Getting started with the IOTA Distributed Ledger Technology software expansion for STM32Cube

Introduction

The X-CUBE-IOTA1 expansion software package for STM32Cube runs on the STM32 and includes middleware to enable the IOTA Distributed Ledger Technology (DLT) functions.

The IOTA DLT is a transaction settlement and data transfer layer for the Internet of Things (IoT). IOTA allows people and machines to transfer money and/or data without any transaction fees in a trustless, permissionless and decentralized environment. This technology even makes micro-payments possible without the need of a trusted intermediary of any kind.

The expansion is built on STM32Cube software technology to ease portability across different STM32microcontrollers.

The software comes with sample implementations to use the IOTA middleware on a NUCLEO-F429ZI or NUCLEO-F746ZG development board.

RELATED LINKS

https://www.iota.org/get-started/what-is-iota

https://docs.iota.org/docs/getting-started/0.1/introduction/what-is-iota

https://www.boazbarak.org/cs127/Projects/iota.pdf

1 Acronyms and abbreviations

Table 1. List of acronyms

Acronym	Description
DLT	Distributed ledger technology
IDE	Integrated development environment
IoT	Internet of things
PoW	Proof-of-Work
SHA-3	Secure Hash algorithm 3

2 What is STM32Cube?

2.1 STM32Cube overview

STMCube™ is an STMicroelectronics initiative aimed at making life easier for the developer by reducing development effort, time and cost for the STM32 range of products.

Version 1.x includes:

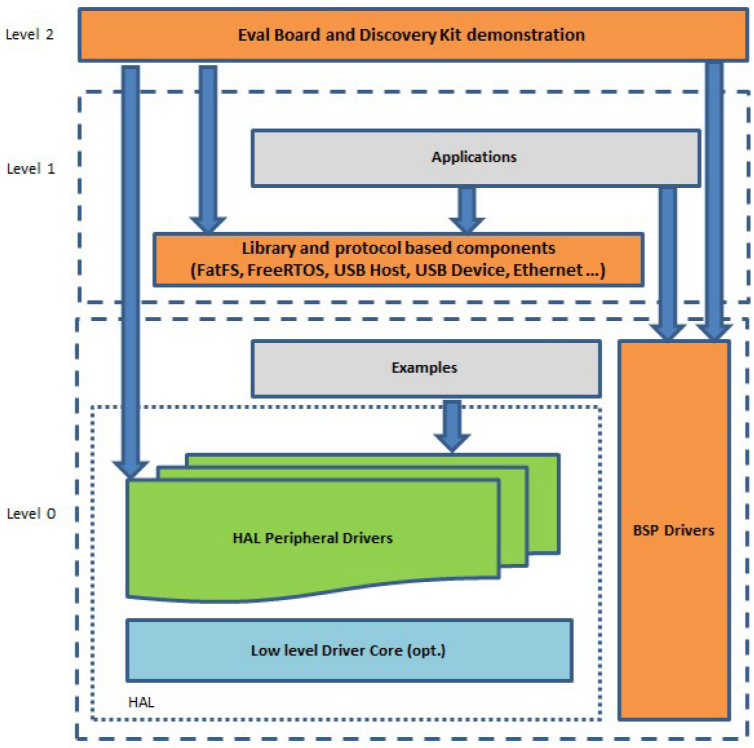
- The STM32CubeMX graphical software configuration tool to generate C initialization code using graphical wizards.
- A comprehensive embedded software platform for each series (e.g., STM32CubeF4 for the STM32F4 series)
  - the embedded STM32Cube HAL abstraction layer software which maximizes portability across the STM32 portfolio
  - a consistent set of middleware components, including RTOS, USB, TCP/IP and graphics
  - all the embedded software utilities come with a full set of examples

Information regarding STM32Cube is available on www.st.com at: http://www.st.com/stm32cube.

2.2 STM32Cube architecture

The STM32Cube firmware solution is based on three independent levels that freely interact with each other, as shown below:

Figure 1. Firmware architecture



Level 0 is divided into three sub-layers:

- The Board Support Package (BSP) layer offers a set of board hardware APIs (audio codec, IO expander, touchscreen, SRAM driver, LCD drivers, etc.) based on modular architecture which can be rendered compatible with any hardware by simply running the low-level routines. The BSP has two parts:
  - component: the driver associated with the external device on the board (not the STM32); the component driver provides specific APIs to the BSP driver external components and can be ported to any other board.
  - BSP driver: links the component driver to a specific board and provides a set of user-friendly APIs. The naming rule of the APIs is BSP_FUNCT_Action(): ex. BSP_LED_Init(), BSP_LED_On()
- The Hardware Abstraction Layer (HAL) provides the low level drivers and the hardware interfacing methods to interact with the upper layers (application, libraries and stacks). It provides generic, multi-instance and function-oriented APIs which render user applications unnecessary by providing ready to use processes. For example, it provides APIs for the communication peripherals (I²S, UART, etc.) for initialization and configuration, data transfer management based on polling, interrupts or DMA processes, and management of any communication errors. There are two types of HAL driver APIs:
  - generic APIs which provide common and generic functions to the entire STM32 series
  - extension APIs which provide specific, customized functions for a particualr family or a certain part number
- Basic peripheral usage examples: this layer includes the examples built for the STM32 peripheral using the HAL and BSP resources only.

Level 1 is divided into two sub-layers:

- Middleware components: a set of libraries covering USB host and device libraries STemWin, FreeRTOS, FatFS, LwIP, and PolarSSL. Horizontal interaction between layer components is handled directly by calling the feature APIs, while vertical interaction with the low level drivers is managed through specific callbacks and static macros implemented in the library system call interface. For example, the FatFs accesses the microSD drive or the USB mass storage class via the disk I/O driver.
- Middleware examples (or applications) for individual components as well as integration examples across several middleware components are provided.

Level 2 is a single layer providing a global, real-time and graphical demonstration based on the middleware service layer, the low-level abstraction layer and basic peripheral usage applications involving board functions

3 X-CUBE-IOTA1 software expansion for STM32Cube

3.1 Overview

The X-CUBE-IOTA1 software package expands STM32Cube functionality with the following key features:

- Complete middleware to build IOTA Distributed Ledger Technology (DLT) applications for STM32-based boards
- Easy portability across different MCU families, thanks to STM32Cube
- Examples to help understand how to develop an IOTA DLT application
- Includes the STM32CubeMX project file (.ioc) for the graphical visualization of the STM32 microcontroller pins, peripherals and middleware configuration
- Free, user-friendly license terms

The software expansion provides the middleware to enable the IOTA DLT on an STM32 microcontroller.

The IOTA DLT is a transaction settlement and data transfer layer for the Internet of Things (IoT). IOTA allows people and machines to transfer money and/or data without any transaction fees in a trustless, permissionless and decentralized environment. This technology even makes micro-payments possible without the need of a trusted intermediary of any kind.

3.1.1 IOTA

Distributed Ledger Technologies (DLTs) are built on a node network which maintains a distributed ledger, which is a cryptographically secured, distributed database to record transactions. Nodes issue transactions through a consensus protocol.

IOTA is a distributed ledger technology specifically designed for IoT.

The IOTA distributed ledger is called the *tangle* and is created by the transactions issued by the nodes in the IOTA network.

To publish a transaction in the tangle, a node has to:

1. validate two unapproved transactions called tips
2. create and sign the new transaction
3. perform sufficient Proof-of-Work
4. broadcast the new transaction to the IOTA network

The transaction is attached to the tangle together with two references pointing to the validated transactions.

This structure can be modeled as a directed acyclic graph, where the vertices represent single transactions and the edges represent references among pairs of transactions.

A genesis transaction is at the tangle root and includes all the available IOTA tokens, called iotas.

IOTA uses a rather unconventional implementation approach based on trinary representation: every element in IOTA is described using trits = -1, 0, 1 instead of bits, and trytes of 3 trits instead of bytes. A tryte is represented as an integer from -13 to 13, encoded using letters (A-Z) and number 9.

The IOTA network includes full nodes and light nodes. A full node is connected to peers in the network and stores a copy of the tangle. A light node is a device with a seed to be used to create addresses and signatures.

The light node creates and signs transactions and sends them to the full node so that the network can validate and store them. Withdrawing transactions must contain a valid signature. When a transaction is considered valid, the full node adds it to its ledger, updates the balances of the affected addresses and broadcasts the transaction to its neighbors.

3.1.2 Transactions and bundles

An IOTA transaction is a 2673 tryte-long string that can withdraw/deposit iotas or send data.

Transactions referring to the same transfer of iotas are packed together in a bundle.

Table 2. Structure of an IOTA transaction

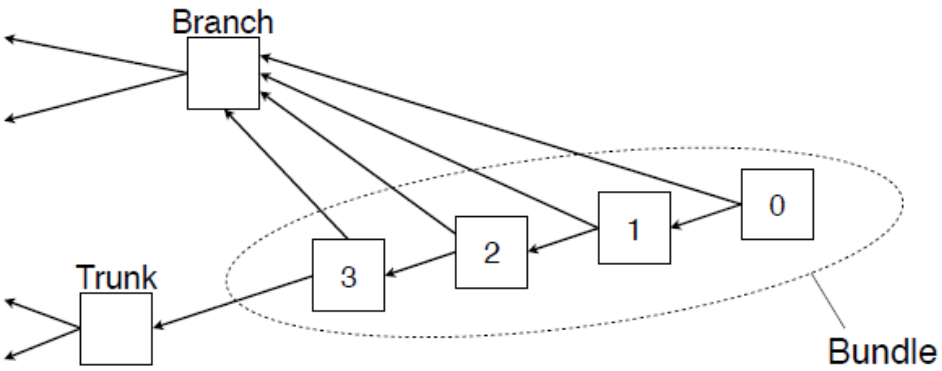
Size (tryte)	Field
2187	Signature fragment
81	Address
27	Value
27	Obsolete tag
9	Timestamp
9	Current index
9	Last index
81	Bundle Hash
81	Trunk Hash
81	Branch Hash
27	Tag
9	Attachment timestamp
9	A.T. lower bound
9	A.T. upper bound
27	Nonce

Referring to the table above:

- signature fragment: contains the transaction signature
- bundle essence:
  - address: a deposit or a withdrawal address
  - value: the amount of iota transferred during the transaction
  - obsolete tag: arbitrary user-defined value
  - timestamp: the timestamp of the bundle essence creation time
  - current index: the index of the transaction in the bundle
  - last index: the total number of transactions in the bundle
- bundle Hash: digest of the bundle essences of the bundle transactions concatenated together
- trunk and branch Hash: respectively, the digests of the first and second transaction referenced by the current transaction
- tag: a user-defined value that can later be used to retrieve the transaction in the tangle
- attachment timestamp: the timestamp of the PoW start time
- lower bound and upper bound: fields for future use, usually set to the minimum and maximum value
- nonce: contains the PoW nonce of the transaction

Transactions belonging to the same bundle are attached to the tangle in a chain as shown in the figure below. The first transaction in the bundle is the tail, while the last one is the head.

Figure 2. **Structure of a bundle**



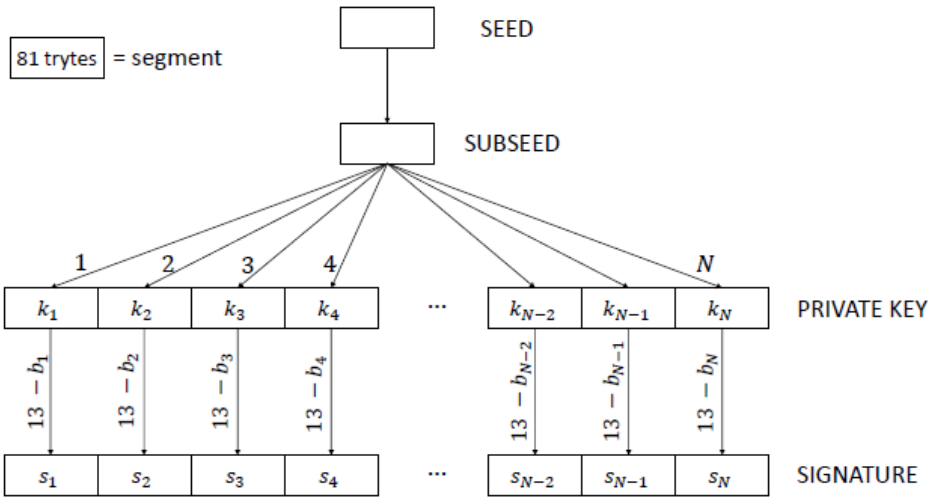
3.1.3 Account and signatures

In IOTA, each node corresponds to one or more accounts. An account is identified with a seed which is an 81 tryte-long string that must be kept secure by the node. From a seed, several pairs of cryptographic keys can be generated depending on an index n and a security level l = 1, 2, 3. Each public key corresponds to an IOTA address which has its own balance in iotas. The sum of the balances of all the addresses of an account is the balance of the account.

Transaction withdrawing iotas are digitally signed. The signature scheme used in IOTA is based on the Winternitz One-Time Signature scheme (W-OTS). A higher security level corresponds to a longer signature and therefore 1, 2, or 3 transactions are needed to store the entire signature. On the contrary, transactions that do not withdraw iotas do not need a signature and the 2187 trytes of the signature can be allocated to send data.

Figure 3. **IOTA signing process**

k_i denotes the i-the key segment, generated by hashing the subseed i times. b_i denotes the i-th normalized bundle hash tryte value. The key segment k_i is hashed 13-b_i times to obtain the corresponding signature segment.



3.1.4 Proof-of-Work

Each transaction in a bundle must include a Proof-of-Work (PoW) nonce which is found using trial and error.

A starting value for the nonce is set and the transaction is hashed.

If the digest ends in a certain amount of 0 trits, indicated as MWM, the nonce is considered valid. Otherwise, the value for the nonce is incremented and the transaction is hashed again.

3.1.5 Hashing

In the IOTA signing process, the Kerl hash function is used.

Kerl converts trinary and binary representations and uses the standard Keccak hash function.

For the PoW nonce computation, the Curl hash function is used.

3.2 Architecture

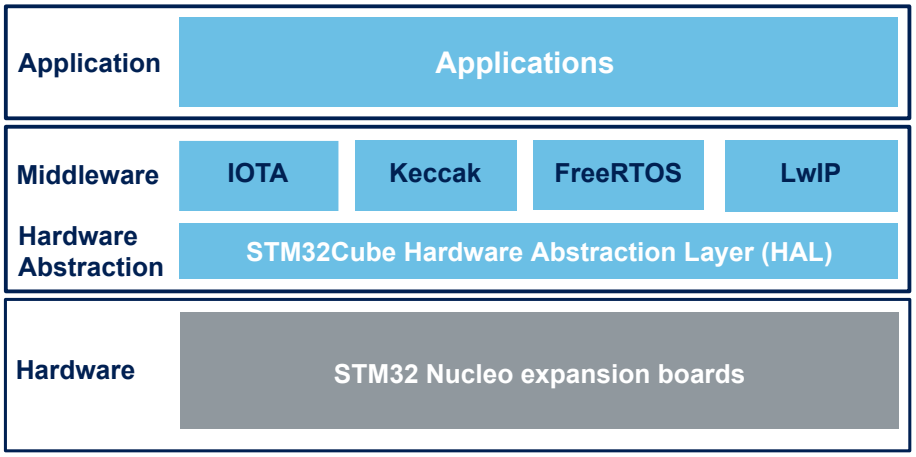
This STM32Cube expansion enables development of applications accessing and using the IOTA DLT middleware.

It is based on the STM32CubeHAL hardware abstraction layer for the STM32 microcontroller and extends STM32Cube with a specific board support package (BSP) for the microphone expansion board and middleware components for audio processing and USB communication with a PC.

The software layers used by the application software to access and use the microphone expansion board are:

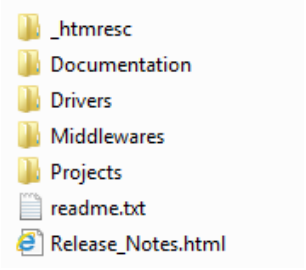
- STM32Cube HAL layer: provides a generic, multi-instance set of APIs to interact with the upper layers (the application, libraries and stacks). It consists of generic and extension APIs based on a common architecture which allows other layers like the middleware layer to function without specific Microcontroller Unit (MCU) hardware configurations. This structure improves library code reusability and guarantees easy device portability.
- Board Support Package (BSP) layer: is a set of APIs which provides a programming interface for certain board specific peripherals (LED, user button etc.). This interface also helps in identifying the specific board version and provides support for initializing required MCU peripherals and reading data.

Figure 4. X-CUBE-IOTA1 software architecture



3.3 Folder structure

Figure 5. X-CUBE-IOTA1 folder structure



The following folders are included in the software package:

- **Documentation**: contains a compiled HTML file generated from the source code and detailed documentation of the software components and APIs
- **Drivers**: contains the HAL drivers and the board-specific drivers for supported board and hardware platforms, including those for the on-board components and the CMSIS vendor-independent hardware abstraction layer for the ARM® Cortex®-M processor series

- **Middlewares**: contains libraries and protocols related to host software and applications enabling the IOTA DLT on the STM32 microcontrollers
- **Projects**: contains sample applications for the supported STM32-based platforms (NUCLEO-F746ZG, NUCLEO-F429ZI), with three development environments, IAR Embedded Workbench for ARM (EWARM), RealView Microcontroller Development Kit (MDK-ARM) and System Workbench for STM32 (SW4STM32)

3.4 How to write applications

This section describes how to write an IOTA application for an STM32-based board, issuing zero-value and transfer transactions and publishing them on the tangle.

3.4.1 Data types

In IOTA, a tryte string denotes the char-encoded tryte values. In this implementation, trits and trytes are represented as 8-bit integers. To save storage space, sometimes tryte values are stored as byte values (243 trits = 81 trytes → 48 bytes).

Table 3. Trits, trytes and decimal relationship

Decimal	Tryte	Tryte (enc)	Decimal	Tryte	Tryte (enc)
0	0 0 0	9			
1	1 0 0	A	-1	-1 0 0	Z
2	-1 1 0	B	-2	1 -1 0	Y
3	0 1 0	C	-3	0 -1 0	X
4	1 1 0	D	-4	-1 -1 0	W
5	-1 -1 1	E	-5	1 1 -1	V
6	0 -1 1	F	-6	0 1 -1	U
7	1 -1 1	G	-7	-1 1 -1	T
8	-1 0 1	H	-8	1 0 -1	S
9	0 0 1	I	-9	0 0 -1	R
10	1 0 1	J	-10	-1 0 -1	Q
11	-1 1 1	K	-11	1 -1 -1	P
12	0 1 1	L	-12	0 -1 -1	O
13	1 1 1	M	-13	-1 -1 -1	N

3.4.2 Initialization

Every application must perform some basic initialization steps to configure and set up the STM32 Nucleo board with the software stack for correct operation.

Step 1. Initialize the STM32Cube HAL library to correctly configure the necessary hardware components.

The `HAL_Init();` API initializes the HAL library and configures Flash prefetch, Flash preread and Buffer cache. It also configures the time base source, vectored interrupt controller and low-level hardware.

Step 2. Configure STM32 Nucleo on-board peripherals and hardware before using them (if needed) by the APIs below.

- `BSP_LED_Init(Led_TypeDefLed);` configures the STM32 Nucleo LED
- `BSP_PB_Init(Button_TypeDef Button, ButtonMode_TypeDefButton_Mode);` configures the user button in GPIO mode or in external interrupt (EXTI) mode
- `BSP_JOY_Init();` configures the joystick if the board is equipped with one

Step 3. Initialize the account manager with the user's seed (SEED_LEN = 81) to compute the corresponding addresses and valid signatures, by using the `account_initialize (char seed[SEED_LEN]);` API.

It initializes the `acc_manager_t` object associated to the device with the user's seed converted to bytes.

3.4.3 Address computation

A Light Node cannot store every possible address associated to its seed, but it computes them when needed. An address is characterized by a security level (1, 2 or 3) and an index (a non-negative integer).

The following functions allow the user to compute the desired address or set of addresses:

- `get_address_bytes(uint32_t idx, uint8_t security, uint8_t *add_bytes);`
- `get_address(uint32_t idx, uint8_t security, char *address);`

These APIs compute the address with security level *security* and index *idx* and stores it in byte or tryte format.

- `get_addresses_bytes(uint32_t qty, uint32_t idx0, uint8_t sec, uint8_t* addresses);`
- `get_addresses(uint32_t qty, uint32_t idx0, uint8_t sec, char* addresses);`

These APIs compute *qty* addresses with security level *sec*, from the one indexed *idx0* onwards in byte or tryte format.

3.4.4 Bundle management

Transactions must be packed together in a bundle to be published on the tangle and are indexed from 0 to last_index.

A `BUNDLE_CTX` object contains the bundle essences of its transactions converted into bytes and the corresponding bundle hash.

The following APIs are used for bundle management:

- `bundle_initialize(BUNDLE_CTX *ctx, uint32_t last_index);` initializes a bundle with the specified dimensions
- `bundle_add_tx(BUNDLE_CTX *ctx, int64_t value, char* tag, int64_t timestamp);` encodes the input parameters in bytes and adds them to the bundle object
- `bundle_finalize(BUNDLE_CTX *ctx);` finalizes a bundle by computing its hash.

Note: During the signing process, the normalized bundle hash is actually signed. For security reasons, the normalized bundle hash cannot contain the M tryte. If this happens, the obsolete tag of the last transaction in the bundle is incremented and the bundle hash is computed again.

3.4.5 Signing

Input transactions require a valid signature, which depends on the normalized bundle hash. A signature is made of *l* fragments, where *l* is the security level of the spending address and each fragment is 2187 tryte long.

The following APIs are used for the signing process.

- `signer_initialize(SIGNER_CTX *ctx, uint32_t address_idx, int8_t security, const tryte_t *normalized_hash);` initializes the signer with the address index and security level required to compute the correct private key, and with the normalized bundle hash to be signed
- `sign_next_fragment(SIGNER_CTX *ctx, uint8_t *signature_bytes);` produces the next signature fragment (called *l* times to generate the required signature fragments)

3.4.6 Proof-of-Work

The Proof-of-Work is the process used to find a valid nonce necessary to finalize a transaction.

The `diver_pow(char *tx_chars, char *nonce);` API computes a valid nonce for the provided transaction.

3.4.7 Full Node API

A Light Node must be able to communicate with a Full Node to obtain the required information to correctly create and finalize transactions. The communication between nodes is performed through an HTTP API provided by the standard IOTA protocol.

In this implementation, the following functions can be used to make requests to a Full Node:

- `iota_api_get_node_info();` outputs the information about the Full Node
- `iota_api_get_balance(char* address);` returns the balance of the provided address
- `iota_api_get_balances(char* addresses, size_t qty, int64_t *balances);` stores the balances of the provided addresses in the provided array *balances*
- `iota_api_was_address_spent_from(char* address);` returns the state of the provided address (true if it was already used to spend, false otherwise)
- `iota_api_were_addresses_spent_from(char* addresses, size_t qty, bool * states);` stores the states of the provided addresses (true if the address was already used to spend, false otherwise) in the provided array *states*
- `iota_api_get_transactions_to_approve(char* trunk, char* branch);` stores the hashes of the transactions to approve in the provided locations
- `iota_api_attach_to_tangle(char* trytes);` takes as input the non-finalized transaction, demands the PoW to the Full Node and updates the trytes with a valid nonce. It also sets the attachment timestamps
- `iota_api_broadcast_trasaction(char* tx);` sends the finalized transaction to the Full Node for broadcast
- `iota_api_get_latest_milestone(char* milestone);` stores the latest milestone received by the Full Node in the provided location
- `iota_api_get_inclusion_state(char* tx_hash);` returns the confirmation state of the transaction with the provided hash (true if the transaction was confirmed, false otherwise)

3.4.8 Creating transactions and bundles

3.4.8.1 *TX_IN and TX_OUT*

IOTA transactions can be divided in two groups, represented by two different structures:

- TX_IN: transactions spending at least 1 iota (require a signature)
- TX_OUT: transactions receiving at least 1 iota or zero-value (do not require a signature)

A TX_IN object contains:

- key_idx: the index of the spending address
- security: the security level of the spending address
- balance: the balance of the spending address
- tag: a tag to retrieve the transaction in the tangle

A TX_OUT object contains:

- message: a byte-encoded message to be inserted in the transaction in place of the signature
- address: the byte-encoded address spending iotas or sending the zero-value transaction
- value: the iotas being spent by the address (greater than or equal to 0)
- tag: a tag to retrieve the transaction in the tangle

3.4.8.2 *Finalization options*

In order to finalize a transaction, a bool array of options must be set:

- `options[0] = local_pow` → if true, PoW is performed locally, otherwise PoW is demanded to the Full Node
- `options[1] = broadcast` → if true, the finalized transaction is sent to the Full Node to be published on the tangle and broadcast to the other nodes, otherwise the transaction remains on the device and not published on the tangle
- `options[2] = reattach` → if true, once the bundle has been finalized and broadcast, the node checks regularly its confirmation state. After a certain amount of minutes, if the bundle has not been confirmed it is reattached.

3.4.8.3 *Sending data through zero-value transactions*

A zero-value transaction does not require a signature, hence the 2187 starting trytes of the transaction can be used to store tryte-encoded data.

Ascii-encoded data can be converted to trytes using the `ascii_to_chars` function.

In order to publish data on the tangle, the following steps must be performed:

Step 1. Define the required number of TX_OUT transactions to store the whole set of data (1 transaction can store up to 2187 trytes)

Step 2. Set the sender address and tag in the transaction objects

Step 3. Set the finalization options

Step 4. Call the `make_zero_value` function to finalize the transaction

Function `wallet_zero_value` in *wallet.c* manages the user inputs and creates a `zero_value` transaction.

The function in the example below has the same functionality without user interaction.

```
/** @brief This function publishes the provided message on the Tangle
  * @param tryte_address: the char-encoded trytes of the sender address
  * @param tryte_message: the char-encoded trytes of the message to be published
  * @param tag: the char-encoded trytes of the tag to be inserted in the transaction
  */
void publish_message(char tryte_address[LENGTH_ADDRESS],                    char try
te_message[LENGTH_SIGNATURE],
                     char tag[LENGTH_TAG])
{
    /* Transaction object */
    TX_OUT zero_value;
    chars_to_bytes(tryte_message, zero_value.message, LENGTH_SIGNATURE);
```

```
    chars_to_bytes(tryte_address, zero_value.address, LENGTH_ADDRESS);
    zero_value.value = 0;
    (void)memcpy(zero_value.tag, tag, LENGTH_TAG);

    /* Finalization options */
    bool options[3];
    options[0] = false; // Outsource the PoW
    options[1] = true;  // Broadcast the finalized transaction
    options[2] = false; // Do not wait for confirmation and reattach

    /* Use the API */
    make_zero_value(&zero_value, 1, options);
}
```

3.4.8.4 *Sending funds through transfer transactions*

To create an iota transfer follow the procedure below.

Step 1. Define and set the details of the TX_IN objects, namely the spending transactions

Step 2. Define and set the details of the TX_OUT objects, namely the receiving transactions

Step 3. Define and set the details of the remainder transaction (if needed)

Step 4. Set the finalization options

Step 5. Call the `make_transfer` function (passing NULL as remainder transaction if not used)

The `wallet_transfer()` function in the *wallet.c* file manages the user input and created a transfer bundle. The example below shows how a transfer can be performed without user interaction.

```
/** @brief This function transfers funds from user's address of index key_idx and
  *        security level sec to the provided receiving address
  * @param key_idx: the spending address' index
  * @param sec: the spending address' security level
  * @param spn_tag: the char-encoded trytes of the tag of spending transaction
  * @param tryte_address: the char-encoded trytes of the sender address
  * @param tryte_message: the char-encoded trytes of the message to be published
  * @param rec_tag: the char-encoded trytes of the tag of receiving transaction
  */
void send_funds(uint32_t key_idx, uint8_t sec, char spn_tag[LENGTH_TAG],
                char tryte_message[LENGTH_SIGNATURE],
                char tryte_address[LENGTH_ADDRESS],                int64_t funds,
                char tag[LENGTH_TAG])
{
    /* Spending */
    TX_IN tx_spn;
    tx_spn.key_idx = key_idx;
    tx_spn.security = sec;
char add[81];
get_address(txs_in.key_idx, tx_in.security, add);
    tx_spn.balance = iota_api_get_balance(add);
(void)memcpy(tx_spn.tag, spn_tag, LENGTH_TAG);

    /* Receiving */
    TX_OUT tx_rec;
    chars_to_bytes(tryte_message, tx_rec.message, LENGTH_SIGNATURE);
    chars_to_bytes(tryte_address, tx_rec.address, LENGTH_ADDRESS);
tx_rec.value = funds;
    (void)memcpy(tx_rec.tag, tag, LENGTH_TAG);

    /* Finalization options */
    bool options[3];
    options[0] = false; // Outsource the PoW
    options[1] = true;  // Broadcast the finalized transaction
    options[2] = false; // Do not wait for confirmation and reattach

    /* Use the API - Suppose there is no need for a remainder tx */
    Make_transfer(&tx_rec, 1, &tx_spn, 1, NULL, options);
```

3.5 IOTA-LightNode application description

The project files for the IOTA-LightNode application can be found in: *&BASE_DIR\Projects\STM32*-Nucleo \Applications\IOTA-LightNode*.

Any application folder contains a readme.txt file providing the user with all useful information on how to run the sample code on the STM32 Nucleo board.

In the IOTA-LightNode application, the STM32 Nucleo:

- checks the user's balance
- creates a zero-value transaction
- creates a iota transfer
- use other functions

3.5.1 Check balance

The Check balance functionality outputs:

- the balance associated with each security level computed by adding up the individual balances of the account addresses
- the total balance associated with the account

3.5.2 Create a zero-value transaction

The *Create zero-value transaction* functionality takes as inputs:

- the sender's address
- the data to be included in the transaction
- a tag to retrieve the transaction in the tangle
- the following options:
  - local PoW (or outsourced)
  - broadcast (or not)
  - wait for confirmation and reattach (or not)

The output is the finalized transaction.

3.5.3 Create a transfer bundle

The *Create transfer bundle* functionality takes as inputs:

- the receiver's details:
  - the address
  - the amount of transferred iotas
  - the data to be included in the transaction
  - a tag to retrieve the transaction in the tangle
- the sender's details:
  - the security level of the spending addresses
  - a tag to retrieve the transaction in the tangle
- the following options:
  - local PoW (or outsourced)
  - broadcast (or not)
  - wait for confirmation and reattach (or not)

The *Create transfer bundle* functionality checks if the addresses at the specified security level have enough iotas to be transferred, and, in that case, it selects a set of addresses from which to spend and a remainder address where to send the unspent iotas to.

The *Create transfer bundle* functionality outputs the finalized transaction.

3.5.4 Other functions

The application provides the user with auxiliary functions to manage their Light Node:

- *Get node info*: requests to the Full Node its specifications and outputs them

- *Get transactions to approve*: requests to the Full Node the hash of two not-yet-approved transactions and outputs them
- *Get balances*: requests to the Full Node the balances of the addresses given as inputs and outputs them
- *Were addresses spent from*: requests to the Full Node if the addresses given as inputs have been already used to spend and outputs the response
- *Get balances and states*: requests to the Full Node the balances of the addresses given as inputs and if they have been already used to spend and outputs the response
- *Get inclusion state*: requests to the Full Node if the transaction corresponding to the hash given in input has already been confirmed

4 System setup guide

4.1 Hardware description

This section describes the hardware components needed for developing applications based on digital MEMS microphones. The following sub-sections describe the individual components.

4.1.1 STM32 Nucleo platform

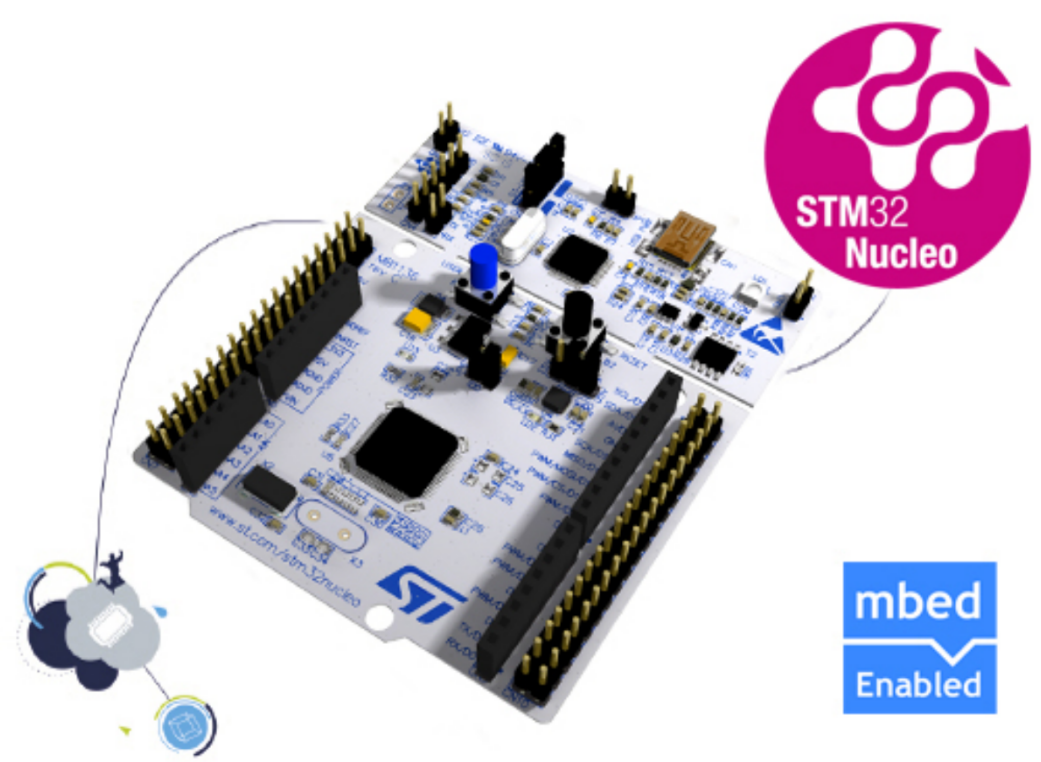
STM32 Nucleo development boards provide an affordable and flexible way for users to test solutions and build prototypes with any STM32 microcontroller line.

The Arduino™ connectivity support and ST morpho connectors make it easy to expand the functionality of the STM32 Nucleo open development platform with a wide range of specialized expansion boards to choose from.

The STM32 Nucleo board does not require separate probes as it integrates the ST-LINK/V2-1 debugger/ programmer.

The STM32 Nucleo board comes with the comprehensive STM32 software HAL library together with various packaged software examples.

Figure 6. STM32 Nucleo board



Information regarding the STM32 Nucleo board is available at www.st.com/stm32nucleo

4.2 Hardware setup

The following hardware components are needed:

1. an STM32 Nucleo development platform equipped with the Ethernet module (typically a Nucleo 144 pins)
2. a USB type A to Micro USB cable to connect the STM32 Nucleo to the PC

4.3 Software setup

The following software components are needed to set up the development environment for creating IOTA DLT applications for the STM32 Nucleo equipped with an Ethernet module:

- X-CUBE-IOTA1: an expansion for STM32Cube dedicated to IOTA DLT applications development. The X-CUBE-IOTA1 firmware and related documentation is available on st.com
- development tool-chain and compiler: the STM32Cube expansion software supports the following environments:
  - IAR Embedded Workbench for ARM® (EWARM) toolchain + ST-LINK/V2
  - RealView Microcontroller Development Kit (MDK-ARM) toolchain + ST-LINK/V2
  - System Workbench for STM32 (SW4STM32) + ST-LINK/V2

4.4 System setup

The STM32 Nucleo development board allows the exploitation of the IOTA DLT features.

The board integrates the ST-LINK/V2-1 debugger/programmer. You can download the relevant version of the ST-LINK/V2-1 USB driver at STSW- LINK009.

Revision history

Table 4. **Document revision history**

Date	Revision	Changes
13-Jun-2019	1	Initial release
18-Jun-2019	2	Updated Section 3.4.8.1 TX_IN and TX_OUT, Section 3.4.8.3 Sending data through zero-value transactions and Section 3.4.8.4 Sending funds through transfer transactions.

Contents

List of figures

List of tables