Aliro software expansion for STM32Cube

Introduction

X-CUBE-ALIRO is a software package that provides a reference implementation of Aliro technology on compatible STM32 microcontrollers.

It allows fast prototyping of Aliro reader devices using ST development platforms and comes preintegrated with Aliro support, already certified with default features. Refer to ST Aliro web page for the supported STM32 platforms.

X-CUBE-ALIRO supports multiple communication flows, including NFC only and BLE remote keyless entry (RKE) modes, as well as BLE-based integration with ultra-wideband (UWB) technology. This flexibility allows developers to tailor their solutions to various use cases and device capabilities.

The software package implements all Aliro transaction types, such as standard expedited transactions and fast transactions, which can operate with or without certificate support. Additionally, step-up transactions allow the use of access and revocation documents, enabling the user to handle advanced access control scenarios, using the data elements in the access decision.

X-CUBE-ALIRO also comes with a preintegrated Bluetooth® LE stack and NFC Forum Type A drivers, streamlining development and reducing integration effort. This ensures compatibility with industry standards and accelerates time-to-market for Aliro-enabled applications.

X-CUBE-ALIRO is compliant with the CSA Aliro specification, ensuring that it meets the technical and interoperability requirements defined by the standard. This compliance allows reliable performance and secure communication across certified devices.

X-CUBE-ALIRO helps to pass the tests against the CSA certification tool.

X-CUBE-ALIRO has been validated for interoperability with major smartphone brands, making it compatible with a wide range of user devices commonly available on the market. This ensures a seamless user experience and simplifies integration into commercial applications.

1 X-CUBE-ALIRO software package

1.1 Features

Integration of Aliro on compatible STM32 microcontrollers with examples of applications for reader device:

- Support for Bluetooth® LE only flow and ready for ultra-wide band integration.
- Support for the NFC only flow operational.
- Certified Bluetooth® LE protocol stack.
- Certified NFC Forum Type A.
- Ready for CSA certification.
- Low power capable.
- Security component (cryptographic library).

Project binary files are provided for immediate demonstration.

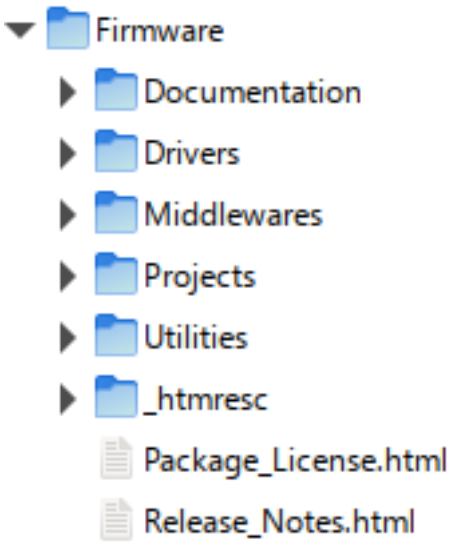
1.2 Architecture

List and describe any environmental, safety, and technical standards that the board is certified for.

1.3 Folder structure

Once you download and unzip X-CUBE-ALIRO, you see the folder structure as shown in Figure 1.

Figure 1. Firmware folder

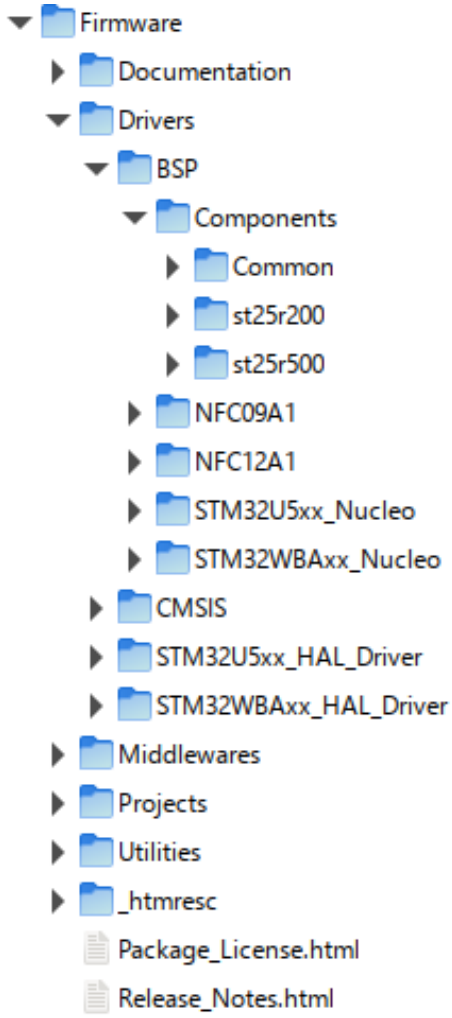


The **Firmware** directory contains all the files required to develop and use the firmware package. As part of this package, the Package_License.html file defines the license terms under which the firmware can be used, while Release_Notes.html documents the package evolution with a detailed list of changes, limitations, and version information. The documentation folder includes the .chm help file, which facilitates the user in getting started with X-CUBE-ALIRO and provides comprehensive technical information, including a complete description of the user APIs, their functions, and parameters.

The **Drivers** directory (shown in Figure 2) provides all the low-level software components required to interface the application with the underlying hardware. Within this directory, the Board support package (**BSP**) folder contains board-specific drivers and components. The **Components** subfolder groups reusable device drivers: the **Common** folder holds shared source files, while the ST25R200 and ST25R500 folders contain device-specific drivers for the corresponding NFC/RFID front-end devices. ST25R500 drivers support both ST25R500 and ST25R300 devices. Additional BSP implementations are provided for the supported expansion boards and kits, such as X-NUCLEO-NFC10A1 and X-NUCLEO-NFC12A1, and under STM32U5xx_Nucleo and STM32WBaxx_Nucleo for the target evaluation boards.

The **CMSIS** folder includes the Arm Cortex® microcontroller software interface standard (CMSIS) files, which define the core processor and device abstraction layer. The **STM32U5xx_HAL_Driver** and **STM32WBaxx_HAL_Driver** folders contain the STM32 hardware abstraction layer (HAL) drivers for the corresponding MCU families, providing a high-level, portable interface to the microcontroller peripherals used by the BSP, middleware, and application code.

Figure 2. Driver subfolder



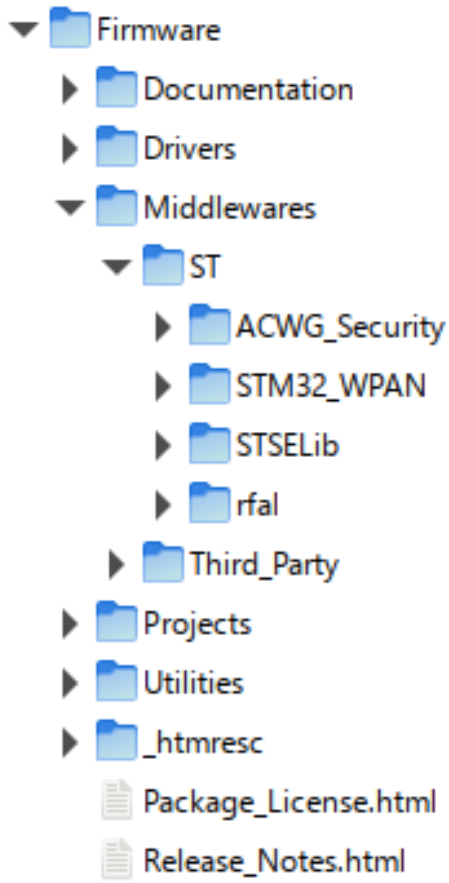
The **Middleware** folder (shown in Figure 3) contains several standard software components commonly used with STM32 microcontrollers. It includes ST middleware related to **ACWG security**, **NFC RFAL**, and Bluetooth® LE **STM32_WPAN**. The ACWG security middleware provides security and authentication services, including cryptographic primitives and protocol support, that can be used to protect communication in Aliro-based systems.

The NFC RFAL (RF abstraction layer) offers a hardware-independent interface for NFC front-ends, simplifying the development of NFC reader/writer, card emulation, and peer-to-peer applications. The Bluetooth® LE STM32_WPAN stack provides Bluetooth® LE connectivity features, including GAP/GATT services, profiles, and link management, enabling low-power wireless communication for STM32-based devices.

The QCBOR and t_cose libraries are included within the **Third_Party** subfolder. QCBOR is an implementation of the concise binary object representation (CBOR) format that focuses on efficient encoding and decoding of structured data, particularly suited for constrained embedded systems. t_cose is a companion library which implements CBOR object signing and encryption (COSE), it is used to sign, verify, and optionally encrypt CBOR data structures. Together, these libraries provide a compact and standards-based foundation for secure data representation and message protection in embedded applications.

Notice: *Please carefully review the documentation related to these libraries. They are third-party components, freely available on GitHub. The alpha releases included in this firmware package are intended solely for testing and prototyping purposes and are not suitable for commercial use. These middleware components are utilized in the Step-Up transaction (seeSection 3.2.4 and Section 3.3.5) and have been successfully tested, with no issues identified.*

Figure 3. Middleware subfolder



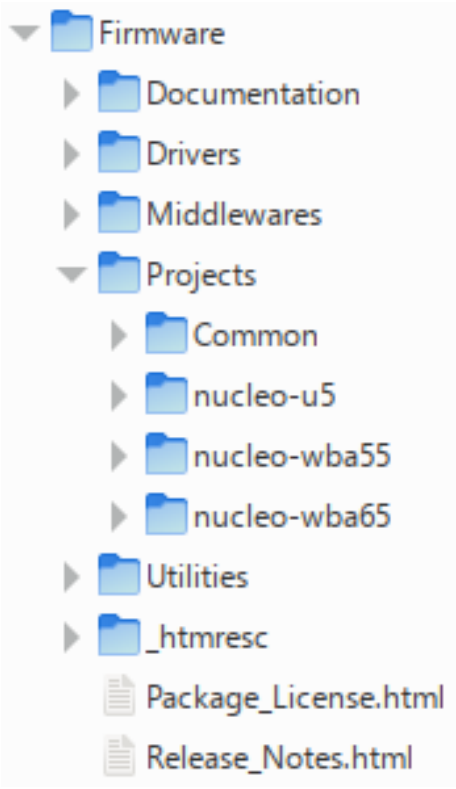
The **Projects** folder (shown in Figure 4) is organized to separate common components from board-specific application codes. The **Common** subfolder contains libraries and source files that are shared by all applications, including both **Aliro** and **WPAN** use cases. In addition to the common code, the **Projects** folder also includes board-specific application files. Specific subfolders are provided for each supported application board, such as NUCLEO-U545RE-Q, NUCLEO-WBA55CG, and NUCLEO-WBA65RI. Each of these board folders contains the application sources, configuration files, and project settings tailored to that hardware platform (such as pin mappings, clock configuration, and peripheral initialization).

Within each board-specific folder, under the **Application** directory, there are further subfolders that indicate the NFC expansion board or NFC configuration used in that project. These NFC-related folders also reflect whether Bluetooth® LE features are supported for that specific board combination. When Bluetooth® LE is available, the application integrates both NFC and Bluetooth® LE functionalities, enabling scenarios where the two technologies can be used alternatively to manage the lock/unlock mechanism.

Below is the list of available projects:

- Projects/nucleo-u5/Applications/Aliro/nfc-only (X-NUCLEO-NFC10A1 (ST25R200)).
- Projects/nucleo-u5/Applications/Aliro/nfc12-only (X-NUCLEO-NFC12A1 (ST25R300)).
- Projects/nucleo-wba55/Applications/Aliro/nfcble (X-NUCLEO-NFC10A1 (ST25R200)).
- Projects/nucleo-wba55/Applications/Aliro/nfc12ble (X-NUCLEO-NFC12A1 (ST25R300)).
- Projects/nucleo-wba65/Applications/Aliro/nfcble (X-NUCLEO-NFC10A1 (ST25R200)).
- Projects/nucleo-wba65 /Applications/Aliro/nfc12ble (X-NUCLEO-NFC12A1 (ST25R300)).

Figure 4. **Project subfolder**



The **Utilities** folder (shown in Figure 5) groups together a set of commonly used support components that are frequently included in STM32 firmware packages. In this context, it contains modules such as **lpm**, **misc**, **sequencer**, **tim_serv**, and **trace**, which provide reusable services to simplify application development and improve code portability across different projects.

The low-power manager (**lpm**) module typically offers an abstraction layer to manage the several STM32 devices low-power modes, handling entry/exit sequences and coordinating power states between different application tasks.

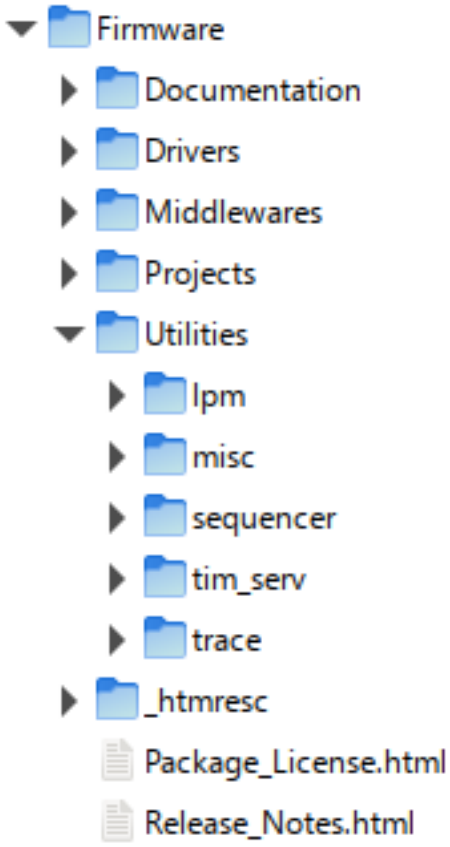
The **misc** subfolder usually collects miscellaneous helper functions and generic utilities that do not belong to a specific peripheral or middleware but are useful in many applications.

The **sequencer** module is generally used as a lightweight task scheduler or event sequencer, allowing the application to organize and execute small tasks in a cooperative way without requiring a full RTOS.

tim_serv (timer service) provides a simple abstraction for software timers, timebase management, and periodic callbacks, often built on top of a hardware timer or SysTick.

Finally, the trace module offers basic tracing and logging capabilities, enabling developers to output debug information over a chosen serial interface to facilitate diagnostics, performance analysis, and troubleshooting during development.

Figure 5. **Utilities subfolder**



1.4 APIs

Detailed technical information with full user API function and parameter description is in a compiled HTML file in the **Documentation** folder.

2 Application description

Aliro is an industry-standard access credential and communication protocol that ensures a secure and consistent experience across user devices to unlock points of entry, defined by the connectivity standards alliance (CSA). This standardized communication protocol revolutionizes access control by enabling seamless interactions between different reader architectures and mobile devices with user credentials.

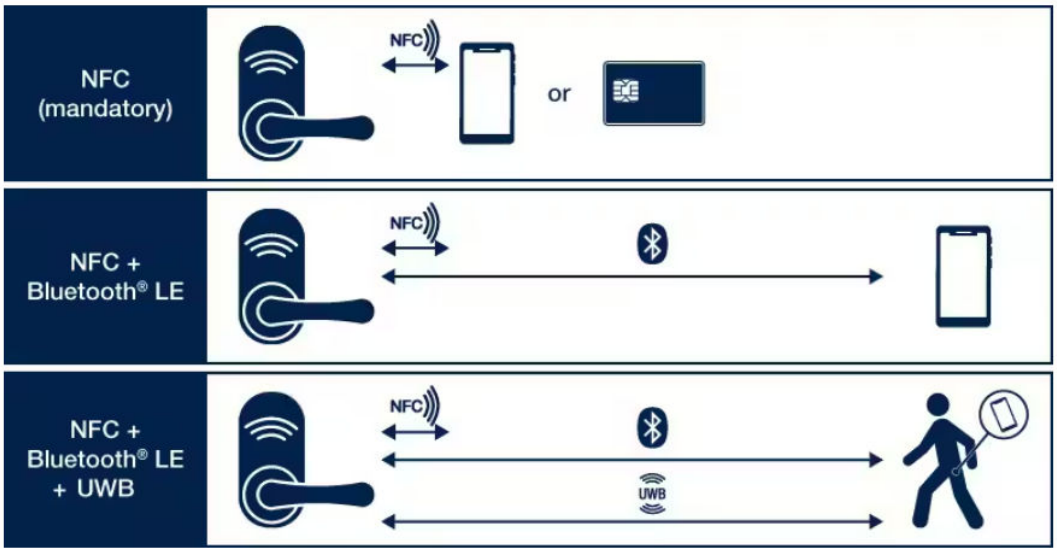
STMicroelectronics is a board member of the CSA and provided technical expertise in the development of the Aliro 1.0 specification. Moreover, ST offers complete hardware and firmware solutions for Aliro applications development.

2.1 Connectivity options

Aliro defines three configurations with different connectivity technologies to enable a secure and contactless exchange of credentials between user devices (for example, mobile, wearable device, or smartcard) and reader devices (for example, access control readers and locks on doors and openings).

- Near Field Communication (NFC) for contactless tap-and-go access use cases.
- NFC + Bluetooth® LE (LE) for access control use cases.
- NFC + Bluetooth® LE + ultra-wideband (UWB) for hands-free access use cases enabled by secure UWB ranging.

Figure 6. Aliro configurations with different connectivity technologies



2.2 Supported hardware and application profiles

Several ST product families support the Aliro standard and are suitable for reader designs. Figure 7 provides a summary matrix of the functionalities supported per board, showing which combinations of Nucleo boards and NFC configuration offer NFC only or combined NFC + Bluetooth® LE features. This also helps the user quickly identify the appropriate project to build and run for a given hardware setup.

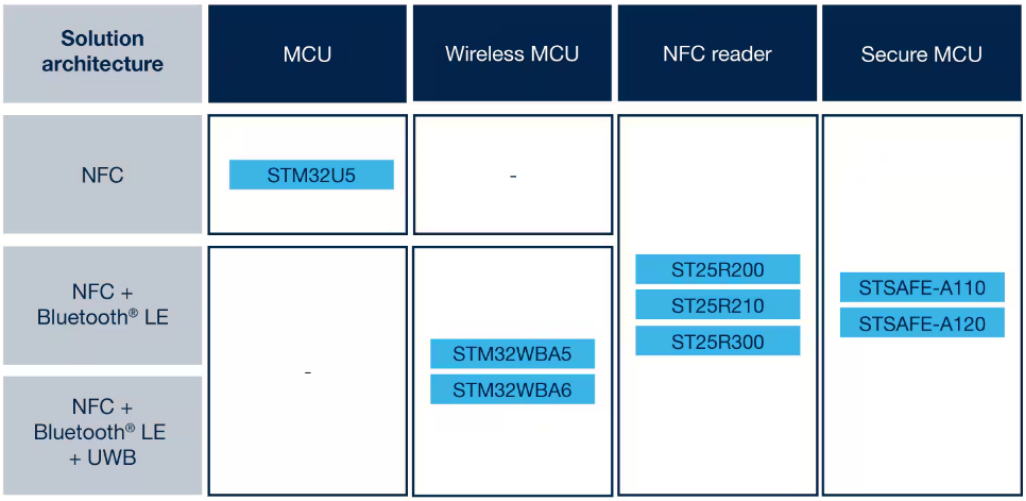
Nucleo boards:

- NUCLEO-U545RE-Q (STM32U545xx).
- NUCLEO-WBA65RI (STM32WBA65xx).
- NUCLEO-WBA55CG (STM32WBA55xx).

X-NUCLEO boards:

- X-NUCLEO-NFC10A1 (ST25R200).
- X-NUCLEO-NFC12A1 (ST25R300).
- X-CUBE-SAFEA1 (STSAFE-A110).
- X-NUCLEO-ESE01A1 (STSAFE-A120).

Figure 7. ST product families that can be combined to support the Aliro standard



3 X-CUBE-ALIRO source code

The complete high-level Aliro protocol code is in the **Common** folder. It also includes high-level protocol files that are still closely related to the hardware but remain platform-independent and shared across all boards, such as those related to NFC.

The board-oriented folders contain the Aliro protocol files that require a direct dependency on the specific hardware being used and therefore must be tied to the implementation platform, such as the Bluetooth® LE services and characteristics.

3.1 Aliro application

The app_aliro.c and app_aliro.c files are the central entry point of the Aliro reader application. It is the place where the system is brought up, where the main execution loop runs, and where the command-line interface over UART is implemented. All the other functional domains – NFC, Bluetooth® LE, UWB, and provisioning – are coordinated from here through two session contexts, context_nfc and context_ble data structures, which represent the current state of an Aliro session over NFC and Bluetooth® LE respectively.

3.1.1 How the Aliro application works

When the firmware starts, the aliro_init function performs the overall application initialization. It begins by setting up logging and time-stamping.

Then, the code verifies and reports the software versions currently running on the board. It retrieves the application build information from the app_build_info data structure and queries the security library via the ACWG_SecurityGetVersion function. Both sets of data are printed and then compared against the Aliro specification version expected by the firmware. If a mismatch is detected – for example, if the application and the security library were built against different specification revisions – it is explicitly logged and treated as a fatal error. This way, aliro_init ensures that all Aliro-related components are version-aligned at startup.

Initialization then moves on to the cryptographic subsystem. The CryptoInit function inside app_aliro.c configures the hardware cryptographic engines by calling SM_initCryptoMCU with the handles for the random number generator (RNG), public key accelerator (PKA), hashing (HASH) and Cryptography (CRYP) peripherals. It then calls the CheckReaderPubK function to verify that the reader public key is present and valid in flash. If provisioning is missing or inconsistent, the application jumps to the aliro_process function with the isAliroRunning variable set to false. This prevents any Aliro transaction on an unprovisioned reader.

With cryptography in place, the aliro_init function proceeds to bring up NFC. This happens in two steps: the AliroNfcHwInit function configures the low-level hardware, such as the SPI bus and GPIOs, and the interrupts associated with the ST25R front-end, meanwhile AliroNfcIni initializes the RFAL NFC stack and associates it with the NFC session context. From this point on, the context_nfc data structure contains both the configuration and the runtime state for Aliro transactions over NFC.

If Bluetooth® LE is supported, meaning that the code has been compiled with ALIRO_BLE defined as 1 in app_aliro.h, the Bluetooth® LE stack is also initialized. The Aliro application module responsible for Bluetooth® LE communication is the app_ble.c file. It works together with the app_aliro.c file through a set of callbacks and by maintaining the context_ble session structure, which is kept consistent with the NFC one.

The hardware described in Section 2.2 does not include UWB functionality; however, the X-CUBE-ALIRO firmware package is prepared to support UWB features. This capability is initialized by calling the weak function UWB_HW_init and is managed through additional weak functions, which are described in Section 3.4.3.

Finally, the aliro_init function focuses on the session parameters initialization that oversees Aliro behavior. For each of the context data structures (context_nfc and context_ble) the initialized parameters are:

- Authentication policy (**context_nfc.init.authPolicy**, **context_ble.init.authPolicy**).
- The handling of “step-up” operations (**context_nfc.init.init.useStepUp, context_ble.init.init.useStepUp**).
- The behaviour of the EXCHANGE phase (**context_nfc.init.useExchange, context_ble.init.useExchange**).
- The way certificates are loaded (**context_nfc.init.useLoadCert, context_nfc.init.useLoadCert**). It depends on whether a certificate has actually been provisioned in flash or not.

Moreover, the tlw_SetAccessElementString function defines the access document ID to be requested during the step-up transaction.

Once initialization has completed, the control passes to the main application loop implemented by the aliro_process function. This loop acts as a scheduler for the different communication stacks. On each step, if the **isAliroRunning** flag is true, that means initialization and check provisioning were successfully completed, the application loop executes the following:

- The NFC state machine by calling the AliroNfcCycle function.
- The Aliro Bluetooth® LE state machine with the tlw_ble_SM_step function (if Bluetooth® LE is enabled).
- The UWB-related processing by calling the UWB_HW_loop weak function.

Moreover, regardless of the isAliroRunning flag state, one activity is always performed on every iteration of the aliro_process function. It is the UART commands handling managed by the ProcessUartInput function. It implements a simple command-line interface that allows an operator to inspect and modify the configuration at runtime. It supports both single-character commands and longer textual commands. This feature is described in Section 5.3.

The app_aliro.c file also includes a small set of event callbacks. These functions are declared as weak so that they can be easily overridden by the user.

For example, when a transaction successfully opens a door, the AliroPort_TRANSACTION_SUCCESS_OPEN function is called. In the default implementation, it controls the LEDs and prints an ASCII-art “OPEN” banner to the serial console. In the same way, the AliroPort_TRANSACTION_SUCCESS_CLOSE function is provided for successful close transactions. On the other hand, if a transaction fails at any point in the protocol, the AliroPort_TRANSACTION_FAILURE function is called. All these functions can be customized by the user to drive, buzzers, or external indicators, while keeping the Aliro protocol logic unchanged.

app_aliro.c includes a few utilities functions that are used as commodities in the application. They are:

- PrintBufferHex and PrintBufferHexAlways, which help print buffers in hexadecimal format.
- PrintTimeStamp, PrintTimeStampReport, and ResetTimeStamp, which are timestamp functions that allow measuring and reporting timing between key events for statistical purposes.
- Some RTC utilities used to read, display, and to update (when allowed) the current date and time.

If the Bluetooth® LE functionality is available, the app_aliro.c file provides some support functions to manage Aliro-specific advertising parameters and dynamic SPSM generation:

- The AliroSvcParamGetInitBuf function helps to build the initial service parameter buffer.
- The AliroSvcVerifyProtVer function verifies the protocol version.
- The isValidSPSM function verifies the SPSM selection, which was negotiated with a client.

Finally, the app_aliro.c file wraps a few HAL functions into Aliro-specific interfaces for UART transmit/receive, delays, system reset, and GPIO management.

3.1.2 How the Aliro application is set

The app_aliro.h file is the public header for the Aliro application. It exposes types, macros and function prototypes used in the firmware, and it is also where most of the compile-time options are defined. From a user perspective, this is the file you look at if you want to enable or disable features with #defines or override behavior using _weak hooks.

Compile-time options and configuration macros

At the beginning, the header defines a few global switches:

- **ALIRO_MEASURE_PERFORMANCE:**
  When set to 1, the performance measurement is enabled. It means that macros like RESET_TIME_STAMP, PRINT_TIME_STAMP, and TIME_STAMP_REPORT point to functions that record and print timing information between key protocol phases (select app, auth0/auth1, load cert, exchange, step-up, door open, etc.). With ALIRO_MEASURE_PERFORMANCE left to 0, these macros point to empty functions, so no measurement action is performed.
- **ALIRO_CERTIFICATION_TEST:**
  This flag enables a certification test scenario.

There are two definers which allow setting the Aliro flows supported over Bluetooth® LE:

- **BLE_RKE_ALIRO_FLOW_SUPPORTED** (0x40):
  It indicates that the “BLE-only” Aliro remote keyless entry flow is supported.
- **BLE_UWB_ALIRO_FLOW_SUPPORTED** (0x80):
  It indicates that the combined “Bluetooth® LE + UWB” Aliro flow is supported.

Moreover, the bit mask:

- **BLE_ALIRO_FLOWS_SUPPORTED:**
  It combines the two Bluetooth® LE flow definers and it is used when advertising data are built, so that the mobile device can see what the reader is capable of.
  By default, it is defined as (BLE_UWB_ALIRO_FLOW_SUPPORTED | BLE_RKE_ALIRO_FLOW_SUPPORTED) to support both flows.

The app_aliro.h header file also centralizes several timeouts for UWB and door operations:

- TIMEOUT_PREPARE_RANGING_DEFAULT.
- TIMEOUT_EXECUTE_RANGING_DEFAULT.
- TIMEOUT_SUSPEND_RANGING_DEFAULT.
- TIMEOUT_RESUME_RANGING_DEFAULT.
- TIMEOUT_DOOR_OPERATION_DEFAULT.

They are the default timeouts in milliseconds, which indicate the maximum allowed time for each phase of the UWB ranging and door operations. At runtime, they can be overridden via the following functions called inside the UWB_HW_init function:

- Set_timeout_prepare_ranging.
- Set_timeout_execute_ranging.
- Set_timeout_suspend_ranging.
- Set_timeout_resume_ranging.
- Set_timeout_door_operation.

The header file includes definitions that adapt compilation to the hardware being used. Using #if defined(...) blocks, the header selects:

- Whether Bluetooth® LE support is compiled in (ALIRO_BLE).
- Which UART and timer handles the application should use (H_ALIRO_UART, HH_ALIRO_UART, H_TIM).
- The RTC handle and backup register (ALIRO_HRTC, ALIRO_RTC_BKP_TIME).
- Which GPIO pins are mapped to drive LEDs (LD2, LD3).
- In which flash sectors the provisioning data and provisioning JSON are stored (PROVISIONING_START_ADDRESS, PROVISIONING_JSON_START_ADDRESS).

This means that by changing these #defines (or the active platform macro), you can retarget the Aliro application to a different STM32 device or board while keeping the core application unchanged.

The PLATFORM_LOG macro implements logging verbosity. A message is printed only if its verbosity level is less than or equal to the current nVerbosityLevel. At startup, nVerbosityLevel is set to **VERB_VERBOSE** but it is possible to change it in runtime.

3.1.3 How the Aliro application is customized

Weak functions for application customization:

As already described above, the header declares several functions with the _weak attribute. This allows the user to customize the code providing his own implementation. The user shall override this function to perform the customized actions.

The most relevant weak functions are:

- Bluetooth® LE life cycle callbacks:
  - **_weak void AliroPortBLE_Init_complete_cb(void):**
    Called when the Bluetooth® LE stack initialization completes (for example, start advertising a second service, toggle a status LED).
  - **_weak void AliroPortBLE_Connection_complete_cb(void):**
    Invoked when a Bluetooth® LE connection is successfully established.
  - **_weak void AliroPortBLE_Disconnection_request(void):**
    Invoked when a Bluetooth® LE disconnection is needed.
  - **_weak void AliroPortBLE_Disconnection_complete_cb(void):**
    Invoked when a Bluetooth® LE connection is closed.
  - **_weak void AliroPortBLE_CocDisconnection_complete_cb(void):**
    Invoked when a Bluetooth® LE credit-based connection (CoC) channel is disconnected.
- Transaction outcome:
  - _weak void AliroPort_TRANSACTION_SUCCESS_OPEN(void).
  - _weak void AliroPort_TRANSACTION_SUCCESS_CLOSE(void).
  - _weak void AliroPort_TRANSACTION_FAILURE(void).

  These three functions are called at the end of an Aliro transaction, depending on its outcome (door open success, door close success, or failure). The default implementations drive LEDs and print banners on the serial console.
  Moreover, some macros are linked to those weak functions. They are:
  - TRANSACTION_SUCCESS_OPEN.
  - TRANSACTION_SUCCESS_CLOSE.
  - TRANSACTION_FAILURE.
- Actuation:
  - _weak uint32_t LockAction(uint32_t lockActionRequest).
    LockAction is the lock-control abstraction function used by the NFC and BLE/UWB transaction flows to apply a secure/unsecure command and to report the reader lock state. It is called by the TLWrapper when preparing reader status changed exchange (lock status update). **The user shall override this weak function to call real lock actuator APIs**.
    The lockActionRequest function parameter can assume the following values:
    1. 0x00: Request to secure the lock.
    2. 0x01: Request to unsecure the lock.
    3. 0xFF: No explicit action request.

    The return value (lock status code) can be:
    1. 0x00: Secure state (locked/armed/closed).
    2. 0x01: Unsecure state (unlocked/disarmed/opened).
    3. 0x02: Obstructed/jammed/stuck.
    4. 0x80: Transition to secure started.
    5. 0x81: Transition to unsecure started.
    6. 0x82: Unknown/unavailable state.
- Access Doc (step-up):
  - _weak uint8_t AliroPort_ParseCriticalityAccessExtensionsData(uint8_t *data, size_t len).
    The AliroPort_ParseCriticalityAccessExtensionsData function is used to analyze the “data” field of the criticality access extensions received during the step-up Aliro flow inside the access document.
    It aims to validate and to interpret extensions marked as critical for the lock actuation decision, allowing the application to accept only supported and secure data.
    The function returns:
    - **0**: valid data; critical extensions were recognized and parsed correctly.
    - **1**: parsing error for critical extensions (for example, unknown or unsupported critical data detected).
      If the function returns 1, the application flow treats the received critical content as invalid and handles it as an error in the Aliro flow.

3.2 NFC application

The app_nfc.c file contains the NFC application functions to support the Aliro protocol. Meanwhile, app_aliro.c manages the overall application. This file is specifically responsible for discovering NFC smartcards/phone, activating them, and running the Aliro protocol over ISO-DEP.

3.2.1 NFC init

At startup, the AliroNfcIni function is called by aliro_init. It initializes the RFAL NFC stack and configures the discovery parameters listed in the discParam structure. It selects which NFC technologies to poll (for example, NFC-A, optionally NFC-V), sets basic P2P/Type 4 parameters (NFCID3, general bytes), installs a notification callback (demoNotif function), and attaches proprietary callbacks for custom NFC handling. Finally, it starts RFAL discovery with the rfalNfcDiscover function and sets the internal state machine to START_DISCOVERY.

3.2.2 NFC loop

The runtime behavior is driven by the AliroNfcCycle function, which is called periodically in the main loop. It runs the rfalNfcWorker function and advances a simple state machine. During the discovery state it waits for RFAL to activate a device. When a smartcard/phone is detected, it logs its type and UID and, if it is an NFC-A Type 4 (ISO-DEP) card, calls the demoAPDU function to execute the Aliro transaction. After handling the smartcard/phone, it uses the rfalNfcDeactivate function to deactivate NFC and to return to START_DISCOVERY state for the next cycle.

3.2.3 Aliro NFC flow

The demoAPDU function is the bridge between NFC and the Aliro protocol. It initializes the TLWrapper through the tlw_init_nfc function and then loops the following actions:

- It lets the TLWrapper prepare the next APDU through the tlw_prepare_nfc function.
- It sends it to the smartcard/phone using the demoTransceiveBlocking function.
- It feeds the response back into the TLWrapper state machine with the tlw_process_nfc function.

When the TLWrapper reports that the sequence is complete, demoAPDU checks whether the transaction ended in a valid success state and, if so, verifies that the endpoint public key matches one of the provisioned devices by calling the VerifyEndpointPubKey function. At the end, the outcome function is called as the weak function described in Section 3.1.3

3.2.4 Aliro TLWrapper detailed NFC flow

The following sequence describes the execution of the Prepare/Process phase to implement the Aliro protocol in TLWrapper, showing how each state generates data, validates responses, and handles the subsequent protocol transition.

1. SELECT - prepare and process:
   - The system builds and sends the Aliro SELECT command.
   - The response is parsed and validated before any authentication step starts.
   - After PROCESS_SELECT, the state machine moves to PREPARE_AUTH0.
2. AUTH0 (with FAST-to-STD fallback):
   - The system generates the AUTH0 request payload using the current session context and selected protocol parameters.
   - AUTH0 cryptographic payload is processed and verified.
   - If the FAST transaction fails (ACWG_Error_FastTransFailed), the flow does not terminate; it falls back to standard AUTH0 (back to PREPARE_AUTH0, with FAST path disabled).
   - If AUTH0 fails in a non-recoverable way, flow goes to control-flow/error handling.
3. LOAD_CERT (optional, can be chained and only for provisioning with certificate available):
   - After AUTH0, the path depends on the useLoadCert flag:
     - 1 or 2 → PREPARE_LOAD_CERT.
     - 4 → chained path via PREPARE_AUTH1_CHAINED.
     - Otherwise → direct AUTH1 (LOAD_CERT bypass).
   - In PROCESS_LOAD_CERT, if certificate data spans multiple frames, the function returns TLW_NEED_LOAD_CERT and remains in LOAD_CERT loop until complete. So, NFC explicitly supports multi-frame certificate loading.

4. AUTH1 (including chained handling):
   - In PROCESS_AUTH1:
     - Validates SW/length.
     - If payload is too short, it returns TLW_NEED_AUTH1 and continues in AUTH1_CHAINED.
     - When complete, it calls ACWG_processAUTH1ResponsePayload.
   - On success:
     - Endpoint key is stored.
     - Auth result is marked successful.
   - On failure:
     - Flow is redirected to reader status exchange/negative completion path.
5. Dynamic decision: STEPUP and EXCHANGE:
   After FAST AUTH0 completion or after AUTH1, the next transitions are selected by runtime policy and signaling bitmap.
   - STEPUP gate:
     - Forced if useStepUp >= 2.
     - Conditional if useStepUp = 1 and tlw_need_stepup(...) is true.
   - Select StepUp AID handling:
     - if the “StepUp AID requested” bit is set, flow goes PREPARE_SELECT_STEPUP first, then PREPARE_STEPUP.
     - Otherwise, it goes directly to PREPARE_STEPUP.
   - EXCHANGE gate:
     - Decided by tlw_need_exchange(...) (useExchange flags + mailbox bits in signaling bitmap).
   - STEPUP/EXCHANGE order:
     - Build-time controlled by:
       - EXCHANGE_THEN_STEPUP.
       - STEPUP_THEN_EXCHANGE.
6. STEPUP and GET RESPONSE:
   - In PROCESS_STEPUP, if more data is required, the function returns TLW_NEED_GET_RESPONSE and state goes to PREPARE_GET_RESPONSE.
   - Then it returns to PROCESS_STEPUP until completion.
7. EXCHANGE:
   - In PROCESS_EXCHANGE, response is validated and exchange payload is processed.
   - After EXCHANGE, STEPUP may be re-evaluated (if the policy says EXCHANGE first).
   - If no more mandatory steps remain, flow goes to reader status exchange.
8. Step completion (NFC):
   Once AUTH/STEPUP/EXCHANGE is completed:
   - Transition to PREPARE_READER_STATUS_EXCHANGE → PROCESS_READER_STATUS_EXCHANGE:
   - Final outcome:
     - NFC_END if authResult is successful.
     - NFC_ERROR if auth/exchange failed.

3.3 Bluetooth® LE application

The Bluetooth® LE implementation of ALIRO is structured in three layers:

- The Bluetooth® LE stack/application layer includes the standard roles and protocols such as GAP, GATT, and L2CAP.
- The ALIRO service layer is responsible for protocol version control and to manage the SPSM.
- The ALIRO TLWrapper layer implements the state machine of the application-level protocol.

Bluetooth® LE initialization in ALIRO is carried out in a few well-defined steps. During the bootstrap phase, the Bluetooth® LE host is initialized together with the GAP, GATT, and the service controller components. Once this basic stack is ready, the ALIRO service and its related characteristics are created.

3.3.1 ALIRO service and GATT characteristics setup

Service and characteristics are defined inside alirosvc files, generated by STM32CubeMX. The ALIRO service is instantiated with the 16-bit UUID **0xFFF2**. Within this service, two main GATT characteristics are defined:

- **R_SPSM (read)**: this characteristic contains the proposed SPSM value and the list of supported protocol versions.
- **D_AC_PV (write)**: this characteristic is used by the peer device to send back the selected protocol version.

The initial payload of the **R_SPSM** characteristic is organized as follows:

- **bytes 0–1**: the proposed SPSM (this value is dynamic).
- **byte 2**: the length of the protocol version list.
- **Subsequent bytes**: supported protocol versions.
- **Last 2 bytes**: capability field.

At the end of the initialization process, a dedicated ALIRO Bluetooth® LE completion callback is invoked. This callback enables advertising management logic and triggers the preparation of the advertising data.

3.3.2 Advertising preparation and payload content

Advertising is started via an advertising request callback. Before each advertising start or update, the payload is rebuilt by the Update_a_AdvData function, which calls the fillAdvBuffer function inside the app_aliro.c file.

The advertising payload is structured respecting the maximum buffer size of 31 bytes as follows:

- **bytes 0–2**: standard Bluetooth® LE flags (0x02, 0x01, 0x06).
- **bytes 3–7**: ALIRO header, which includes the AD type for service data, the service UUID, and the supported flows.
- **byte 8**: TX power.
- **bytes 9–16**: reader group ID.
- **bytes 17–18**: reader group sub-ID.
- **bytes 19–22**: expiry timestamp (stored in little-endian format in the buffer).
- **byte 23**: RFU (reserved for future use).
- **bytes 24–30**: a 7-byte dynamic tag, computed using AES over the MAC address and the timestamp.

Advertising is also automatically restarted after a Bluetooth® LE disconnection or a credit-based connection-oriented channel (CoC) disconnection, ensuring that the device remains discoverable and ready for new connections.

3.3.3 Begin credit-based connection-oriented channel (L2CAP CoC) connection

After the LE connection is completed, the application transitions into the CONNECTED SERVER state and notifies the ALIRO service layer that a new link is available.

At this stage of the connection, the peer device starts the protocol negotiation by first reading the **R_SPSM** characteristic, and then writing its choice into **D_AC_PV**.

When the write to **D_AC_PV** is received, the application extracts the protocol version selected by the peer. This value is then passed to a verification routine, which checks whether the requested version is included in the list of versions supported by Aliro.

If the requested version matches one of the supported ones, it is accepted and stored as the currently active protocol version for the session. If, instead, the version is not recognized or not supported, the negotiation is considered invalid and the application reacts by ending the connection, preventing the session from continuing with an incompatible protocol.

Following successful negotiation, a vendor-specific event is received to request the opening of L2CAP CoC.

The application then retrieves the CoC parameters (MTU, MPS, initial credits, SPSM, and channel number) and validates the SPSM. If the SPSM does not match the previously proposed value, the CoC channel is rejected. If it matches, the application sends a connect confirmation.

Once the CoC connection has been accepted, an ATT MTU exchange is performed, and the CoC data session becomes fully operating.

At this stage:

- Incoming CoC data (Rx) is forwarded to the ALIRO TLWrapper by calling the tlw_ble_receivedData callback function.
- Outgoing data from the TLWrapper (Tx) is sent over the CoC channel by the COC_SendData function, which is passed as a pointer in the tlw_ble_receivedData function parameters.

3.3.4 Aliro Bluetooth® LE flow

The tlw_init_ble function is triggered once on the notification frame and it is handled by the tlw_ble_SM_step loop, which alternates tlw_process_ble and tlw_prepare_ble until frames are consumed or responses are emitted. At the transaction completion, as well as at the Bluetooth® LE disconnection or CoC channel drop, the tlw_reload_ble function is invoked. This routine resynchronizes the session context (including derived keys, signaling bitmap, and internal state) and restores the system to a consistent and secure state for a new negotiation, ensuring that no residual data from the previous session can compromise subsequent authentication or access.

3.3.5 Aliro TLWrapper detailed Bluetooth® LE flow

The Aliro state machine over Bluetooth® LE implemented in the TLWrapper follows a two-step model in which each logical phase is split into a preparation state and a processing state. In the PREPARE states, the outgoing payload is built (APDUs, events, session parameters), while in the PROCESS states, the received payload is validated and interpreted, updating the cryptographic context and the progression of the protocol.

1. AUTH0 (cryptographic bootstrap):
   - In PREPARE_AUTH0, the first command to the reader is generated.
   - In PROCESS_AUTH0, the response is processed (ACWG_processAUTH0ResponsePayload).
2. LOAD_CERT (certificate loading/validation):
   - PREPARE_LOAD_CERT builds the certificate request.
   - PROCESS_LOAD_CERT checks the response.
3. AUTH1 (completion of mutual authentication):
   - PREPARE_AUTH1 creates the final authentication message.
   - PROCESS_AUTH1 validates the response (ACWG_processAUTH1ResponsePayload).

   The next step after AUTH1 (and then potentially again after STEPUP/EXCHANGE) is not fixed, it depends on runtime and compile-time flags.
4. STEPUP (stronger authentication):
   a. PREPARE_STEPUP prepares an additional envelope/challenge.
   b. PROCESS_STEPUP processes the response.

   When exiting STEPUP, the machine re-evaluates whether EXCHANGE is required, based on the compiled policy.
5. EXCHANGE (protected application data exchange):
   a. PREPARE_EXCHANGE generates the exchange APDUs.
   b. PROCESS_EXCHANGE validates the response (ACWG_processExchangeResponsePayload).
6. Exit from the auth/exchange phase:
   a. If the endpoint key checks and the policy conditions are satisfied, the machine transitions to PREPARE_AP_COMPLETED.
   b. If the endpoint key is not valid or the conditions are inconsistent, it transitions to PREPARE_NOTIFICATION_EVENT_ERR with a generic error.
   c. From this point, the flow differs according to the session:
      - BLE-only: RKE.
      - Bluetooth® LE + UWB.

   AUTH0–LOAD_CERT–AUTH1 establish the cryptographic trust, STEPUP and EXCHANGE are inserted conditionally based on the flags, and the flow reaches AP_COMPLETED state only when all gates are successfully passed.

3.3.5.1 *BLE-Only: RKE flow*

1. RKE command decryption & interpretation:
   – The incoming RKE command is decrypted via **ACWG_decryptBle**.
   – The payload is then mapped to the corresponding action:
      ◦ 0x00 – Secure/Lock request.
      ◦ 0x01 – Unsecure/Unlock request.
2. Reader status & context realignment:
   – After command evaluation, the system enters the reader status exchange phase.
   – Communication is closed and Bluetooth® LE context is then synchronized via **tlw_reload_ble**.

3.3.5.2 *BLE-UWB flow*

1. UWB session initialization (BLE-Assisted):
   The protocol enters the UWB preparation pipeline:
   – PROCESS_TIME_SYNC: Time synchronization.
   – PROCESS_INITIATE_RANGING_SESSION: UWB session setup.
   – Message exchange sequence:
      ◦ PREPARE_M1 → PROCESS_M2 → PREPARE_M3 → PROCESS_M4.
      These steps establish the secure UWB ranging session.
2. Active UWB ranging phase:
   After completing M4, the system begins operational ranging:
   – PREPARE_RANGING.
   – EXECUTE_RANGING.
   – Optional session flow control:
      ◦ HANDLE_SUSPEND.
      ◦ HANDLE_RESUME.
   – CLOSE_RANGING.
3. Reader status & context realignment:
   – After command evaluation, the system enters the reader status exchange phase.
   – Communication is closed and Bluetooth® LE context is then synchronized via **tlw_reload_ble**.

3.4 UWB custom module API

3.4.1 Purpose

This module provides the application-level weak functions used to configure, run, and control an ALIRO UWB ranging session.

It covers:

- UWB session context initialization.
- M1/M2/M3/M4 message handling.
- Ranging life cycle control (prepare, execute, close, suspend, resume).
- UWB hardware initialization and periodic servicing.

3.4.2 Integration notes

All the proposed weak functions arise the following rules:

- All session callbacks receive ACWG_SessionContext_t *ctx.
- Loop callbacks receive uint32_t *timeout as an output control:
  – Timeout = 1 means exit loop and continue state-machine flow.
  – Timeout = 2 is used as a timeout/fallback condition during the execute phase.
- Distance-based open decision is performed against DISTANCE_OPEN.
- Default “no valid measurement yet” condition uses DEFAULT_FAR_DISTANCE.

3.4.3 API reference

The APIs described here are intended to be called by the Aliro session state machine. Those functions are in the custom_weak_override_uwb.c file and they can be customized by the user. They are grouped by type and described below.

Session and synchronization

- **Initiate_uwb_ctx**(*ctx) initializes UWB session context before negotiation starts.
- **Handle_uwb_Time_Sync**(*ctx) processes UWB time synchronization frame for the current session.

UWB negotiation messages (M1/M2/M3/M4)

Parameters can be set through the **ctx->uwb_ctx** structure:

- **Fill_uwb_M1**(*ctx) fills outbound M1 with reader-selected UWB capability and session fields.
- **Handle_uwb_M2**(*ctx) handles inbound M2 and stores/validates peer-selected configuration.
- **Fill_uwb_M3**(*ctx) fills outbound M3 with final session parameters selected by the reader.
- **Handle_uwb_M4**(*ctx) handles inbound M4 containing final peer-provided session values (for example, Time0/STS-related data).

Ranging life cycle: prepare phase

- **Prepare_ranging_init**(*ctx) is a one-time initialization to prepare the ranging phase.
- **Prepare_ranging_loop**(*ctx, *timeout) is periodically called to achieve the preparing phase until the timeout condition is met.
  - Set timeout = 1 to exit the preparing loop.

Ranging life cycle: execute phase

- **Execute_ranging_init** (*ctx) one-time initialization to prepare the execute phase.
- **Execute_ranging_loop** (*ctx, *timeout) is periodically called to achieve the ranging phase until the timeout condition is met.
  - If no valid ranging value is observed before guard timeout, set timeout = 2.
  - If the distance drops below open threshold, set timeout = 1.

Ranging life cycle: close phase

- **Close_ranging_init** (*ctx) one-time initialization to prepare the close phase.
- **Close_ranging_loop** (*ctx, *timeout) is periodically called to achieve the closing phase until the timeout condition is met.
  - Set timeout = 1 to exit the closing loop.

Ranging life cycle: suspend phase

- **Suspend_ranging_init** (*ctx) is a one-time initialization to prepare the suspend phase.
- **Suspend_ranging_loop** (*ctx, *timeout) is periodically called to achieve the suspending phase until the timeout condition is met.
  - Set timeout = 1 to exit the suspend loop.

Ranging life cycle: resume phase

- **Resume_ranging_init** (*ctx) is a one-time initialization to prepare the resume phase.
- **Resume_ranging_loop** (*ctx, *timeout) is periodically called to achieve the resuming phase until the timeout condition is met.
  - Set timeout = 1 to exit resume loop.

Measurement

- **Get_ranging_measurement** (*ctx) returns the latest mean ranging result in centimeters.
  - Return uint32_t mean distance in cm.

Hardware services

- **UWB_HW_init** () initializes UWB hardware-related timing parameters and local measurement state.
  - It loads default timeout values to prepare, execute, suspend, resume, and for door operation.
  - It resets internal measurement buffers/state.
  - It applies startup delay.
- **UWB_HW_loop** () is a periodic hardware service function called from the main loop.

3.4.4 State-machine sequence

The UWB follows a state machine described below:

1. UWB_HW_init.
2. Initiate_uwb_ctx.
3. Fill_uwb_M1 → Handle_uwb_M2 → Fill_uwb_M3 → Handle_uwb_M4.
4. Prepare_ranging_init → Prepare_ranging_loop.
5. Execute_ranging_init → Execute_ranging_loop.
   - Optional branches:
     a. Suspend_ranging_init → Suspend_ranging_loop.
     b. Resume_ranging_init → Resume_ranging_loop.
6. Close_ranging_init → Close_ranging_loop.

3.5 STSAFE

A test function is integrated to verify that the I²C communication is operational and that the entire ecosystem is correctly configured. To use it, you need to:

- Set the *#define STSAFE_TEST* flag to 1 in main.h.
- Modify *stse_handler.device_type = STSAFE_A120* depending on the STSAFE model being used.
- The stse_init function performs both the driver and hardware initialization.
- The stsafe_echo function uses the stse_device_echo API to verify that communication is working correctly.

4 Start with X-CUBE-ALIRO

4.1 Select project

Choose the project that matches the hardware configuration you have from the list of projects presented in Section 1.3.

4.2 How to build a project

Inside each project folder, there is the structure needed to manage the project with different IDEs or, if preferred, as a CMake-based project. The supported development environments include IAR and STM32CubeIDE.

5 Using X-CUBE-ALIRO

Once the correct firmware has been loaded into the reader, follow these steps to make the device operational:

Step 1. Connect the device via USB to access the serial interface.

Step 2. Provision the device with the correct keys.

Step 3. If necessary, set the various operating parameters using the available commands.

5.1 Serial interface

All projects implement a textual human interface via the ST-LINK programmer. Please follow the instructions carefully to ensure successful use of the serial interface:

Step 1. Connect the ST-LINK programmer inside the Nucleo board to your PC via a USB port.

Step 2. Open a serial terminal (in this guide, Tera Term is used).

Step 3. Select **File** followed by **New Connection** to select the serial COM port exposed by the ST-LINK programmer.

Step 4. Select **Setup** followed by **Serial Port** to configure the serial COM port parameters, then click on **New Settings**.

As soon as the Aliro application firmware is running, the log messages are displayed.

Figure 8. Tera Term: new connection

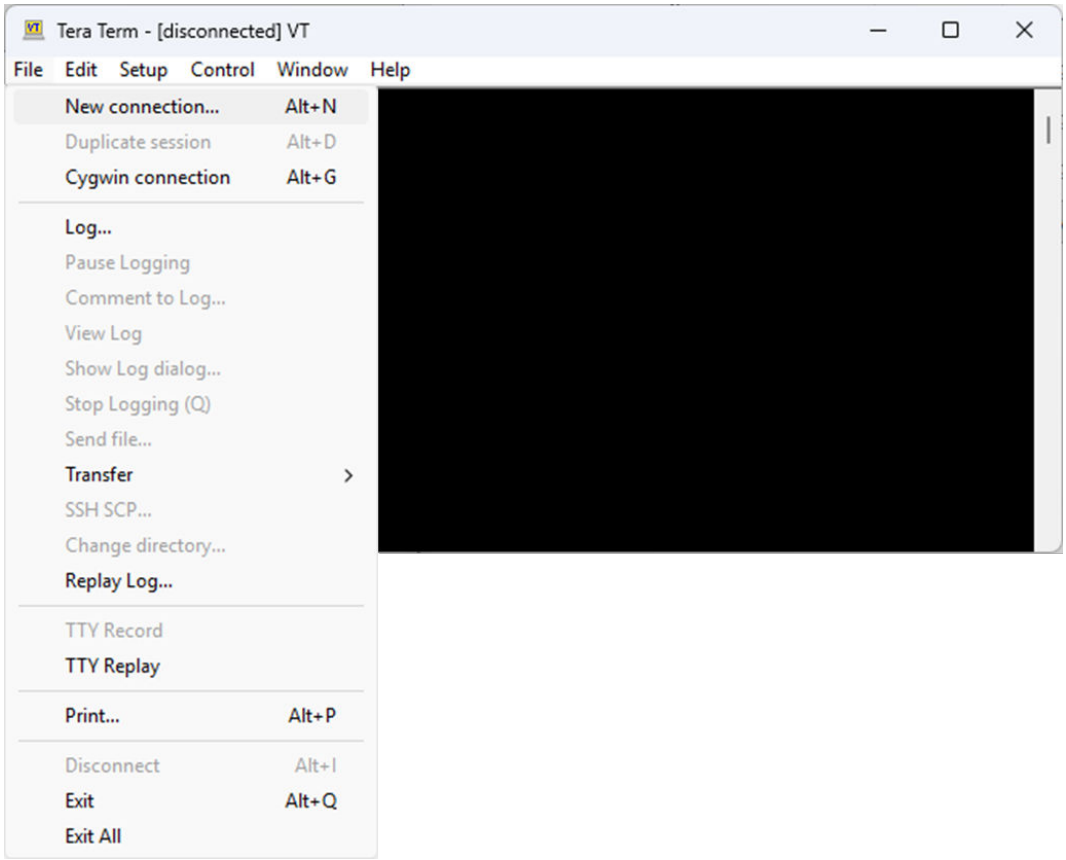


Figure 9. **Tera Term: access the serial port parameters**

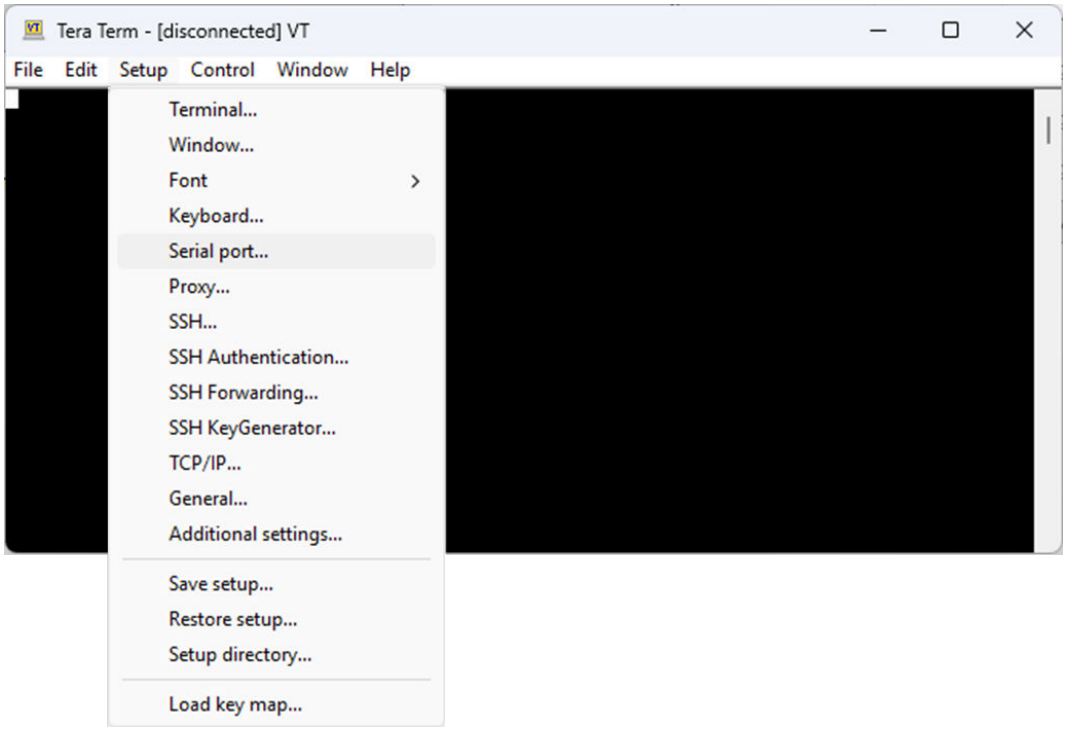
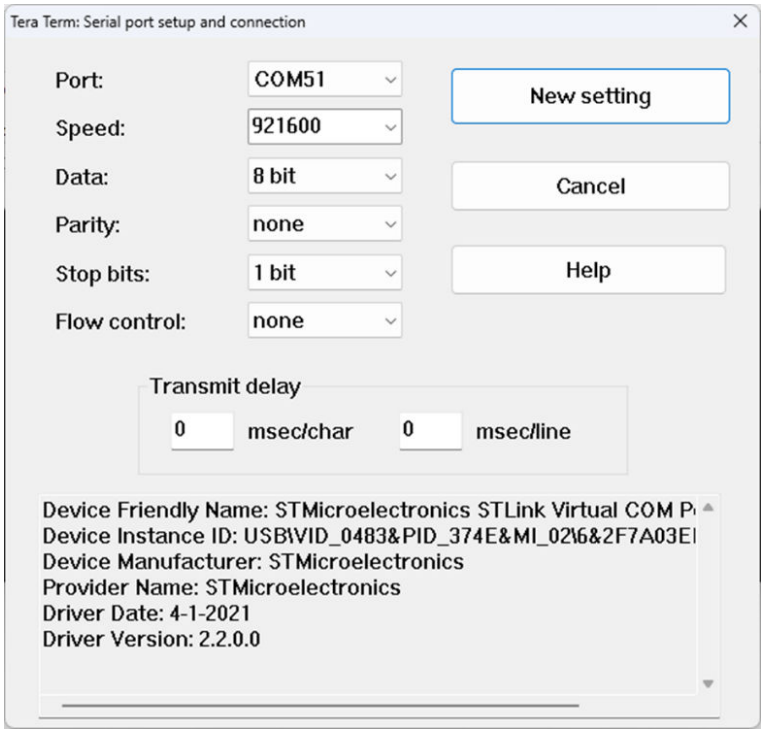


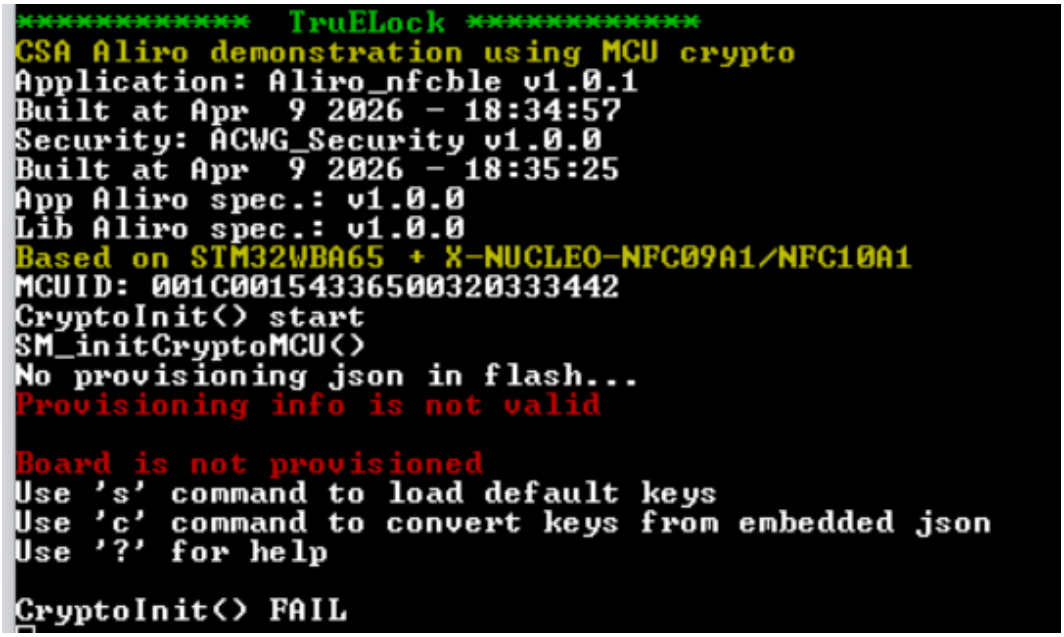
Figure 10. **Tera Term: serial port parameters**



5.2 First provisioning

At first, run the reader that doesn't have a provisioning.

Figure 11. Startup log trace



You can choose between two different ways to provision your reader:

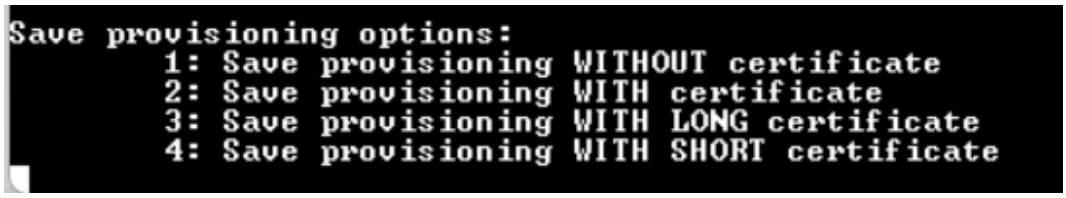
- Load default provisioning and then update the keys by serial interface command.
- Upload the json file on MCU memory through CubeProgrammer.

5.2.1 Use default provisioning

Follow the serial interface instruction to load the default keys.

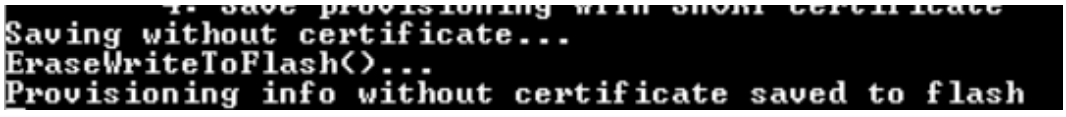
Step 1. Type 's' + Enter on the terminal to show the available default keys to save. Then, type from '1' to '4' + Enter to save the keys selected.

Figure 12. Provisioning choices



Step 2. After the choice, the following message is printed.

Figure 13. Log message indicating that the choice has been saved



Step 3. Restart the reader by typing ‘r’ + Enter.

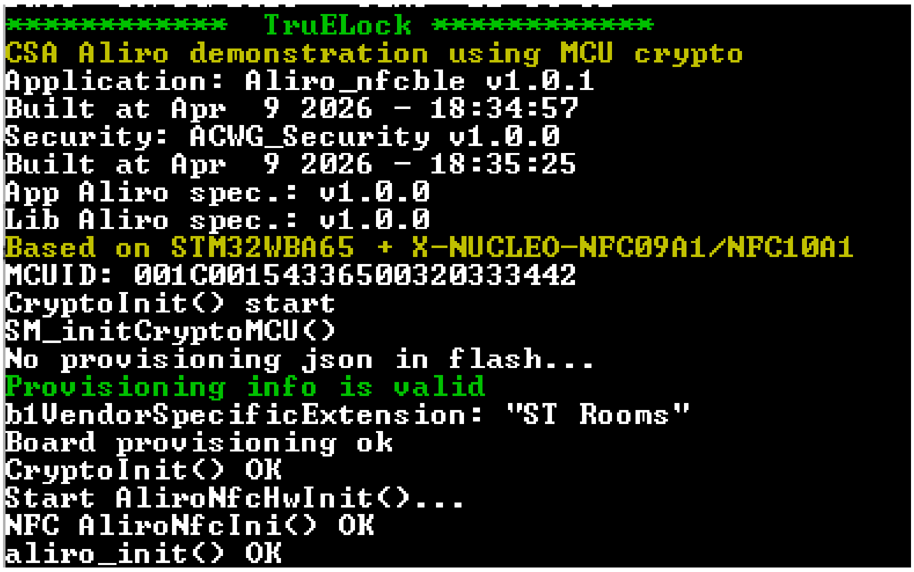


Figure 14. Startup log trace with valid provisioning

Step 4. To verify the loaded provisioning, type ‘p’ + Enter to view it.

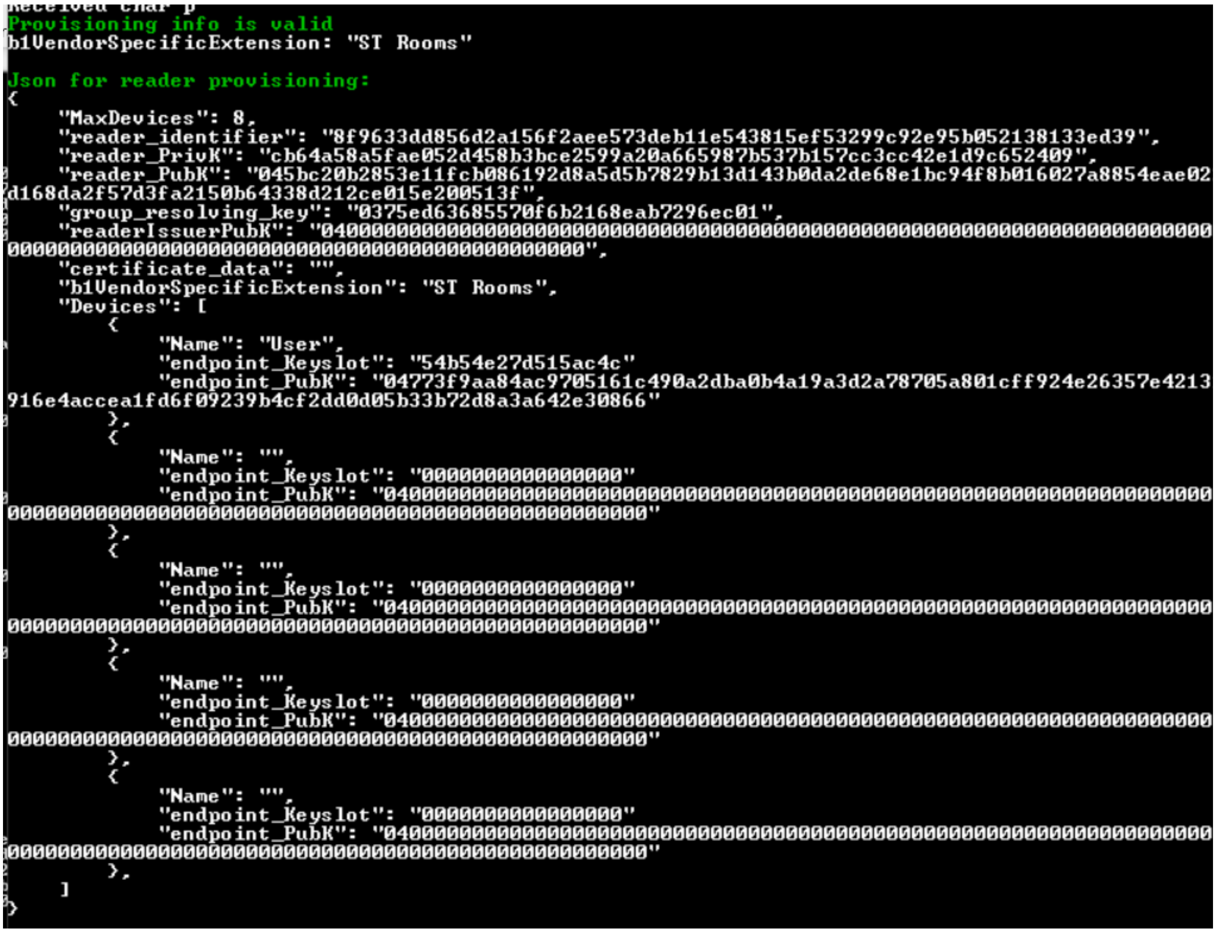


Figure 15. Stored provisioning info

5.2.2 Update keys by serial interface commands

It is possible to update the provisioning keys by using the serial interface. The following commands are indeed for this purpose:

- **setRDiden-**{reader ID + reader subID}.
- **setRDprvk-**{reader private key}.
- **setRDpubk-**{reader public key}.
- **setRDgrrk-**{reader group resolving key}.
- **setCrlspk-**{credential issuer public Key}.
- **setDEVpbk-**{device access public Key}.
- **setAccElm-**{access element name for step-up session}.

Each of these commands is followed by the value, typically expressed as a continuous hexadecimal string without spaces. For example, the reader identifier may be set with a command such as:

setRDiden-8f9633dd856d2a156f2aee573deb11e543815ef53299c92e95b052138133ed39

Public and private keys may be:

setRDprvk-cb64a58a5fae052d458b3bce2599a20a665987b537b157cc3cc42e1d9c652409

setRDpubk-045bc20b2853e11fcb086192d8a5d5b7829b13d143b0da2de68e1bc94f8b016027a8854eae0226d4205e8f710d168da2f57d3fa2150b64338d212ce015e200513f

Access credential key may be:

setDEVpbk-04773f9aa84ac9705161c490a2dba0b4a19a3d2a78705a801cff924e26357e4213191db83789a25916e4accea1fd6f09239b4cf2dd0d05b33b72d8a3a642e30866

Important: If a single key is updated without adjusting the other linked keys accordingly, NFC or Bluetooth® LE transactions may fail. After changing keys, the board is typically restarted so that the new configuration is fully applied.

5.2.3 Upload provisioning json file on flash MCU

To perform this procedure, you must:

1. Generate a provisioning JSON file and change the file extension to .bin.
2. Upload the generated .bin file to the MCU's flash using STM32CubeProgrammer.

5.2.3.1 Generate the provisioning file

To generate a provisioning JSON file, you need to create a formatted JSON as shown in the following example, filling the fields with your own keys and parameters.

```
{
      "reader_identifier": "8f9633dd856d2a156f2aee573deb11e543815ef53299c92e95b052138133ed39"
,
      "reader_PrivK": "cb64a58a5fae052d458b3bce2599a20a665987b537b157cc3cc42e1d9c652409",
      "reader_PubK": "045bc20b2853e11fcb086192d8a5d5b7829b13d143b0da2de68e1bc94f8b016027a8854
eae0226d4205e8f710d168da2f57d3fa2150b64338d212ce015e200513f",
      "group_resolving_key": "00000000000000000000000000000000",
      "certificate_data": "",
      "readerIssuerPubK": "",
      "b1VendorSpecificExtension": "ST Rooms",
      "Devices": [
            {
                  "Name": "Alice",
                  "endpoint_PubK": "04773f9aa84ac9705161c490a2dba0b4a19a3d2a78705a801cff924e2
6357e4213191db83789a25916e4accea1fd6f09239b4cf2dd0d05b33b72d8a3a642e30866",
            },
            {
                  "Name": "Dave",
                  "endpoint_PubK": "04773f9aa84ac9705161c490a2dba0b4a19a3d2a78705a801cff924e2
6357e4213191db83789a25916e4accea1fd6f09239b4cf2dd0d05b33b72d8a3a642e30866",
            },
            {
                  "Name": "Alex",
                  "endpoint_PubK": "04773f9aa84ac9705161c490a2dba0b4a19a3d2a78705a801cff924e2
6357e4213191db83789a25916e4accea1fd6f09239b4cf2dd0d05b33b72d8a3a642e30866",
            }
```

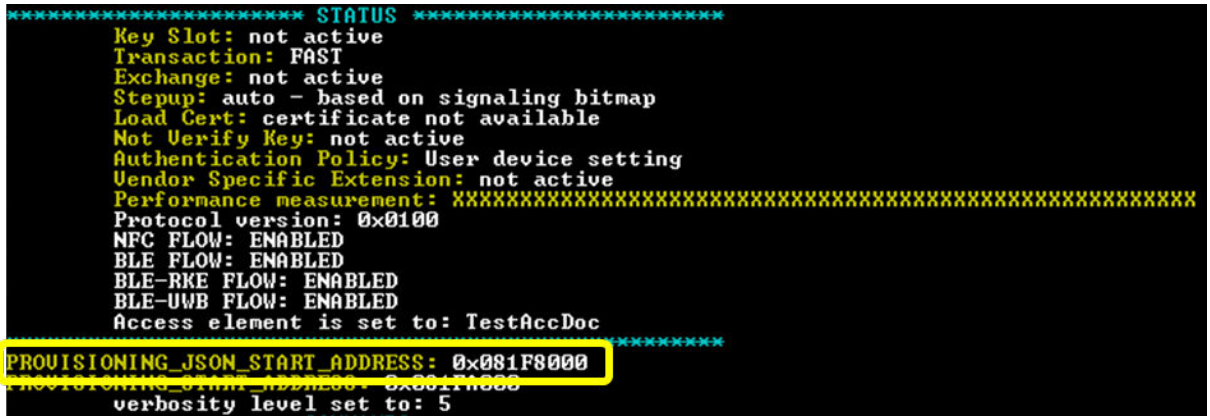
```
        ]
}
```

After that, the file extension needs to be changed from .json to .bin.

5.2.3.2 *Load provisioning file on MCU*

The first step to upload the provisioning file to the MCU's flash is to retrieve the address where the file should be uploaded. To do this, we use the serial interfaces '?' + Enter command.
PROVISIONING_JSON_START_ADDRESS tells us what we need.

Figure 16. Memory starting address where the JSON file is to be saved

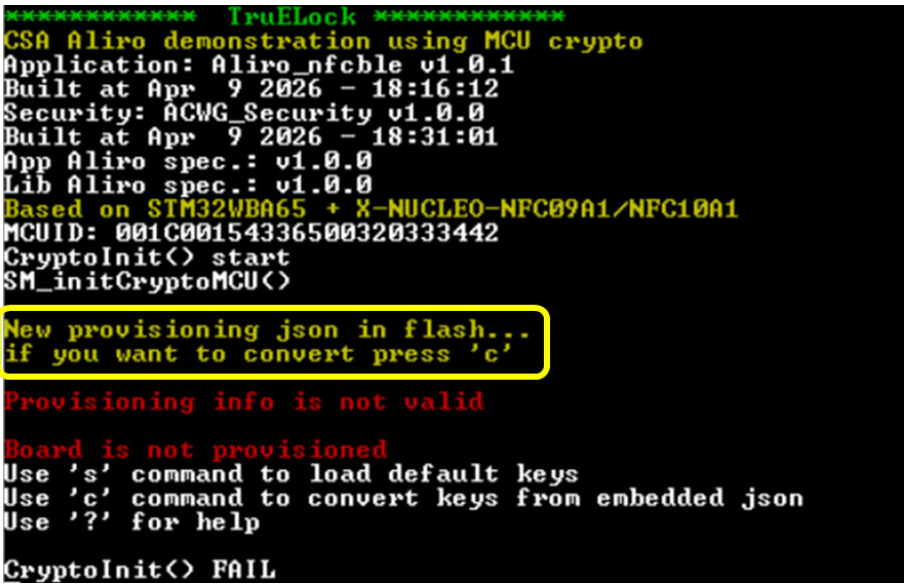


Now, we can move to STM32CubeProgrammer to load the .bin file on the flash.
Once STM32CubeProgrammer is open, you can proceed with the steps listed below:

1. Connect the device.
2. Move to the Erasing & Programming tab.
3. In the Erase flash memory tab, scroll down until you find the recovered address.
4. Check the sector corresponding to the address.
5. Press Erase selected sector and click OK.
6. Move to Download and click Browse to select the previously generated .bin file.
7. In the Start address field, enter the destination address.
8. Finally, press Start Programming and click OK until the end.
9. Now you can disconnect the device and reset the board.

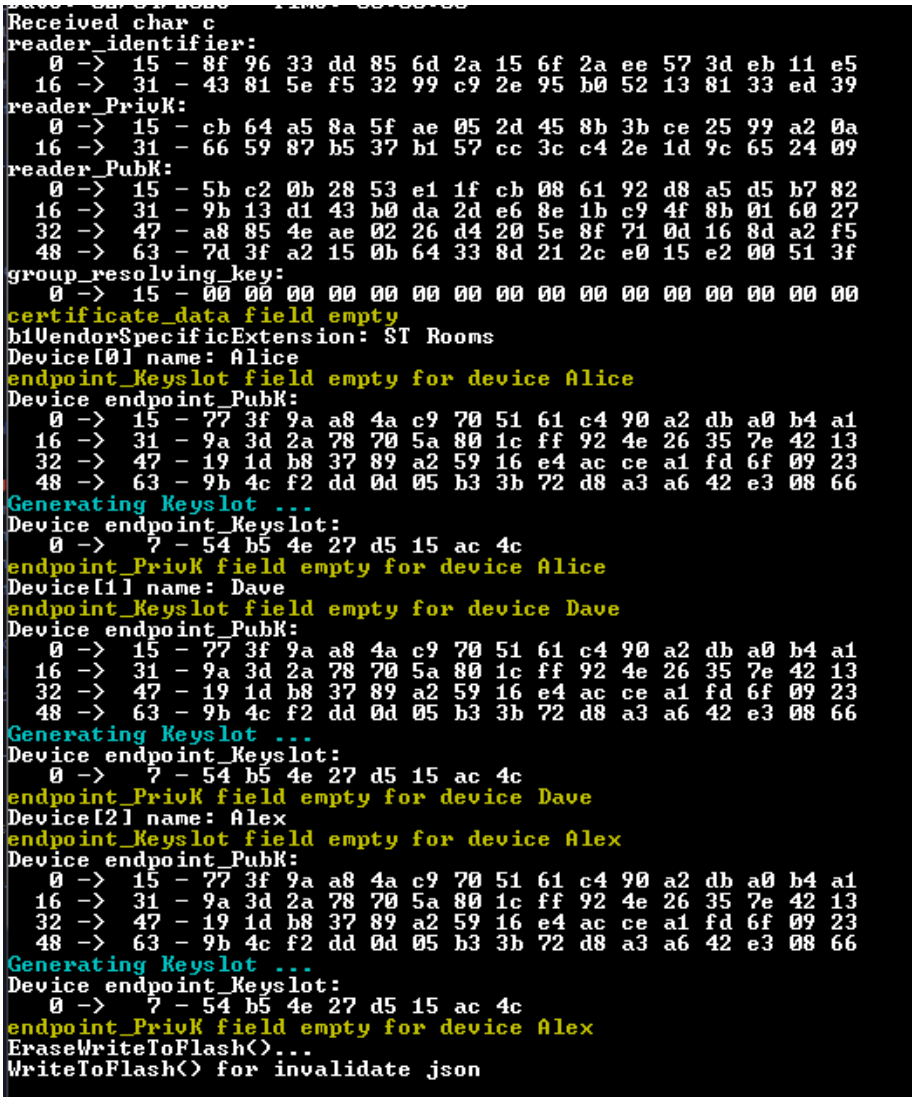
After the reboot, you are notified that a new json provisioning has been uploaded to flash, as shown in the following figure.

Figure 17. Startup with stored JSON file



At this point, you can activate the provisioning contained in the file using the serial interface command 'c' + Enter. At the end of the processing, if there are no errors, a screen like the one in the following figure is displayed.

Figure 18. Report after loading provisioning JSON file



You can reboot the board with the 'r' + Enter command to verify that the provisioning is valid upon reboot and then use the 'p' + Enter command to verify that the keys entered are correct.

5.3 Serial Command-line Interface

The ProcessUartInput function handles all commands received through the UART interface. It provides runtime configuration capabilities, system diagnostics, provisioning management, and debugging tools for the Aliro firmware.

The parser recognizes:

- Single-character commands.
- Extended multi-byte string commands.

5.3.1 Single-character commands

These commands consist of a single byte and trigger an immediate action.

System status summary and list of commands - ?

Typing '?' + Enter displays a comprehensive status dump that includes:

- NFC configuration (Exchange, StepUp, LoadCert, Fast mode).
- Keyslot activation status.
- Transaction mode (STANDARD / FAST).
- Provisioning information.
- Bluetooth® LE and UWB feature status.
- Current verbosity level.
- Complete list of supported commands.

Figure 19. Report showing system status and commands list

```
********************* STATUS *********************
        Key Slot: not active
        Transaction: FAST
        Exchange: not active
        Stepup: auto - based on signaling bitmap
        Load Cert: certificate not available
        Not Verify Key: not active
        Authentication Policy: User device setting
        Vendor Specific Extension: not active
        Performance measurement: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        Protocol version: 0x0100
        NFC FLOW: ENABLED
        BLE FLOW: ENABLED
        BLE-RKE FLOW: ENABLED
        BLE-UWB FLOW: ENABLED
        Access element is set to: TestAccDoc
*************************************************
PROVISIONING_JSON_START_ADDRESS: 0x081F8000
PROVISIONING_START_ADDRESS: 0x081FA000
        verbosity level set to: 5
********************* COMMANDS *********************
        0-6 Set verbosity level
        ? - System status and commands list
        a - Set Authentication Policy
        b - Toggle Vendor Specific Extension
        f - Toggle Fast transaction
        k - Toggle Key Slot activation
        e - Toggle EXCHANGE
        l - Set LOAD_CERT
        u - Toggle STEPUP
        v - Toggle Not Verify Key
        d - Ranging Suspend/Resume Action (not active work in progress)
        p - Provisioning status
        s - Save default provisioning info
        c - Convert provisioning info from json
        w - NFC wakeup mode (LPCD)
        n - Switch NFC only / NFC+BLE / NFC+BLE+UWB
        z - Save last access credential PuBK
        'date:' - Set system date
        'time:' - Set system time
        'settime-' - Set system date and time
        'setRDiden-' - Set reader group identifier and sub identifier
        'setRDprvk-' - Set reader private key
        'setRDpubk-' - Set reader public key
        'setRDgrrk-' - Set reader group resolving key
        'setCrIspk-' - Set Credential issuer public key
        'setDEVpbk-' - Set reader access credential public key
        'setAccElm-' - Set access element name for stepup (session)
        r - system restart
        ! - system pause
*************************************************
```

The following tables detail same of the allowed commands. To execute a command, type the related symbols and press Enter.

Table 1. Verbosity levels

Command	Level	Description
0	VERB_NULL	No logging
1	VERB_ALWAYS	Always log
2	VERB_CRITICAL	Critical errors only
3	VERB_ERROR	Errors only
4	VERB_NORMAL	Standard information
5	VERB_VERBOSE	Verbose debug output (default)
6	VERB_INFO	Full informational output

Table 2. Flow commands

Command	Function
a	Configure authentication policy
b	Toggle vendor-specific extension
f	Toggle FAST transaction mode
k	Toggle key-slot activation
e	Configure EXCHANGE mode (READ/WRITE/SET/ALL/AUTO)
l	Configure LOAD_CERT mode
u	Modify STEPUP behavior (disabled / auto / always)
v	Toggle not-verify-Key flag
d	Ranging suspend/resume action

Table 3. Provisioning commands

Command	Function
p	Display provisioning JSON contents
s	Save provisioning data (various modes)
c	Convert provisioning from JSON
z	Show and optionally save the last transaction access credential public key

Table 4. System commands

Command	Function
n	Switch NFC only / NFC+BLE / NFC+BLE+UWB
w	NFC wake-up mode (LPCD)
r	System restart
!	Pause/Resume the Aliro processing loop (UART still active)

5.3.2 Extended (string-based) commands

Extended commands consist of a keyword followed by parameters.

Time & date management (if USE_RTC enabled)

Table 5. Date and time setting commands

Command	Format	Function
date:	date: DD/MM/YYYY	Set system date
time:	time: HH:MM[:SS]	Set system time
settime-	settime- YYYY/MM/DD HH:MM:SS	Set date and time

The parser includes validation of date/time formats and calendar correctness.

Provisioning and key management

All commands follow this form:
set<COMMAND>-<HEXDATA>
Data must be provided as **hex ASCII**.

Table 6. Provisioning management commands

Command	Data size	Operation
setRDpubk-	64 bytes (prefixed by 04)	Save reader public key
setRDprvk-	32 bytes	Save reader private key
setRDiden-	32 bytes	Save reader identifier
setRDgrrk-	16 bytes	Save group resolving key
setCrIspk-	64 bytes (prefixed by 04)	Save credential issuer public key
setDEVpbk-	64 bytes (prefixed by 04)	Save access credential public key
setAccElm-	ASCII string	Set access element name for stepup

Echo service

echo-<string> prints the received string and the current date/time.

Hex translator mode

If the received buffer has one of the following length, the function prints a formatted **0xHH** hex table representation of the data:

- 32 bytes.
- 64 bytes.
- 128 bytes.
- 130 bytes.

Revision history

Table 7. **Document revision history**

Date	Revision	Changes
06-May-2026	1	Initial release.

Contents

List of tables

List of figures

IMPORTANT NOTICE – READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice.

In the event of any conflict between the provisions of this document and the provisions of any contractual arrangement in force between the purchasers and ST, the provisions of such contractual arrangement shall prevail.

The purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgment.

The purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of the purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

If the purchasers identify an ST product that meets their functional and performance requirements but that is not designated for the purchasers' market segment, the purchasers shall contact ST for more information.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.