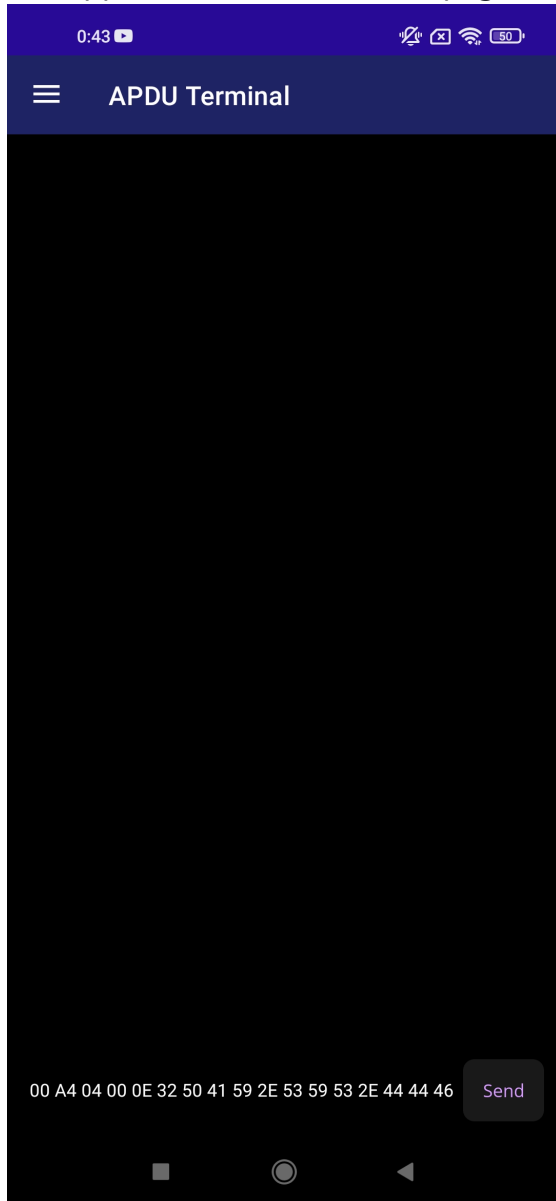
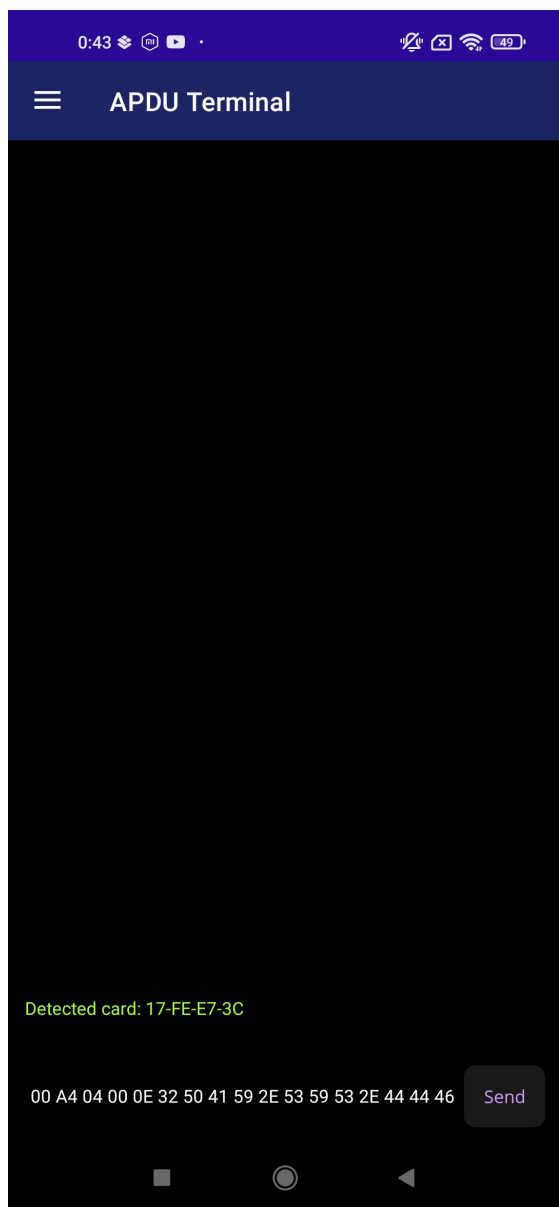


Getting Started - example on EMV expired card

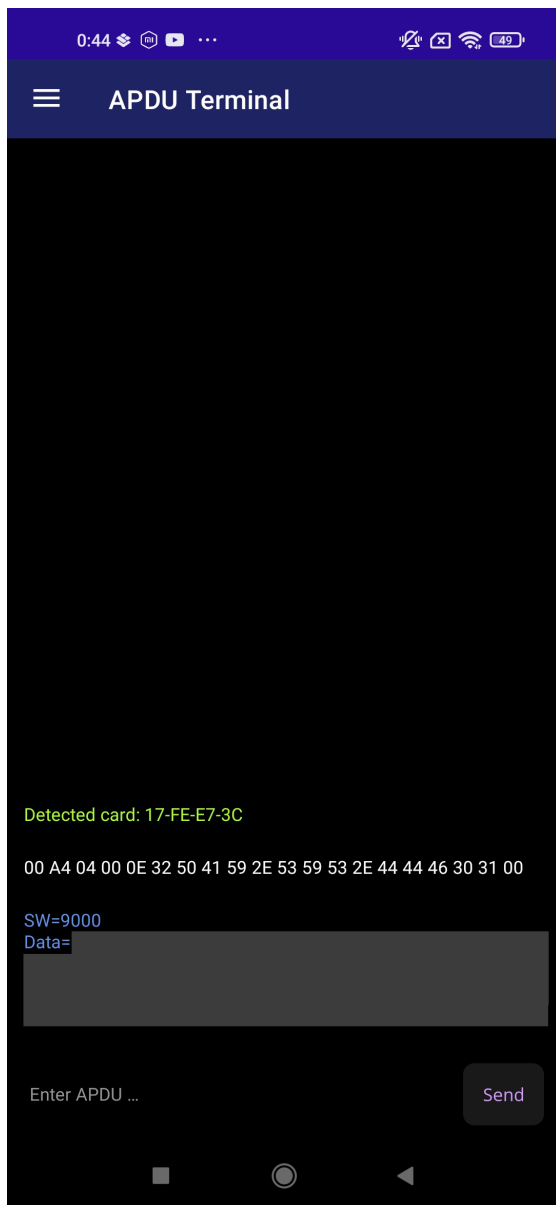
The app starts on the Terminal page. Here, you can send raw APDU commands in hex.



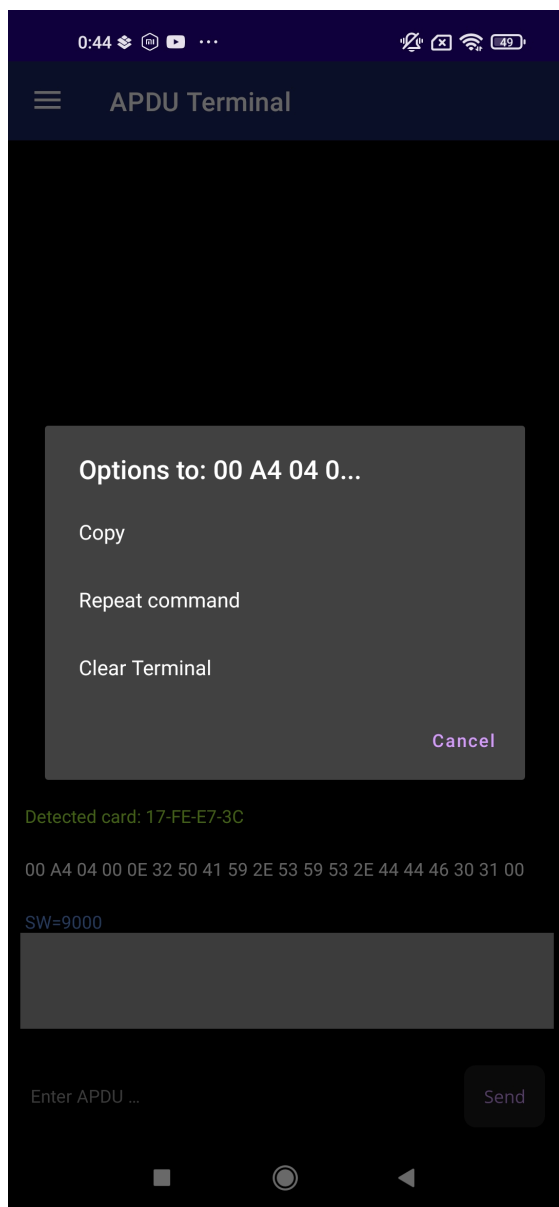
The default command in the field (00 A4 04 00 0E 31 50 41 59 2E 53 59 53 2E 44 44 46 30 31 00) is a SELECT by Name command for '2PAY.SYS.DDF01'.



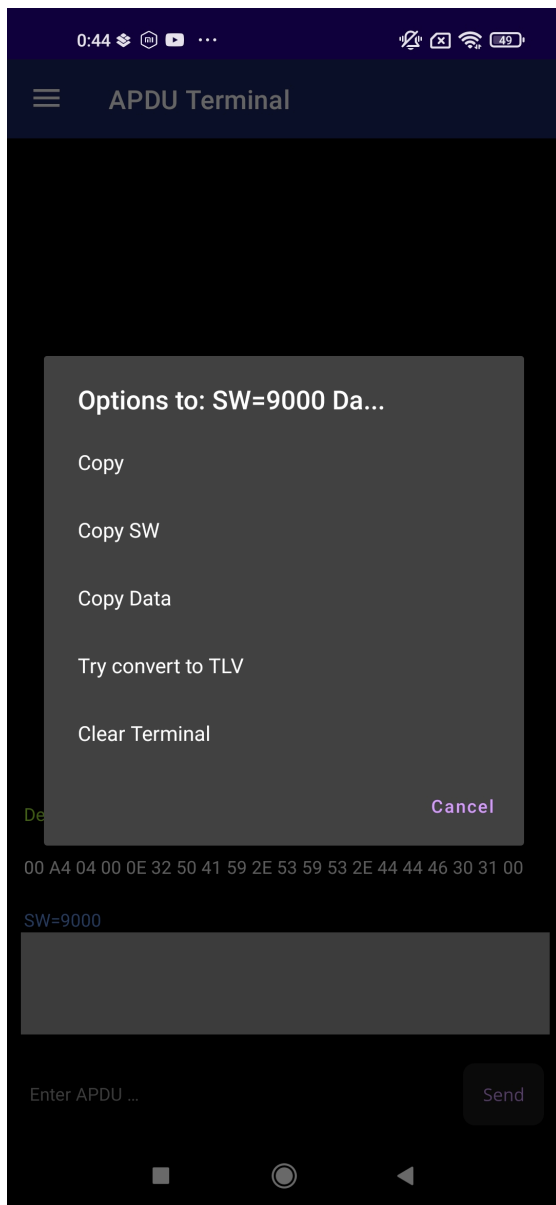
When the card is detected, a green message notifies you about it.



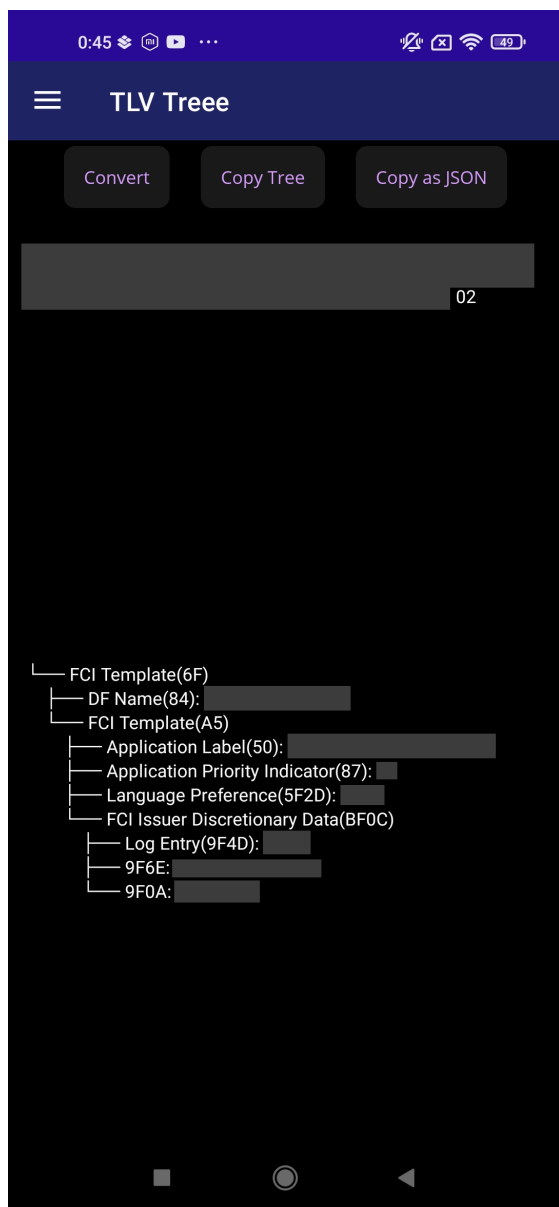
After placing the card near the phone and keeping it there until the notification appears, you can send a command. The response includes two parts: the SW (Status Word) and the Data, both in hexadecimal format. The SW indicates the response status from the card. For example, in this case, a value of 90XX means that the selected file exists and was successfully selected.



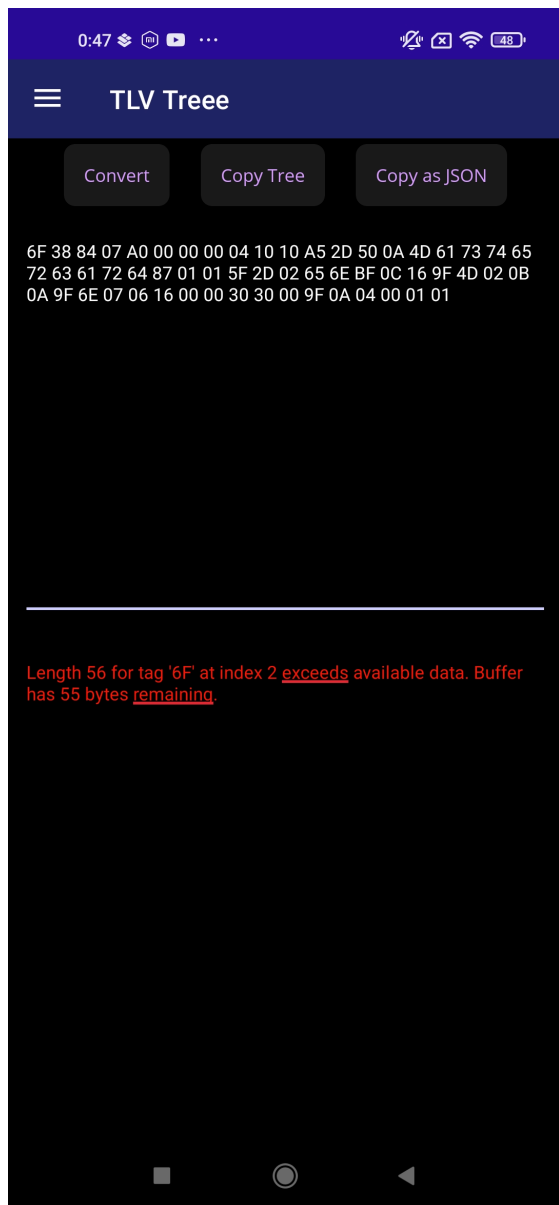
You can use a single tap gesture on the message in the Terminal to see possible options.



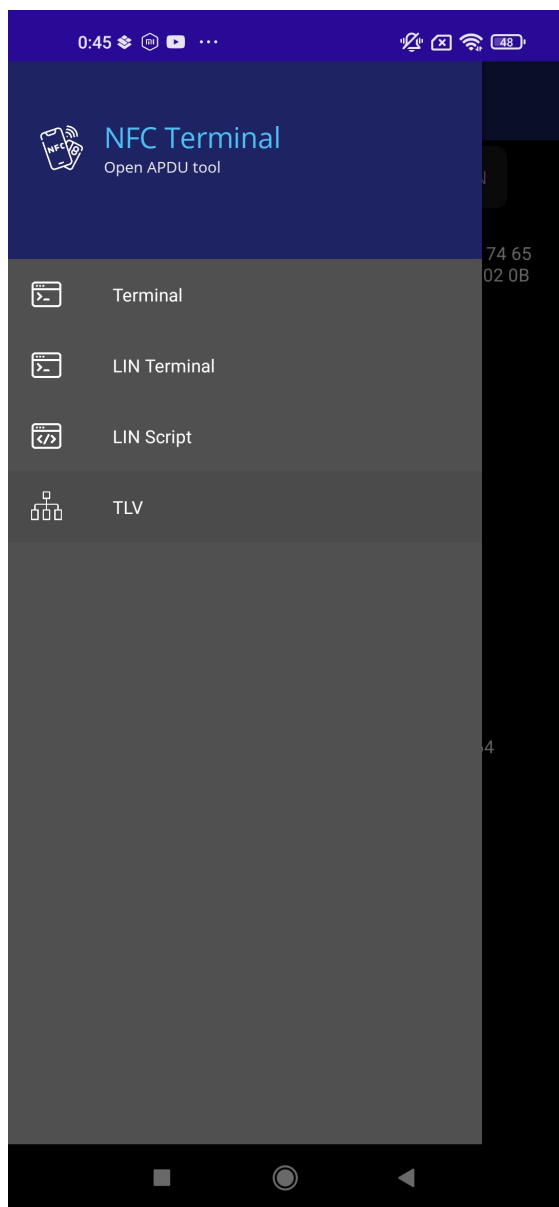
Command messages and responses have different options. In the response, you can additionally copy only a part of the data or try to parse the response data as a TLV tree.



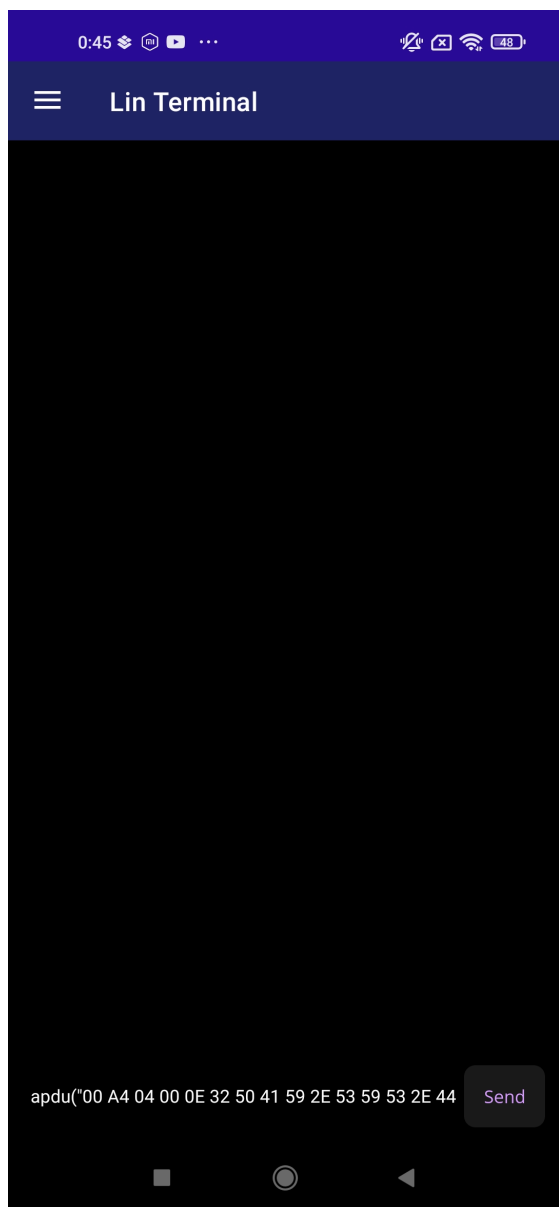
If you choose to parse the TLV, the app automatically navigates to this page and shows the result (if the data is valid). From here, you can modify the input, copy the output from the field below, or use one of the two buttons: 'Copy Tree', which copies the tree as displayed in the box below, or 'Copy JSON', which copies the tree in JSON format.



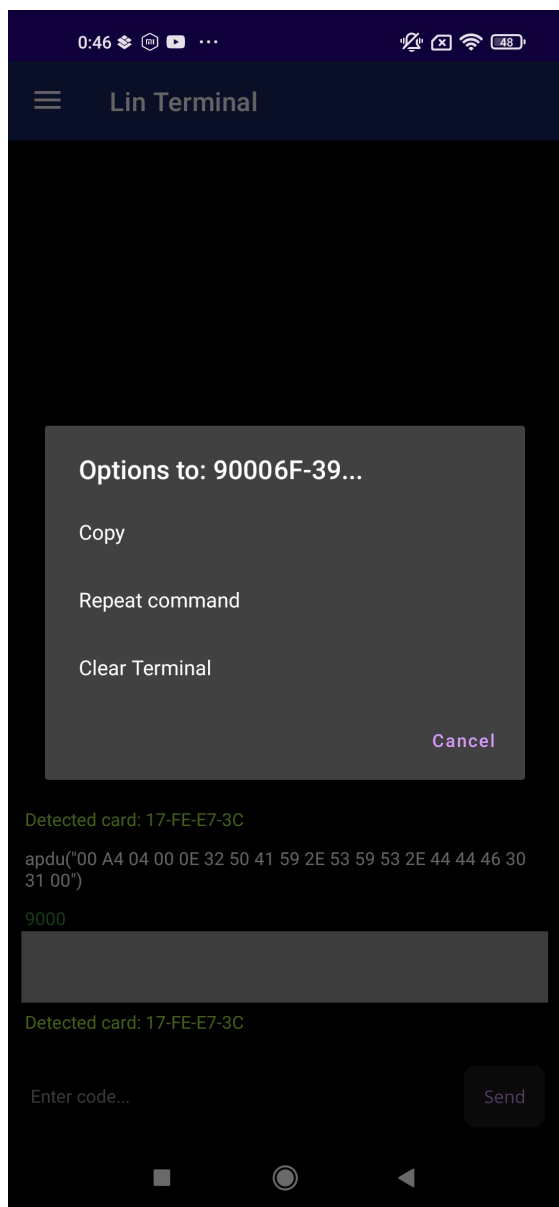
If the given hex data fails during tree parsing, you'll see an error message instead of the tree. The message will try to explain what might be wrong.



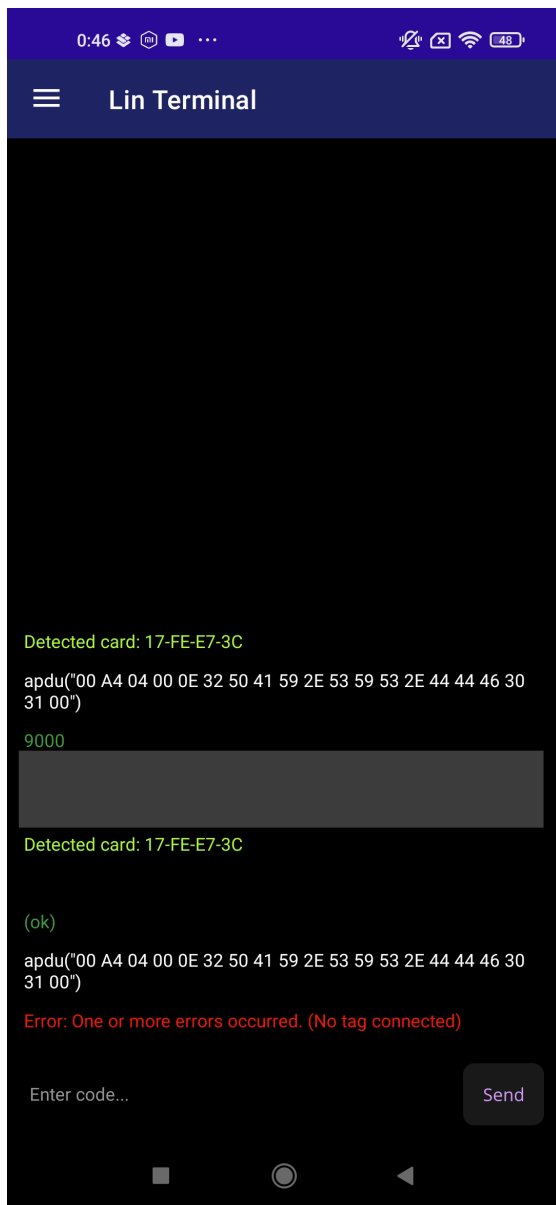
Using the snackbar, you can navigate through all four pages.



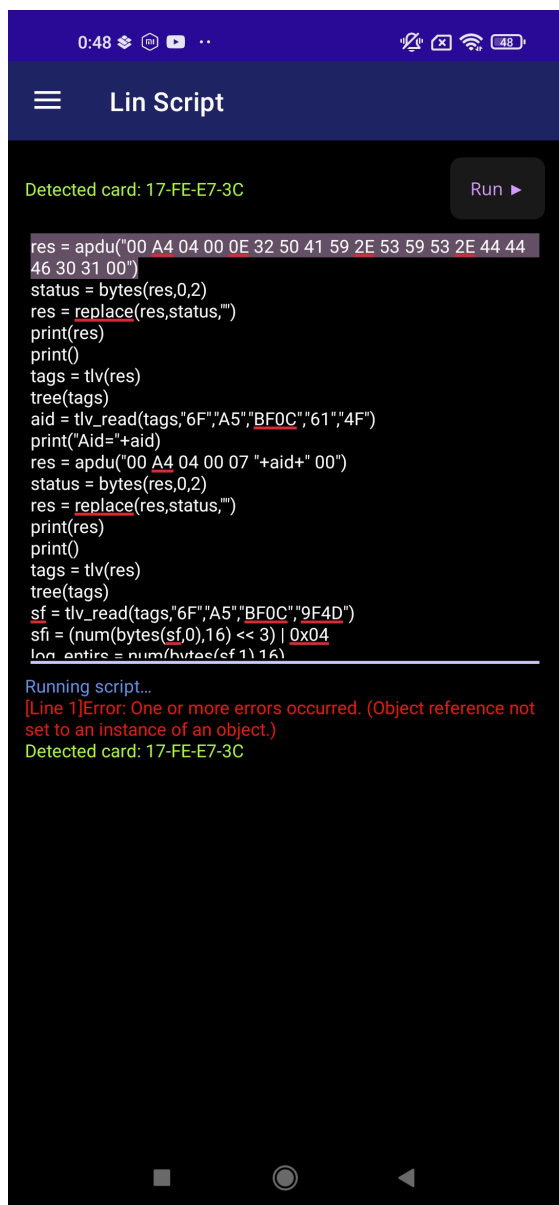
Let's navigate to the LIN Terminal. From here, you can also communicate with the card using the scripting language 'LIN' ([see LIN](#)) in the interactive console.



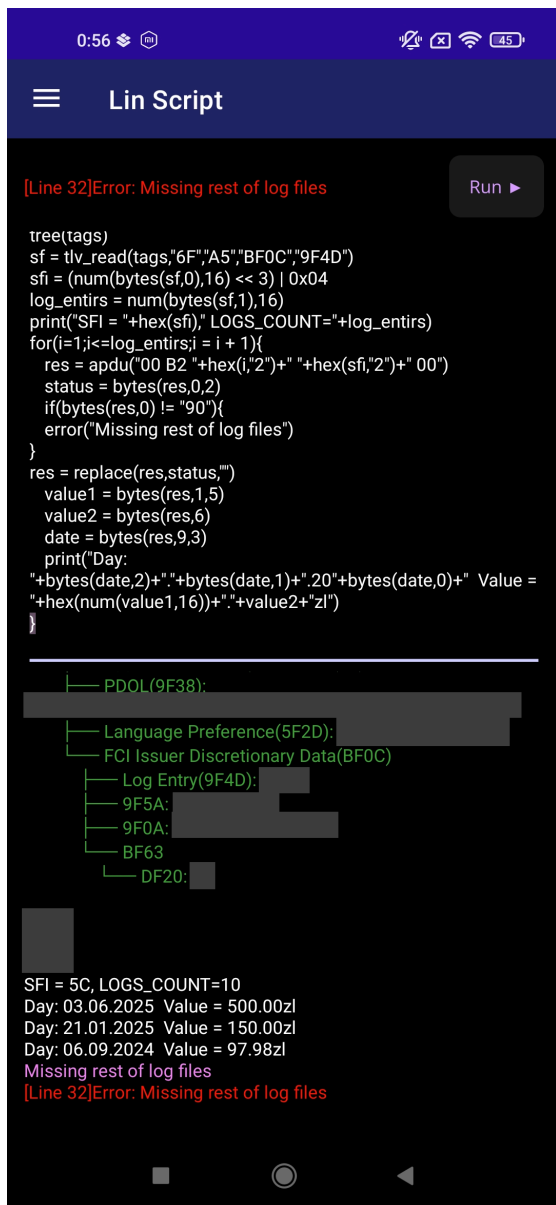
Here, you get the same default command, but written in LIN. Messages about card detection also appear here.



Card detection works the same on all three pages. There's no specific disconnect message — but if the card is out of range or communication is interrupted, you'll see a message when you try to send something to the card



Let's move to the LIN Script page. Here, you can develop multi-line scripts using the LIN language. A new error message might appear, as shown below. This usually means the card isn't connected — or that it was never connected during the current session. Later, you may see a standard error instead. \



The code file includes an example LIN script. This script tries to read log files from an EMV card and display the transaction date and value. The script ends with an error, but that's expected because not all log files may exist or the card may not grant access to them.

You can modify the script, re-run it, and have fun! (P.S. There's a function to highlight the line where the error occurred, but it's not fully implemented yet—so don't rely on it too much.)

LinExtension API

Overview

`LinExtension` registers a set of helper functions into the Lin interpreter, providing:

- Basic I/O (`print`, `error`, `printc`)
- Numeric and string conversions (`num`, `hex`, `ascii`)
- Substring and search utilities (`substr`, `replace`, `find`)
- Byte-level slicing (`bytes`)
- NFC/Smartcard APDU transmission (`apdu`)
- TLV parsing and navigation (`tlv`, `tree`, `tlv_read`)

Below you'll find a description of each function, its parameters, return value, and a simple usage example. (Note: internal argument validation logic is not detailed here.)

1. `print`

Signature (in Lin):

```
print(arg1, arg2, ..., argN)
```

Description: Prints any number of arguments to the console in plain white text. Each argument is converted to its string representation (via `ToString()`) and joined with commas.

- **Parameters:** – `arg1`, `arg2`, ..., `argN` (any type)
- **Return Value:** Returns `null` (purely an output operation).
- **Example:**

```
print("Result is", 123, myVariable)
```

Output in console (white text):

```
Result is,123,<value of myVariable>
```

2. `error`

Signature (in Lin):

```
error(arg1, arg2, ..., argN)
```

Description: Similar to `print`, but displays the message in violet (error color) and then throws an exception (`Exception`) with the same joined text. Useful for signaling an error from within a Lin script.

- **Parameters:** – `arg1`, `arg2`, ..., `argN` (any type)
- **Return Value:** Does not return normally—throws an exception immediately after printing.
- **Example:**

```
if (someConditionFailed) {  
    error("Something went wrong:", errorCode)  
}
```

Output in console (violet text):

```
Something went wrong:,<errorCode>
```

Then execution halts with an exception carrying that message.

3. `printc`

Signature (in Lin):

```
printc(colorHexString, arg1, arg2, ..., argN)
```

Description: Prints all arguments (from the second onward) in a custom color specified by the first argument (hex format, e.g. `"#FF0000"`). The first argument is treated as a hex color string, then the rest are joined with commas and drawn in that color.

- **Parameters:**
 1. `colorHexString` (string) – e.g. `"#00FF00"`, `"FFAA00"`, etc.
 2. `arg1`, `arg2`, ..., `argN` (any type) – values to print in that color.
- **Return Value:** Returns `null` (purely an output operation).
- **Example:**

```
printc("#FF00FF", "Status", 200, "OK")
```

Output (magenta color):

```
Status,200,OK
```

4. apdu

Signature (in Lin):

apdu(hexString)

Description: Sends an APDU command over NFC (using the provided `INfcService` instance). Accepts a hex string, transmits it to the card, waits for the response, and then splits the response into “data” and Status Word (SW). Returns a formatted string:

```
[SW]\t[DATA]
```

SW - 4-digit hex `status` word

DATA - response `data in` bytes hex with hyphens

Important:

- If no NFC listener or card connection is available (for example, if the card is not present, NFC was never initialized or connected, or the card got disconnected), a `NullReferenceException` or related error may be thrown. In such cases, you should catch the exception (or check that NFC is connected) before calling `apdu`.
- **Example error conditions:**
 1. **Card not present:** NFC transceive will fail or return an empty response—handle by checking for `null` or catching a `NullReferenceException`.
 2. **NFC service not initialized / never connected:** Attempting to call `nfc.TransceiveAsync(...)` on a `null` service object will cause a `NullReferenceException`.
 3. **Card disconnected mid-execution:** The `Task` launched for `TransceiveAsync` may throw an exception or return incomplete data—catch exceptions and verify response length.
- **Parameters:** – `hexString` (string) – ASCII-hex representation of the APDU (e.g. `"00A404000E315041592E5359532E4444463031"`).
- **Return Value:** `string` formatted as:

```
9000\t6F-23-84-0E-A0-00-00-00-03-10-10-...
```

where `"9000"` is the SW (status word) and the bytes after the tab are the data payload in hyphenated hex form.

- **Example:**

```
response = apdu("00A404000E315041592E5359532E4444463031")
print("Card says:", response)
```


Possible response:

```
9000\t6F-23-84-0E-A0-00-00-00-03-10-10-A0-00-00-00-62-03-01-0C-01-02
```

5. num

Signature (in Lin):

```
num(stringNumber [, base])
```

Description: Converts a string representing a number into a 64-bit integer (`Int64`). If only one parameter is given, the default base is 10. If two parameters are supplied, the second must be an integer indicating the radix (e.g. 2, 8, 10, 16).

- **Parameters:**

1. `stringNumber` (string) – e.g. "FF", "1011", "123".
2. `base` (integer, optional) – the numeric base (e.g. 2 for binary, 16 for hex). Defaults to 10 if omitted.

- **Return Value:** `Int64` (the parsed integer value).

- **Example:**

```
x = num("1011", 2)
// x == 11
y = num("FF", 16)
// y == 255
z = num("42")
// z == 42 (default base 10)
```

6. hex

Signature (in Lin):

```
hex(value [, padString])
```

Description: Works in two modes:

1. `value**` is a number (`Int64`):** Converts the integer to an uppercase hex string (no leading "0x"). If `padString` is provided (a string, e.g. "4"), it specifies the minimum width (number of hex digits) and pads with leading zeros as needed.
2. `value**` is a string:** Converts each character of the input string to its ASCII byte, then outputs a space-separated hex representation for each byte.

- **Parameters:**

1. **value** – either a numeric type (Int64) or a **string**.
2. **padString** (string, optional) – when **value** is numeric, this defines the minimum field width (e.g. "4" → at least 4 hex digits; "2" → at least 2 hex digits).

- **Return Value:** **string** (hexadecimal representation).

- **Examples:**

```
// Number → hex
hex(255)
// "FF"
hex(255, "4")
// "00FF"
hex(16, "2")
// "10"
hex(16, "4")
// "0010"

// String → ASCII hex
hex("ABC")
// "41 42 43"
hex("Hi!")
// "48 69 21"
```

7. **ascii**

Signature (in Lin):

`ascii(hexString)`

Description: Takes a hex-encoded string (possibly containing spaces or hyphens) and decodes it into its ASCII text equivalent. Each pair of hex digits is treated as one byte and translated to its ASCII character.

- **Parameters:** – **hexString** (string) – e.g. "48656C6C6F", "48 65 6C 6C 6F", or "48-65-6C-6C-6F".
- **Return Value:** **string** (decoded ASCII text).
- **Example:**

```
text = ascii("48656C6C6F")
// "Hello"
name = ascii("4A-6F-68-6E")
// "John"
```

8. substr

Signature (in Lin):

```
substr(sourceString, beginIndex, length)
```

Description: Extracts a substring from the given `sourceString`, starting at `beginIndex` (zero-based), and of the specified `length`. This mirrors C#'s `String.Substring`.

- **Parameters:**

1. `sourceString` (string) – the original text.
2. `beginIndex` (integer) – zero-based starting position.
3. `length` (integer) – number of characters to extract.

- **Return Value:** `string` (the extracted substring).

- **Example:**

```
substr("ABCDEFGH", 2, 3)
// "CDE"
substr("Hello, world", 7, 5)
// "world"
```

9. replace

Signature (in Lin):

```
replace(sourceString, oldValue, newValue)
```

Description: Replaces **all** occurrences of `oldValue` in `sourceString` with `newValue`. Equivalent to C#'s `String.Replace`.

- **Parameters:**

1. `sourceString` (string) – original text.
2. `oldValue` (string) – substring to be replaced.
3. `newValue` (string) – replacement substring.

- **Return Value:** `string` (result after substitution).

- **Example:**

```
replace("abracadabra", "a", "o")
// "obrocodobro"
replace("2025-06-06", "-", "/")
// "2025/06/06"
```

10. find

Signature (in Lin):

```
find(sourceString, searchString)
```

Description: Returns the zero-based index of the first occurrence of `searchString` inside `sourceString`. If not found, returns `-1`. Corresponds to C#'s `String.IndexOf`.

- **Parameters:**

1. `sourceString` (string) – text to search in.
2. `searchString` (string) – text to search for.

- **Return Value:** `Int32` (index of first match, or `-1` if none).

- **Example:**

```
find("linextension", "ext")  
// 3  
find("linextension", "xyz")  
// -1
```

11. bytes

Signature (in Lin):

```
bytes(hexString, startIndex [, count])
```

Description: Takes a hex-encoded string (possibly with spaces or hyphens), cleans it to a contiguous hex stream, then treats every two hex digits as one byte. Starting at byte index `startIndex` (zero-based), it returns `count` consecutive bytes (as hex). If `count` is omitted, defaults to 1.

- **Parameters:**

1. `hexString` (string) – e.g. `"A0B1C2D3"`, `"A0 B1 C2 D3"`, `"A0-B1-C2-D3"`.
2. `startIndex` (integer) – zero-based index of the first byte to extract.
3. `count` (integer, optional) – number of bytes to extract. Defaults to 1 if omitted.

- **Return Value:** `string` – the extracted hex bytes concatenated (no separators).

- **Examples:**

```
bytes("A0B1C2D3", 1)  
// "B1"           (2nd byte)  
bytes("A0 B1 C2 D3", 1, 2)  
// "B1C2"         (bytes 1 and 2)
```

```
bytes("A0-B1-C2-D3", 0, 3)
// "A0B1C2"    (first 3 bytes)
```

12. tlv

Signature (in Lin):

```
tlv(hexString)
```

Description: Parses a hex-encoded TLV (Tag-Length-Value) data stream into a `TlvObject`. You can then navigate or inspect the resulting TLV hierarchy. Internally, it returns an object you can pass to `tree` or `tlv_read`.

- **Parameters:** – `hexString` (string) – a hex stream representing TLV data (e.g. "6F108407A0000000031010A5049F6501FF").
- **Return Value:** `TlvObject` – a hierarchical representation of parsed TLV.
- **Example:**

```
obj = tlv("6F108407A0000000031010A5049F6501FF")
```

`obj` can now be passed to `tree(obj)` or navigated with `tlv_read`.

13. tree

Signature (in Lin):

```
tree(tlvObject)
```

Description: Generates a text-based (ASCII-art) "tree" representation of the TLV hierarchy contained inside a `TlvObject`. Useful for visual inspection of nested TLV structures.

- **Parameters:** – `tlvObject` (`TlvObject`) – the result of a prior `tlv(...)` call.
- **Return Value:** `string` – multi-line ASCII tree (with branches and indentation).
- **Example:**

```
obj = tlv("6F108407A0000000031010A5049F6501FF")
print(tree(obj))
```

Might output something like:

```
└─ 6F ... (Tag 6F, Length 10)
   └─ 84 A0 00 00 00 03 10 10 (Application ID)
```

```
└─ A5 ... (Container)
  └─ 9F65 FF (Value = FF)
└─ ... other nodes
```

14. `tlv_read`

Signature (in Lin):

```
tlv_read(tlvObject, key1, key2, ..., keyN)
```

Description: Walks down a TLV hierarchy inside a `TlvObject` along a sequence of tag keys (strings). If the final node is a leaf containing a value, it returns that value (typically a byte array or hex string). If it's a container node, it returns another `TlvObject`. If any step of the path is invalid (tag not found or current node is not a container), it throws an `IndexOutOfRangeException` explaining where traversal failed.

- **Parameters:**

1. `tlvObject` (`TlvObject`) – root of the TLV tree.
2. `key1, key2, ..., keyN` (string) – tags (e.g. "6F", "A5", "9F65") describing the path from the root to the desired node.

- **Return Value:** – If the final node is a container: returns a `TlvObject`. – If the final node holds a raw value: returns that value (byte array or hex).

- **Example:**

```
root = tlv("6F108407A0000000031010A5049F6501FF")
// Suppose inside tag 6F → tag A5 → tag 9F65 is a value
val = tlv_read(root, "6F", "A5", "9F65")
```

If `9F65` is a leaf containing raw data, `val` holds that data; if it's itself a sub-TLV container, `val` is a `TlvObject` for further navigation.

Usage Summary

All of these functions are registered inside `LinExtension`'s constructor (when you create `new LinExtension(...)`). Once registered, you can call them directly in your Lin script, for example:

```
// Registering LinExtension somewhere in your host app:
linExt = new LinExtension(page, nfcService, onMessageCallback)
```

```
// Then inside Lin script:
print("Hello, Lin!")
number = num("FF", 16)
// Should print "Decimal of FF is,255"
```

```

print("Decimal of FF is", number)
asciiText = ascii("48656C6C6F20")
print("Decoded ASCII:", asciiText)
// "Hello "
hexData = hex("ABC")
// "41 42 43"
print("Hex of 'ABC' is", hexData)
part = substr("LinExtension", 3, 5)
// "Exten"
print("substr:", part)
replaced = replace("2025-06-06", "-", "/")
print("date with slashes:", replaced)
// "2025/06/06"
found = find("LinExtension", "Ext")
// 3
print("found at index", found)
deco = bytes("A0B1C2D3", 1, 2)
// "B1C2"
print("slice bytes:", deco)

tlvObj = tlv("6F108407A0000000031010A5049F6501FF")
print(tree(tlvObj))
leaf = tlv_read(tlvObj, "6F", "A5", "9F65")
print("leaf value (hex):", leaf)

apduResult = apdu("00A404000E315041592E5359532E44444463031")
print("APDU response:", apduResult)
printc("#00FF00", "Success code:", apduResult)

```

Namespace Libs.TLV

Classes

[TlvObject](#)

Represents a node in a parsed TLV (Tag-Length-Value) structure. Provides access to nested TLV elements via an indexer and exposes the value of a primitive TLV element as a hexadecimal string.

[TlvParser](#)

[TreePrinter](#)

Provides functionality to generate an ASCII-art representation of a tree stored in a nested Dictionary<string, object> structure.