Encrypted Modbus RTU

This page describes how to integrate a NiLAB integrated-drive linear motor (NLi / GDi family) into a PLC-based machine controller using the NiLAB Encrypted Modbus RTU protocol.

The encryption layer runs transparently on top of standard Modbus RTU over RS-485. No changes to the physical wiring or to the standard Modbus RTU register map are required. Drives with encryption disabled communicate as standard Modbus RTU slaves and are fully backward-compatible with any generic Modbus master.

The cryptographic implementation is provided as a **closed, pre-compiled library** for each supported development environment (CODESYS, C/C++, .NET). The internal algorithm is proprietary to NiLAB GmbH. The library is available to integrators under a NiLAB technology partner agreement — contact NiLAB for access.

If encryption is enabled on the drive, the drive will **only** respond to encrypted frames. Any plaintext Modbus request (FC3, FC6, FC16) will be silently discarded. You must use the NiLAB encryption library to communicate with an encryption-enabled drive.

1. Concept and protocol overview

1.1 What the encryption layer provides

The NiLAB Encrypted Modbus RTU layer adds three security properties to the standard Modbus RTU protocol:

* **Authenticity** — every frame carries a cryptographic tag. The drive verifies this tag before processing any request. Forged, corrupted or replayed frames are silently discarded without any response.
* **Confidentiality** — the Modbus PDU (function code and register data) is encrypted. An observer on the RS-485 bus cannot read register values or setpoints.
* **Anti-replay** — a monotonic frame counter embedded in each frame prevents a captured valid frame from being re-injected at a later time.

The protection is bidirectional: both the master→drive request and the drive→master response are encrypted and authenticated.

1.2 Key

Each NiLAB drive is provisioned at the factory with a **unique 128-bit AES key**, stored in protected flash memory. The key is tied to the drive serial number in the NiLAB key database.

To obtain the key for a specific drive, contact NiLAB GmbH with the drive serial number. The key is delivered as a 32-character hexadecimal string, for example:

```
1D23586E43A218620EC5C7486274CAD0
```

This key must be treated as a secret. It must be stored in protected memory on the PLC side and must never be transmitted over the bus or written to system logs.

1.3 Encrypted frame format

The encrypted frame replaces the standard Modbus function code and PDU with a fixed-structure wrapper identified by function code **0x65**:

```
Standard Modbus RTU (plaintext):
[ Node ID (1) ][ FC (1) ][ PDU (N) ][ CRC16 (2) ]

NiLAB Encrypted Modbus RTU:
[ Node ID (1) ][ 0x65 (1) ][ Counter (4) ][ Ciphertext (N) ][ Tag (8) ][
CRC16 (2) ]
```

Field	Size (bytes)	Description
Node ID	1	Modbus slave address, unchanged
0x65	1	NiLAB encrypted PDU marker
Counter	4	Frame counter, big-endian, strictly increasing
Ciphertext	N	Encrypted Modbus PDU (same content as the plaintext PDU would be)
Tag	8	Cryptographic authentication tag over Counter + Ciphertext
CRC16	2	Standard Modbus CRC16 over all preceding bytes

The overhead added by the encryption wrapper is **12 bytes** per frame (4 counter + 8 tag) compared to the equivalent plaintext frame.

The **CRC16** uses the standard Modbus polynomial and byte order (low byte first) and is calculated over the entire frame including Node ID, function code 0x65, counter, ciphertext and tag — exactly as in standard Modbus RTU.

1.4 Frame counter

The frame counter is a 32-bit unsigned integer, transmitted big-endian (most significant byte first). It is independent per direction:

* The **master (PLC)** increments its TX counter with every request it sends.
* The **drive** tracks the last accepted master TX counter and rejects any frame with a counter that is not strictly greater than the last accepted value.
* The drive uses its own TX counter for encrypted responses; the master tracks this to detect replayed responses.

Counters start at 0 on first commissioning. **They must be saved to non-volatile memory** and restored on startup — see Section 5.

1.5 Drive behavior summary

State of drive	Plaintext FC3/FC6 from master	Encrypted FC 0x65 from master (correct key)	Broadcast sync (FC 0x80, node 0x00)
Encryption **disabled** (factory default)	Accepted, normal response	Not understood, exception response	Always accepted
Encryption **enabled**	Silently discarded	Accepted, encrypted response	Always accepted

2. Getting started

2.1 Prerequisites

- A NiLAB NLi or GDi integrated-drive linear motor with encryption firmware (ask NiLAB for the firmware version that supports encryption).
- The 16-byte AES key for the specific drive, obtained from NiLAB (see Section 1.2).
- The NiLAB crypto library for your development environment (see Section 3).
- RS-485 wiring: standard Modbus RTU bus, 2-wire, with correct termination.

2.2 Communication parameters

The encrypted protocol uses the same serial parameters as standard NiLAB Modbus RTU:

Parameter	Value
Baud rate	115200 bps (default)
Data bits	8
Parity	None
Stop bits	1
Protocol	Modbus RTU
Node ID	Configured on drive (default: 1)

2.3 Frame sizes for timing calculations

The total frame size depends on the inner PDU size. For the most common operations:

Operation	Inner PDU (bytes)	Total encrypted frame (bytes)
FC6 write, 1 register	5	1+1+4+5+8+2 = **21**
FC3 read, 1 register	5 (request)	1+1+4+5+8+2 = **21**
FC3 read, 1 register	4 (response)	1+1+4+4+8+2 = **20**
FC3 read, N registers	3+2N (response)	1+1+4+(3+2N)+8+2 = **19+2N**

Use these values to set appropriate serial receive timeouts on the PLC side.

3. NiLAB crypto library

NiLAB provides a closed, pre-compiled library for each supported PLC development environment. The library exposes a minimal, well-defined API(see Section 4) and handles all cryptographic operations internally. No knowledge of the underlying algorithm is required for integration.

3.2 Obtaining the library

The library is distributed under a NiLAB technology partner agreement. To request access:

* Contact NiLAB GmbH with subject line “Encrypted Modbus Library Request”
* Specify your development environment (see table above)
* Provide the serial number(s) of the NiLAB drive(s) you are integrating

NiLAB will supply the library package together with the corresponding drive key(s).

---

4. Library API

All library variants expose the same logical API, adapted to the conventions of the target environment. The following description uses pseudocode notation.

4.1 Initialization

```
\
NiLAB_Init(ctx, key[16], tx_counter, rx_counter)\
```

Initialises a crypto context for one drive connection.

Parameter	Type	Description
ctx	opaque handle	Context object — allocate one per drive
key	byte[16]	The 16-byte AES key for this drive
tx_counter	uint32	Initial TX counter (0 for first commissioning, saved value on restart)
rx_counter	uint32	Initial RX counter (0 for first commissioning, saved value on restart)

4.2 Build a write request frame (FC6)

```
frame_len = NiLAB_BuildWriteFrame(ctx, node_id, reg_address, value,
frame_buf)
```

Builds a complete encrypted FC6 write frame, ready to transmit over RS-485.

Parameter	Type	Description
ctx	handle	Initialised context
node_id	uint8	Modbus slave address
reg_address	uint16	Register address to write
value	uint16	Value to write
frame_buf	byte[]	Output buffer (minimum 23 bytes)
return value	int	Number of bytes to transmit

4.3 Build a read request frame (FC3)

```
frame_len = NiLAB_BuildReadFrame(ctx, node_id, reg_address, qty, frame_buf)
```

Builds a complete encrypted FC3 read frame for `qty` consecutive registers.

4.4 Parse and decrypt a response frame

```
status = NiLAB_ParseResponse(ctx, raw_frame, frame_len, node_id,
reg_values_out)
```

Verifies the authentication tag, checks the counter, and decrypts the drive response.

Return value	Meaning
NILAB_OK (0)	Frame authenticated and decrypted successfully. reg_values_out valid.
NILAB_ERR_AUTH (1)	Authentication tag mismatch — frame forged, corrupted or wrong key
NILAB_ERR_REPLAY (2)	Frame counter not increasing — possible replay attack
NILAB_ERR_SHORT (3)	Frame too short to be a valid encrypted response
NILAB_ERR_TIMEOUT(4)	No response received within timeout

4.5 Read current counter values (for NVM persistence)

```
tx_counter = NiLAB_GetTxCounter(ctx)
rx_counter = NiLAB_GetRxCounter(ctx)
```

Returns the current counter values. Save these to non-volatile memory periodically and on shutdown (see Section 5).

---

5. Counter persistence

The anti-replay mechanism requires that the master TX counter is strictly increasing across power cycles. Both `tx_counter` and `rx_counter` must be saved to non-volatile memory and restored on startup.

Recommended strategy: on every PLC startup, initialise the context with the last saved counter value plus a safety margin (e.g. +1000). This ensures that even if the NVM write was delayed at shutdown, the restored counter is still higher than any counter the drive accepted in the previous session.

Edit

5.1 CODESYS (RETAIN variables)

```
VAR \RETAIN
  nTxCounterNVM : UDINT := 0;
  nRxCounterNVM : UDINT := 0;
END_VAR

(* On startup, once: *)
NiLAB_Init(ctx, key, nTxCounterNVM + 1000, nRxCounterNVM);

(* After each successful transaction: *)
nTxCounterNVM := NiLAB_GetTxCounter(ctx);
nRxCounterNVM := NiLAB_GetRxCounter(ctx);
```

CODESYS RETAIN variables are automatically saved to flash on power cycle — no additional NVM write is needed.

5.2 C / bare-metal

```
/* On startup: */
uint32_t tx_saved, rx_saved;
NVM_Read(&tx_saved, &rx_saved);   /* your NVM read function */
NiLAB_Init(&ctx, key, tx_saved + 1000, rx_saved);

/* After each successful transaction: */
NVM_Write(NiLAB_GetTxCounter(&ctx), NiLAB_GetRxCounter(&ctx));
```

5.3 First commissioning

On first commissioning (no NVM data available), pass 0 for both counters. The drive will accept the first frame from counter 0 and set its internal reference accordingly.

---

6. Integration examples

The following examples show the typical call sequence for the three supported environments. They assume the library has been imported into the project and the context has been initialised in the

startup section.

6.1 CODESYS V3.5 — Structured Text

Add the NiLAB library to your CODESYS project via **Tools → Library Manager → Add Library → NiLAB_ModbusCrypto**.

```
(*
 ---------------------------------------------------------------
 Startup / initialisation (call once from PLC_PRG, first scan)
 --------------------------------------------------------------- *)
\VAR
  ctx   : NiLAB_Ctx;          (* Crypto context -- one per drive *)
  key          : ARRAY[0..15] OF BYTE := [
                    16#1D, 16#23, 16#58, 16#6E, 16#43, 16#A2, 16#18, 16#62,
                    16#0E, 16#C5, 16#C7, 16#48, 16#62, 16#74, 16#CA, 16#D0
];
  aFrame       : ARRAY[0..31] OF BYTE;
  aResponse    : ARRAY[0..31] OF BYTE;
  nFrameLen    : UDINT;
  nStatus      : INT;
  nRegValue    : WORD;
  bInitDone    : BOOL := FALSE;
END_VAR

IF NOT bInitDone \THEN
  NiLAB_Init(ctx := ctx,
              key := key,
              tx_counter := nTxCounterNVM + 1000,
              rx_counter := nRxCounterNVM);
  bInitDone := TRUE;
END_IF

(*
 ---------------------------------------------------------------
 Write register 0x0010 = 0x1234 on node \1
 --------------------------------------------------------------- *)
nFrameLen := NiLAB_BuildWriteFrame(
              ctx         := ctx,
              node_id     := 1,
              reg_address := 16#0010,
              value       := 16#1234,
              frame_buf   := aFrame);

(* Send aFrame (nFrameLen bytes) via your serial/RS-485 block *)
ComSend(port := COM1, data := aFrame, length := nFrameLen);

(* Wait for response, then: *)
ComReceive(port := COM1, data := aResponse, length => nRespLen);
```

```
nStatus := NiLAB_ParseResponse(
             ctx       := ctx,
             raw_frame := aResponse,
             frame_len := nRespLen,
             node_id   := 1,
             reg_values_out := nRegValue);

IF nStatus = NILAB_OK \THEN
  (* write confirmed, nRegValue contains echoed value *)
  nTxCounterNVM := NiLAB_GetTxCounter(ctx);
  nRxCounterNVM := NiLAB_GetRxCounter(ctx);
\ELSE
  (* handle error: nStatus = NILAB_ERR_AUTH, NILAB_ERR_REPLAY, etc. *)
END_IF

(*
 ---------------------------------------------------------------
 Read register 0x0020 on node \1
 --------------------------------------------------------------- *)
nFrameLen := NiLAB_BuildReadFrame(
              ctx         := ctx,
              node_id     := 1,
              reg_address := 16#0020,
              qty         := 1,
              frame_buf   := aFrame);

ComSend(port := COM1, data := aFrame, length := nFrameLen);
ComReceive(port := COM1, data := aResponse, length => nRespLen);

nStatus := NiLAB_ParseResponse(
             ctx           := ctx,
             raw_frame     := aResponse,
             frame_len     := nRespLen,
             node_id       := 1,
             reg_values_out := nRegValue);

IF nStatus = NILAB_OK \THEN
  wMotorStatus := nRegValue;   (* use the register value *)
\END_IF
```

Replace `ComSend` / `ComReceive` with the actual serial function blocks available in your CODESYS runtime (e.g. SL_SER_SND / SL_SER_RCV on Wago, RS on generic IEC 61131-3, `ModbusSerial` master blocks if you bypass the function code layer). The NiLAB library operates at raw byte level and is independent of the serial transport block used.

6.2 C / C++ (bare-metal, RTOS, Linux)

Add `nilab_modbus_crypto.h` to your include path and link against `libnilab_modbus_crypto.a` (ARM) or `nilab_modbus_crypto.lib` (x86 Windows).

```
#include "nilab_modbus_crypto.h"

/*
 --- Initialisation (call once at startup) --- */
static NiLAB_Ctx g_ctx;

static const uint8_t g_key[16] = {
  0x1D, 0x23, 0x58, 0x6E, 0x43, 0xA2, 0x18, 0x62,
  0x0E, 0xC5, 0xC7, 0x48, 0x62, 0x74, 0xCA, \0xD0
};

void motor_comm_init(void)
{
  uint32_t tx_saved, rx_saved;
  NVM_Read(&tx_saved, &rx_saved);   /* your NVM read */
  NiLAB_Init(&g_ctx, g_key, tx_saved + 1000, rx_saved);
}

/*
 --- Write register 0x0010 = 0x1234 on node 1 --- */
int motor_write_register(uint16_t reg_addr, uint16_t value)
{
  uint8_t frame[32];
  uint8_t response[32];

  int flen = NiLAB_BuildWriteFrame(&g_ctx, 1, reg_addr, value, frame);

  rs485_send(frame, flen);   /* your RS-485 send */
  int rlen = rs485_receive(response, sizeof(response), 200 /*ms*/);

  NiLAB_Status st = NiLAB_ParseResponse(&g_ctx, response, rlen, 1, NULL);

  if (st == NILAB_OK) {
      NVM_Write(NiLAB_GetTxCounter(&g_ctx), NiLAB_GetRxCounter(&g_ctx));
  }
  return (int)st;
}

/*
 --- Read register 0x0020 on node 1 --- */
int motor_read_register(uint16_t reg_addr, uint16_t *value_out)
{
  uint8_t frame[32];
  uint8_t response[32];

  int flen = NiLAB_BuildReadFrame(&g_ctx, 1, reg_addr, 1, frame);

  rs485_send(frame, flen);
  int rlen = rs485_receive(response, sizeof(response), 200);
```

```
  NiLAB_Status st = NiLAB_ParseResponse(&g_ctx, response, rlen, 1,
value_out);
  return (int)st;
}
```

6.3 Ladder Logic (generic)

Ladder Logic does not have native support for byte-level cryptographic operations. The recommended approach for Ladder-based PLCs is:

- Declare the NiLAB function block (`FB_NiLAB_ModbusCrypto`) as a called block in the project. In most Ladder environments that also support IEC 61131-3, function blocks can be called from Ladder networks as a box instruction. - Use the Ladder network only for trigger logic, mode selection and status monitoring. The function block handles all cryptographic and communication operations internally.

```
Network 1: Enable block on rising edge of StartComm \contact
--[P]--[StartComm]----[FB_NiLAB.xEnable := TRUE]--

Network 2: Select write \mode
--[WriteCmd]----------[FB_NiLAB.xWrite := TRUE]---
--[/WriteCmd]---------[FB_NiLAB.xWrite := FALSE]--

Network 3: Pass register address and \value
--[MOVE]-- nTargetReg  --> FB_NiLAB.\nRegAddress
--[MOVE]-- nSetpoint   --> FB_NiLAB.nWriteValue

Network 4: Process result on \Done
--[FB_NiLAB.xDone]---[ProcessResultSubroutine]----

Network 5: Error \handling
--[FB_NiLAB.xError]--[SET]--AlarmBit-------------
```

For PLCs that do **not** support function blocks or Structured Text subroutines at all (pure relay-ladder systems), direct encryption is not feasible at the PLC level. In this case, contact NiLAB for information on the **NiLAB Crypto Gateway** — a compact external module that acts as a transparent encryption bridge between the PLC's standard Modbus RTU port and the drive bus.

---

7. Error handling

All APIfunctions return a status code. The following table describes the correct response to each error condition in a machine application:

Status code	Meaning	Recommended action
NILAB_OK	Transaction successful	Normal operation, update NVM counters

Status code	Meaning	Recommended action
NILAB_ERR_AUTH	Frame authentication failed	Log event, retry up to 3 times; if persistent, check key
NILAB_ERR_REPLAY	Counter mismatch or replay detected	Log event, reset context with NiLAB_Init and saved counters
NILAB_ERR_SHORT	Response frame too short or wrong node ID	Check RS-485 wiring and termination
NILAB_ERR_TIMEOUT	No response received within timeout	Check drive power, address and baud rate

In a safety-relevant axis application, any persistent `NILAB_ERR_AUTH` or `NILAB_ERR_REPLAY` error should trigger an immediate axis stop and operator alarm. These errors should not occur in normal operation and may indicate tampering or a hardware fault.

---

8. Troubleshooting

Symptom	Likely cause	Solution
Drive does not respond to any frame	Encryption enabled, PLC sending plaintext	Use NiLAB library. All plaintext frames are rejected by design.
Drive responds only to first frame after power-on	Counter mismatch after PLC restart	Restore saved counters at startup (Section 5)
Persistent NILAB_ERR_AUTH errors	Wrong key for this drive	Verify key against NiLAB key database using drive serial number
Intermittent NILAB_ERR_AUTH errors	RS-485 bus noise corrupting frames	Check cable, termination resistors (120 Ω at both ends), cable length
NILAB_ERR_REPLAY after PLC restart	Saved counter is lower than drive expects	Increase NVM safety margin from +1000 to +10000 and recommission
Communication works but drive ignores motion commands	Drive in error/protection state	Use NiLAB Starter to check drive fault register before PLC control

---

9. Reference

* NiLAB Security & Vulnerability Disclosure Policy

Contact and support

For library access, key provisioning or technical integration support:

* **Web:** www.nilab.at
* **Technical support portal:** www.ni-lab.online

To report a security vulnerability in NiLAB products: Report a vulnerability

From:
https://dokuwiki.nilab.at/ - **NiLAB GmbH Knowledgebase**

Permanent link:
https://dokuwiki.nilab.at/doku.php?id=integrated_drive_motors:modbus_crypto

Last update: **2026/07/02 05:56**