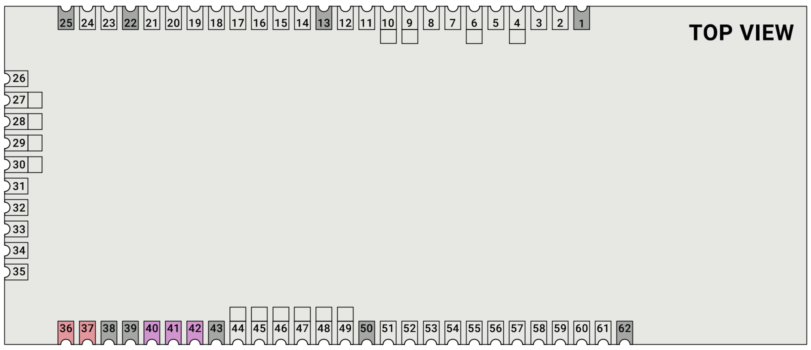


Figure 26 - MBC power pins

In addition to this feature, the SAMD21 of the MBC-WB can internally control the power source of the ESP32. Since the latter is the more consuming of the two chips (this is the reason why it has 3 power pins instead of 2), to grant the highest level of power saving, the SAMD21 has been chosen as the hardware supervisor and controller of the power. The SAMD21 can switch on and off the power of the ESP32 by controlling a specific power gating circuit. This means that the user can completely cut-off the power to the ESP32 simply by toggling one GPIO from the SAMD21. To better understand how the power gating circuit looks like and works, please check the related section in the schematic.

When the ESP32 is powered down, the OTA functionality and consequently the updates over-the-air, will no longer be available.

Beside the ability to turn the ESP32 on and off, the SAMD21 can also reset it, simply by controlling a single GPIO. Even the ESP32 can reset the SAMD21 in the same way. This cross-control functionality is especially important to allow the firmware update in both directions (when the update is done via USB, the SAMD21 must reset the ESP32, when the update is done via OTA, the ESP32 must reset SAMD21). These two signals can be externally controlled via the exposed pins.

To correctly control the reset signals using the external pinout, please read carefully the instructions reported in the pinout section.

To allow an internal fast and simple control over the reset of each chip, a custom function has been inserted in the firmware platform: `switchCompanion()`. By using the define `COMPANION_OFF` or `COMPANION_ON`, the companion MCU can be put in reset or in run mode.

When the SAMD21 is powered down, no USB interface can be shown. Consequently, neither the Serial communication nor the firmware update (via USB) will be available.

Setting an MCU in reset state, however, may not lead to sufficient power reduction. To further reduce the power consumption of the MBC-WB, making it energy-efficient when battery powered, several strategies has been adopted for both the MCUs.

Low Power modes: SAMD21

Low power states in SAMD21 are managed by the ArduinoLowPower library. Based on the official Arduino library, this custom version has been extended to ensure the maximum flexibility and power reduction.

To use the APIs contained in the library it is sufficient to include its header file in the code with the following instruction: #include "ArduinoLowPower.h".

This library offers three different levels of power saving:

- *idle* – that allows 3 level of power optimization with the fastest wake-up time
- *sleep* – that ensures a good power optimization but with a slower wakeup time
- *deep sleep* – that ensures the best power optimization but with a slowest wakeup time

Table 14 - SAMD power scheme

Sleep mode	CPU clock	AHB clock	APB clock	Oscillators	Main clock	Regulator mode	RAM mode
`IDLE_0`	Stop	Run	Run	Run/Run if req.[1] Run/Run if req.	Run	Normal	Normal
`IDLE_1`	Stop	Stop	Run	Run/Run if req.[1] Run/Run if req.	Run	Normal	Normal
`IDLE_2`	Stop	Stop	Stop	Run/Run if req.[1] Run/Run if req.	Run	Normal	Normal
`STANDBY`	Stop	Stop	Stop	Stop/Stop.[1] Run/Run if req.	Stop	Low Power	Low Power

[1] the first row is with RUNSTDBY = 0 while the second is with RUNSTDB = 1. The left status is obtained with ONDEMAN = 0 while the right one with ONDEMAND = 1.

Before entering in one of these modes it is recommended to stop all unused peripherals to get better performances.

Sleep and deep sleep modes require that both Systick timer and USB peripheral are disabled. This means that by using one of these modes both are automatically disabled preventing not only communication but also the firmware update, at least as long as the sleep condition persists. As soon as the SAMD21 exits from sleep mode, the Systick timer and the USB peripheral are automatically enabled again, restoring full functionalities.

The SAMD21 can stay in low power mode forever, if desired, or unless a specific event previously configured to wake the board up occurs. The wake-up event can be bounded to a GPIO level change event or to a timeout event of the RTC timer.

GPIO wake-up event can be configured with the following function:

```
LowPower.attachInterruptWakeup(uint32_t pin, voidFuncPtr callback, uint32_t mode);
```

Where: `pin` represents the GPIO number used to wake up the board, `callback` is an optional function that can be called when the board wakes up (this parameter can also be `NULL`) and `mode` represent the type of event on the specified pin (it can be `FALLING`, `RISING` or `CHANGE`).

If a timed event is instead chosen as a trigger to wake-up the board, a time interval expressed in milliseconds can be passed as parameter to `idle()`, `sleep()` or `deepSleep()` functions. The RTC timer will be automatically configured to wake-up the board after the desired time elapses.

For example, to enter in sleep mode and wake-up the board after 10 seconds the following function can be called:

```
LowPower.sleep(10000);
```

Even with a timed event a callback function can be called once the MCU wake-up. To set it up just call the previously introduced function `LowPower.attachInterruptWakeup()` by passing `RTC_ALARM_WAKEUP` as pin instead of a GPIO pin number.

In case a battery powered application requires the SAMD21 to repeatedly enter and exit from an idle or sleep condition (for example to wake up, read a sensor value and then go back to sleep), have the Systick timer and the USB enabled at any wake-up event, it could lead to an excessive and unnecessary amount of energy drained from the battery.

In this case `idle()`, `sleep()` and `deepSleep()` functions can be called with an additional parameter to specify the SAMD21 that after the execution of the wake-up associated callback, it must not perform a complete wake-up, instead it can go back to low power mode. To enable this behavior a wake-up callback function must be configured with the `attachInterruptWakeup()`, then call `sleep` or `idle` function with an additional *true* parameter:

```
LowPower.sleep(true);
```

Or, if a timed wake up is required:

```
LowPower.sleep(10000, true);
```

This ensures the board will immediately enter in the selected low power mode after your callback function is executed.

In cases where an idle or a sleep/deep sleep condition is not required, but a reduction in power consumption is still desired, for example to keep some functions always active while preserving energy, a solution is offered by the main clock frequency reduction. Reducing the main clock frequency can significatively reduce the power consumption. However, once the MCU frequency is modified, all the active peripherals need to be reinitialized to adapt at the new frequency.

When the SAMD21 is running at a frequency lower than 48MHz, no USB interface can be shown. Consequently, neither the Serial communication nor the firmware update (via USB) will be available until the frequency is restored to its default value (48MHz).

The clock frequency can be changed using the following function:

```
LowPower.changeClockFrequency(frequencies frequency);
```

Where the *frequency* parameter can be one of the following values:

- `FREQ_48MHz`
- `FREQ_32MHz`
- `FREQ_16MHz`
- `FREQ_8MHz`
- `FREQ_4MHz`
- `FREQ_1MHz`

Benchmarks

Power consumption is strictly related to the application and the peripherals used. Different combinations may lead to different power consumption levels. The table 12 reports some reference values of power consumption measured on SAMD21 operating in the different power modes, with the ESP32 powered off.

These values are provided for reference and may vary from what the user gets in his final application. These measurements were acquired on a DBC board with an MBC-WB-BPS at an ambient temperature of 25°C running a While(1) algorithm, with a single CDC interface enabled.

Table 15 - SAMD21 down-clocking power consumption

Frequency [MHz]	Current consumption [mA]
48	9.13
32	5.18
16	3.92
8	2.60
4	1.96
1	1.48

Low Power modes: ESP32

The ESP32 is a powerful device that has several wireless interfaces on the same chip area (Wi-Fi/BT/BLE). This can lead to a relatively power-hungry behavior depending on which peripherals are enabled and in which state it is running. If the final application requires a wall power supply, this power consumption may not be as important, but if this device is battery powered, it could lead to a high current draw, capable of consuming battery power too quickly.

To reduce the ESP32's power consumption, it can be turned off by the SAMD21, or it can activate its own low power modes.

There is no absolute best solution when it comes to power reduction. The right solution depends strictly on the requirements of the final application in terms of energy consumption and needed features. A general suggestion is to turn on the wireless interfaces only when needed and then turn them off again as soon as possible, if possible. This can guarantee good power savings.

To reduce power consumption, ESP32 can be turned off by SAMD21, or it can activate one of its own low power modes. When ESP32 enters sleep mode, its state is maintained in RAM while any unneeded digital peripherals are powered down.

The following picture shows function block diagram of ESP32 chip.

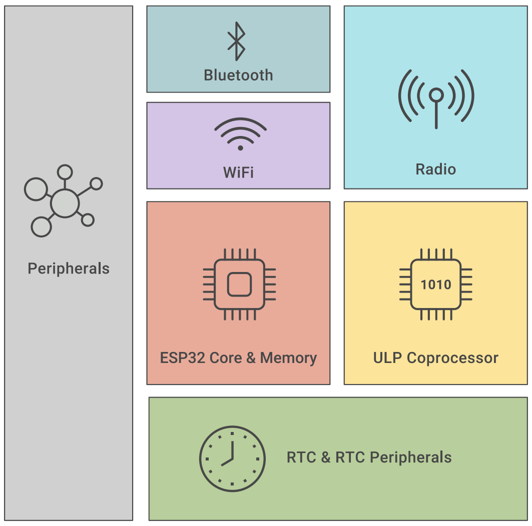


Figure 27 - ESP32 hardware block diagram

The ESP32's advanced power management offers 5 configurable power modes:

- Active Mode
- Modem Sleep Mode
- Light Sleep Mode
- Deep Sleep Mode

Table 16 - ESP32 power scheme

Sleep mode	CPUs	Wi-Fi	Bluetooth	Radio	ULP Co-CPU	Peripherals	RTC	Current consumption [mA]
Active	Run	Run	Run	Run	Run	Run	Run	~ 150 - 260
Modem Sleep[1]	Run	Stop/Run	Stop/Run	Stop/Run	Run	Stop	Run	~ 3 - 20
Light Sleep[2]	Clock gated[2]	Stop/Run	Stop/Run	Stop/Run	Run	Stop	Run	~ 1
Deep Sleep[3]	Stop	Stop	Stop	Stop	Run	Stop	Run	~ 0.01 – 0.15

[1] To keep Wi-Fi / Bluetooth connections alive, the CPU, Wi-Fi, Bluetooth, and radio are woken up at predefined intervals. It is known as ***Association sleep pattern***. During this sleep pattern, the power mode switches between the Active mode and Modem Sleep mode. ESP32 can enter Modem Sleep mode only when it connects to the router in station mode. It stays connected to the router through the DTIM beacon mechanism. To save power, ESP32 disables the Wi-Fi module between two DTIM Beacon intervals and wakes up automatically before the next Beacon arrival. The sleep time is decided by the DTIM Beacon interval time of the router which is usually 100ms to 1000ms.

[2] During light sleep mode, digital peripherals, most of the RAM and CPU are clock-gated. Clock gating reduces the dynamic power consumption by disabling parts of the circuitry switching off the clock pulses directed to them. Before entering light sleep mode, ESP32 preserves its internal state and resumes operation upon exit from the sleep (***Full RAM Retention***).

[3] During deep sleep mode, the main CPU is powered down, while the ULP co-processor does sensor measurements and wakes up the main system, based on the measured data from sensors. This sleep pattern is known as ULP ***sensor-monitored pattern***. Along with the CPU, the main memory of the chip is also disabled. So, everything stored in that memory is wiped out and cannot be accessed. In this mode, power is shut off to the entire chip except RTC module. So, any data that is not in the RTC recovery memory is lost, and the chip will thus restart with a reset. This means program execution starts from the beginning once again.

For a detailed description of all the low power modes of the ESP32 and how to enable them, refer to the official Espressif documentation. Since this section of the official documentation is well described, as well as the functions that allow Sleep Modes use, no dedicated library has been developed, preferring to leave the user the freedom to use the official one.

Storage

The MBC-WB has an oversized external flash memory compared to the typical needs of the ESP32. Therefore, this can be partitioned into different memory areas depending on the features the user wants to enable. By default, among the partitions containing the bootloader, the running firmware etc., one is kept completely free and available for the user to store everything they need: from different firmware versions, to lookup tables for algorithms or even data to be processed or transmitted later in the event of loss of Internet connectivity.

This last partition can be formatted to host a FAT file system or a SPIFFS file system. To better understand how the flash has been partitioned by default, please refers to this section.

SPI Flash File System (SPIFFS) is a file system intended for flash devices on embedded targets. It can work with files using C standard library and POSIX APIs. However, it does not support directories. It is possible to work with SPIFFS file system in Arduino environment by using the SPIFFS library provided by Espressif® (some examples are already available in the platform under *File → Examples → Examples for Briki MBC-WB → SPIFFS*).

FAT is one of the most known file systems. It can work with files, it supports directories and, differently from SPIFFS, it can work with encrypted data. In the Arduino environment it is possible using FFat library to access this filesystem (some examples are available in the platform under *File → Examples → Examples for Briki MBC-WB → FFat*).

For both the filesystems an external tool has been created. These tools allow the transfer of large amounts of bytes from the USB interface directly inside the flash's SPIFFS (or FATFS) file system. These tools are the *MBC ESP32 Spiffs* and the *MBC ESP32 Fatfs*.

MBC ESP32 Spiffs and MBC ESP32 Fatfs

The MBC ESP32 Spiffs tool was derived from Espressif's ESP32FS. Once the tool is running, it basically looks for a "data" subfolder within the sketch folder and generates a .bin file with the contents of the "data" directory. If no data directory is found, the tool asks if a SPIFFS with default content can be loaded or if an empty SPIFFS file should be created. After choosing, another choice must be made, whether upload the file via USB, OTA or not to upload it at all (create it only). In case USB upload is chosen, any open serial monitors must be closed or the upload process may fail.

The FAT tool behaves in the same way as the SPIFFS one, the only difference is the content of the .bin file created: in the first case it contains a FATFS while in the second one a SPIFFS. To summarize, to customize the free storage partition with a custom content, there are a few steps to follow:

— identify the sketch/firmware folder (under Arduino IDE this can be easily done by selecting Sketch → Open sketch folder menu or by clicking *Ctrl + K*, in Visual Studio Code + PIO it's only needed to use the integrated file explorer)
— create a "data" folder in this location
— add all the needed files inside the created data folder
— launch the MBC ESP32 Spiffs or the MBC ESP32 Fatfs tool.

To choose which filesystem to use, whether SPIFFS or FAT, under Arduino IDE select the Tools menu. If the Briki MBC-WB is selected as the board to use, a dedicated entry will appear *ESP Flash Partition*. Using this entry is possible to make the choice between SPIFFS or FFAT storage.

The partition table will be modified accordingly to the selected File System.

When using storage tool updating with USB (SPIFFS or FATFS it is the same), any serial monitor must be closed, or the upload will fail.

The default storage content is the one needed to work with **web panel** from WiFi2Control library

Electrical characteristics

Table 17 - Electrical characteristics

Symbol	Description	Min.	Typ.	Max.	Unit
VCC_SAM	Input supply voltage for SAMD21	2.3	3.3	3.6	V
VCC_ESP	Input supply voltage for ESP32	2.3	3.3	3.6	V
ICC_SAM	Current absorption from SAMD21	6u	10m	50m	A
ICC_ESP	Current absorption from ESP32	10u	150m	300m	A
OPE_T	Operating temperature	-40		80	°C