Bennet Hattesen

Slipmux for RIOT

Faculty of Engineering and Computer Science
Department of Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Contents

Abbreviations

CF. Configuration Frame. In Slipmux, a configuration frame is a SLIP encoded CoAP message that is prefixed with the Frame Type (FT) and postfixed with a checksum. 6, 9, 10

CRC-16. 16 bit wide cyclic redundancy check 5

CoAP. Constrained Application Protocol. Specified in RFC 7252. 6, 7, 10

DF. Diagnostic Frame. In Slipmux, a Diagnostic Frame is a SLIP encoded Diagnostic Message (DM) that is prefixed with the Frame Type (FT). 5

DM. Diagnostic Message. In Slipmux, a Diagnostic Message is a string of UTF-8 encoded characters without semantics except that it is intended to be human-readable. 5, 6

DT. Diagnostic Transfer. In Slipmux, Diagnostic Transfer describes the transmitting and receiving of a Diagnostic Frame (DF). 5, 6

FT. Frame Type. In Slipmux, there are three types of frames that can be exchanged. Diagnostic, Configuration and (IP-) Packets. They are identified by the first byte in the frame. 4, 5, 6, 9, 11

UART. Universal Asynchronous Receiver Transmitter 2, 3, 5

nanocoap. RIOT provides two CoAP implementations: GCoAP and nanocoap. The latter is the smaller one, but has fewer features. 6

no_std. A Rust directive to prevent loading the standard library. Colloquial used to describe bare metal, embedded system environments. 10

stdio. standard input/output stream 5

1 Introduction

Reliable and efficient data exchange between host computers and embedded systems is a requirement during development of Internet-of-Things (IoT) and cyber-physical applications. Even as high-speed wireless and Ethernet interfaces have become ubiquitous, Universal Asynchronous Receiver Transmitter (UART) communication — commonly referred to as serial communication — remains one of the most widely used mechanisms for connecting heterogeneous devices. UART offer simplicity, low cost, and broad hardware support, which makes it especially attractive in environments where resource constraints or legacy compatibility must be considered. However, the simplicity of UART at the physical and link layers places the burden of reliability, framing, and error handling on the higher-level protocol implementation. Despite the prevalence of UART-based links between embedded devices and host systems, the corresponding protocols are often either plain text (ASCII) streaming or application-specific.

Embedded operating systems such as RIOT OS exemplify these challenges. RIOT OS targets resource-constrained IoT devices and provides a lightweight multitasking environment with a small memory footprint [1]. These characteristics are advantageous for low-power constrained systems but also impose strict limitations on a communication stack design. In contrast, host computers typically operate under general purpose operating systems with abundant processing power, extensive libraries, and more permissive concurrency models. The diffrence in computational resources allows to externalise certain computations from the constrained device to the host system. Externalising computation reduces the resource requirements of the embedded application. This is especially desirable for tasks that are orthogonal to the main application running on the constrained system. For example, a configuration must be parsed and applied on the constrained device but creation and validation of that configuration can be offloaded to the host computer.

Slipmux is a protocol designed for serial communication [2]. It offers lightweight framing to encapsulate and transport three different types of data over an UART. As an iteration of the serial line IP protocol (SLIP) it maintains the same escaping scheme and the transport of IP packets [3]. The newly added types of data are plain text as UTF-8 strings and constrained application protocol (CoAP) [4] messages. This sets slipmux in an ideal position where it can aid to offload computation by providing machine-to-machine communication via CoAP, while not only maintaining backwards compatibility via plain text and SLIP, but also by remaining light on the resources it requires.

1.1 Related Work

This work is a continuation of the report [5] by the same author. The previous work motivated the need of change for traditional shells on constrained devices and outlined a possible future shell. In particular, it focussed on user experience (UX) and user interface (UI) enhancements when interacting with such a device. Both UX and UI are orthogonal tasks that can be externalized to the host computer in order to free up resources on the constrained device while also yielding better results. Slipmux has been implemented before by Lobaro [6] in C [7] and Golang [8] but both projects seem incomplete. In addition, the projects are abandoned with the last public code change in May 2020 [9].

1.2 Objective

The goal of this work is to provide infrastructure for future work. Specifically, it should enable building an application on top of it during the next project. This infrastructure consists of two implementations of slipmux. One implementation is a driver within RIOT and is written in the C programming language. The other is a higher level library in the Rust language, intended for development on general purpose computers running Linux or MacOS.

1.3 Outline

The remainder of the report is organised as follows. Section 2 introduces the technical background of communication via UART and outlines the key features of the slipmux protocol. Section 3 introduces the RIOT SLIP driver and its extension into a full slipmux driver, followed by Section 4 with technical background on the new Rust based slipmux library. Finally, Section 5 concludes with a summary of findings and directions for future work.

2 slipmux

Slipmux is a lightweight multiplexing approach for using a single UART interface to support diagnostic output, device configuration, and IP packet transfer on constrained IoT platforms [2]. Many experimental IoT boards offer low-cost serial ports which are

typically used either for firmware loading, human-readable input / output or SLIP-based IP packet transport (RFC 1055 [3]). Slipmux unifies these functions in a protocol that maintains compatibility with existing SLIP tools while adding well-defined framing for non-packet traffic.

At the physical and link layers, slipmux assumes a UART connection consisting of a TX-, an RX- and a GND pin. Further physical interfacing is out of scope for slipmux. It adopts the SLIP frame delimiter and escape mechanisms (`0xC0` and `0xDB`) without modification. Each slipmux frame begins with a single "initial byte" that identifies the Frame Type (FT):

- `0x45`–`0x4F`: IPv4 packets (retaining the first byte of the IP header)
- `0x60`–`0x6F`: IPv6 packets
- `0x0A`: UTF-8 diagnostic messages (ASCII-safe, newline-prefixed, unidirectional from device to host)
- `0xA9`: CoAP configuration messages with a 16-bit PPP-style CRC appended

Packet transfer frames remain bitwise identical to SLIP, ensuring immediate interoperability with existing utilities, such as tunslip [10]. Diagnostic frames provide human-readable debugging output multiplexed over the same link, while CoAP frames enable structured configuration and state retrieval. For CoAP, the draft defines the URI scheme `coap+uart` to address resources on serial devices and specifies that frames failing CRC validation must be silently discarded.

Slipmux does not implement session-level interleaving; messages are strictly sequential. However, it allows aborting a partially transmitted frame by sending the SLIP escape–END sequence (`0xDB 0xC0`). Unknown or unsupported initial bytes must be ignored to preserve forward compatibility.

Security considerations mirror those of the encapsulated payloads: diagnostic text should be sanitized to avoid terminal escape exploits, and CoAP exchanges provide no intrinsic protection unless supplemented with object security (RFC 8613) [11].

By combining diagnostic output, configuration, and packet forwarding on a single serial link, slipmux eliminates the need for separate interfaces or ad hoc command shells and aligns well with the minimalism and scriptability expected in constrained IoT environments.

3 The Slipmux Driver in RIOT

Slipmux uses the 16 bit wide cyclic redundancy check (CRC-16) IBM-SDLC to check for correctness of received frames. In preparation for the new slipmux driver, this CRC-16 subtype was added to RIOT on 16th June 2025 via #21552 [12]. RIOT already provides a SLIP driver [13], which could easily be extended to handle slipmux as well.

3.1 The SLIP Driver of RIOT

RIOT already provides a SLIP driver [13]. As SLIP is a predecessor to slipmux, this driver is a good starting point for the integration of slipmux. As the unescaping of the incoming data stream is done byte wise, a state machine is used that maintains the driver state in between the callbacks. For outgoing packets, the escaping is done on the fly as listed in Listing 1. The behaviour of the existing SLIP driver follows the scheme below:

A. Creates a network interface in the RIOT network stack.
B. Registers a callback for the UART receive interrupt.
C. That callback receives one new byte from the UART per call.
D. A state machine keeps track of the SLIP escaping and framing.
E. Data bytes for the incoming network packets are stored in a chunked ring buffer.
F. A fully received SLIP frame completes the current active chunk in the ring buffer.
G. The network interface is informed that a new packet is available.
H. Once the packet is processed in the network stack, the chunk gets released and is available for new data.

The SLIP driver contains an optional extension that follows slipmux and allows for Diagnostic Transfer (DT). It is used to provide regular standard input/output stream (stdio) over UART while SLIP is used.

The behaviour of the existing stdio extension can be described as below:

A. Registeres itself to RIOT as a stdio provider.
B. Encodes all stdout as Diagnostic Frame (DF).
C. Adapts the drivers state machine for the new Frame Type (FT).
D. Directly passes received data bytes to stdin (no intermediate buffer).

For outgoing DT, the extension reuses the on the fly escaping (Listing 1). Since this escaping is not FT aware, the Diagnostic Message (DM) gets prefixed with its FT first.

3.2 Extending the Driver for Configuration Messages

With SLIP and DT already implemented, only the processing of configuration messages is missing to fully implement slipmux. This breaks down to a few tasks:

A. Add the new FT to the state machine.
B. Allocate and manage memory for receiving and processing the new frames.
C. If the CRC is correct, extract the configuration message from the frame.
D. Process the configuration message.
E. Transmit the response, if any, after processing.

Adding a new FT to the existing state machine is straightforward as it repeats the existing pattern of the other two FTs. The final state machine is shown in Figure 2. It keeps track of the frames FT, that is currently decoded. This is necessary so the decoded data is passed to the correct consumer. For Configuration Frames (CFs) and packets, this is their respective chunked ring buffer. DMs are directly passed to the stdin pipe. When a CF is fully received, the decoder marks the chunk in the ring buffer as completed and then notifies the waiting processing thread via a thread flag. Afterwards it returns to the idle state. Similarly, when a packet is completed, the respective chunk is marked as completed too. In this case the waiting thread is the network interface and it is notified using a `netdev_event`. The resulting processing pipeline, in which minor behaviour changes depend on the frame type, is depicted in Figure 1.

As mentioned, incoming CFs are stored in a chunked ring buffer. This is identical to the handling of packets. Ring buffer have the useful property that their content is not copied nor moved during consumption. As such they are a common choice for buffering data streams. This ring buffer is chunked as it operates with variable length content, which fits storing CFs as their length is variable and not known in advance.

To process fully received configuration messages, the slipmux driver starts its own Constrained Application Protocol (CoAP) server. The server thread waits for the notification that a new configuration message is available in the ring buffer. It expects the data to be in the format of CoAP requests with the postfixed checksum as defined in [2] (see Section 2). The server first checks whether the checksum is correct and if it is, processes the CoAP request using the nanocoap backend, otherwise the request is silently dropped. The processing result is a CoAP response. The result is prefixed with the FT and postfixed with a newly calculated checksum. The prepared response is sent via the SLIP driver, which handles further esacping and framing (see Listing 1).

Even though the processing thread is a CoAP server, the network stack is not involved. This approach is convenient for devices that do not have other networking capabilities as it removes the heavy dependency of a network stack. On the other hand, this is wasteful for devices that already run their own CoAP server, which currently cannot be reused.

This extension got merged into RIOT on 7th July 2025 via #21418 [14].

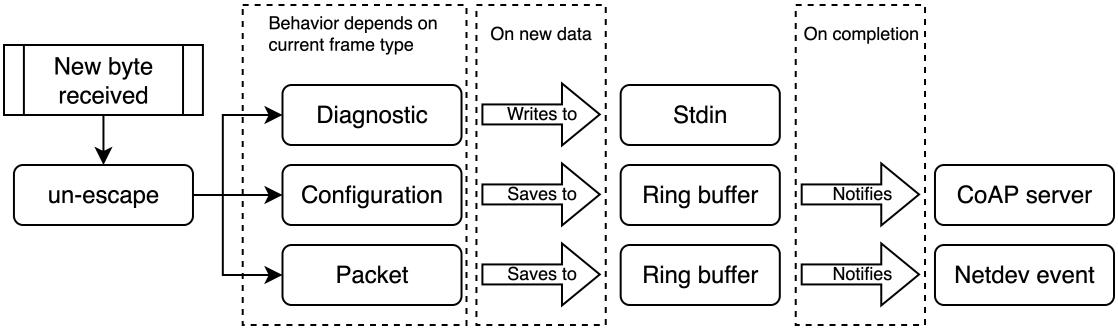

Figure 1: Abstract overview on the processing pipeline of the slipmux decoder in the RIOT implementation. When a new byte is received and feeded into the slipmux decoder it first is un-escaped (if neccessary) then, depending on the current frame type, it is written to its destination. For diagnostic data this is `Stdin`, Configuration and Packet data is saved into the respective ring buffer. Once a frame if fully received, the ring buffer is finished and the consumer of the buffer is notified.

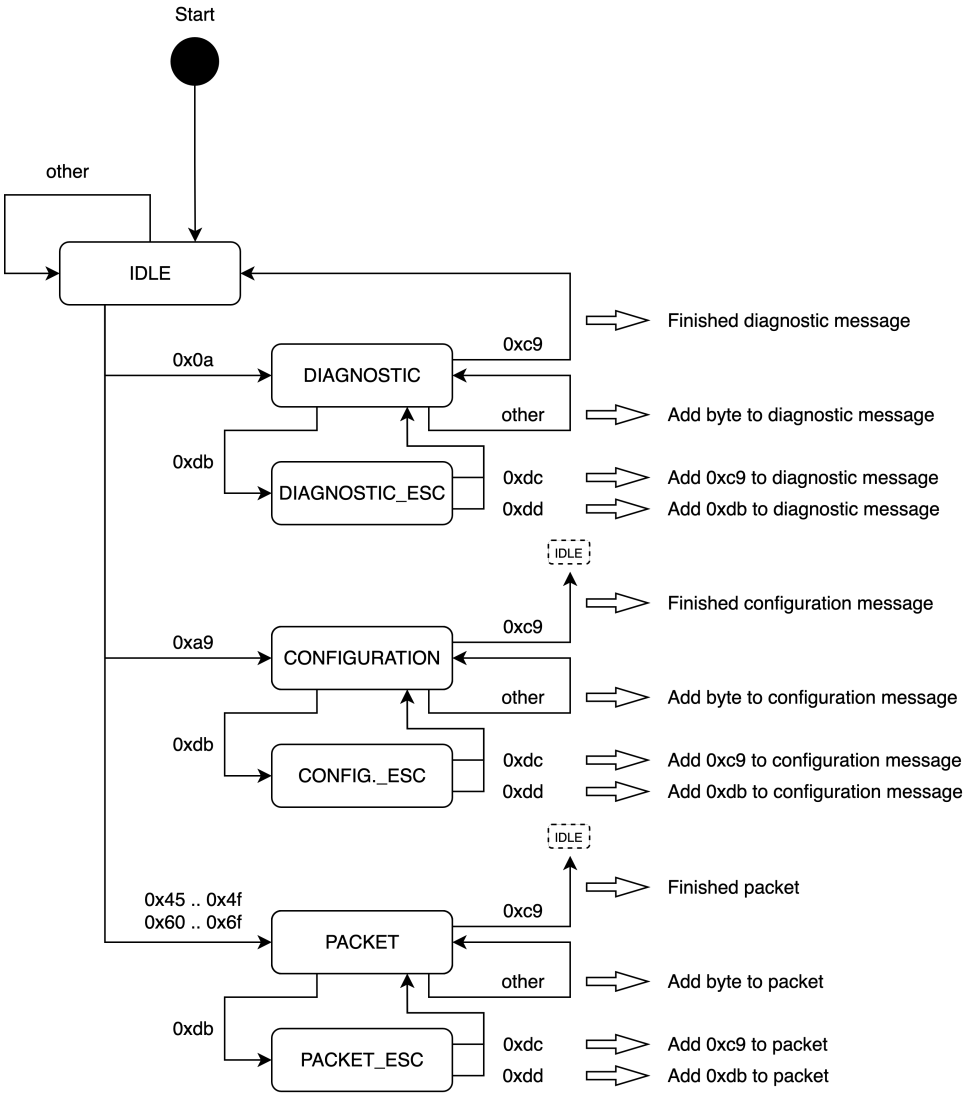

Figure 2: The state machine of the slipmux decoder used in the RIOT implementation. Transitions are stepped whenever a new byte is received. Each transition is guarded by the value(s) that the new byte might have. A transition annotated with 'other' matches all remaining byte values for the given state. The bytes `0x0a`, `0xa9` and the ranges `0x45..0x4f` and `0x60..0x6f` are the encoded FT. The byte `0xdb` starts an escape sequence which is followed by either `0xdc` (escaped End-Frame) or `0xdd` (escaped escape). Lastly, `0xc9` ends a frame. Illegal and error related transitions are omitted from this diagram. The outlined arrows hint the corresponding side effects for the application.

```
void slipdev_write_bytes(UART_t UART, const uint8_t *data, size_t len)
{
    for (unsigned j = 0; j < len; j++, data++) {
        switch (*data) {
            case SLIPDEV_END:
                /* escaping END byte*/
                slipdev_write_byte(UART, SLIPDEV_ESC);
                slipdev_write_byte(UART, SLIPDEV_END_ESC);
                break;
            case SLIPDEV_ESC:
                /* escaping ESC byte*/
                slipdev_write_byte(UART, SLIPDEV_ESC);
                slipdev_write_byte(UART, SLIPDEV_ESC_ESC);
                break;
            default:
                slipdev_write_byte(UART, *data);
        }
    }
}
```

Listing 1: Data is encoded for slipmux and transmitted in a single function. If a framing control byte is in the passed data, it gets replaced by the appropriate escape sequence. This encoding is done without awareness for the FT, the passed data must already be pre- and or postfixed according to their FT.

3.3 Remaining Work

- slipmux allows to abort frames during transmission on a byte boundary. This implementation does not support this. Neither on the receiving side nor the transmitting side.
- There is no API for sending CFs proactively. Sending of this FT is only implemented reactively, as a response to a received CF. It is unclear if such an API is needed.
- A shortcoming of the current integration into the existing SLIP driver is the inability to build the slipmux driver without the need to include a network stack. While a correct slipmux implementation must be able to receive IP packet frames, they can be discarded silently if the device has no interest in them. In such case, this implementation would still includ an entire IP network stack even if it is not used. This

is not a shortcoming of the driver but of the underlying buildsystem integration and configuration. As mentioned, the driver architecture already supports this.
- The processing of CFs runs in an extra thread, which directly utilizes the nanocoap-parser and -handler. This does not take an already running CoAP server into account. This might result in inconsistent behaviour of CoAP endpoints, depending on if they are accessed via the network or via slipmux. In addition, this wastes both storage and working memory.
- The upcoming Unicoap API (#PR21582) can be used to resolve the CoAP server integration issues [15]. Once Unicoap is available, its usage for the slipmux driver will be investigated.

4 Slipmux as a Crate

In the upcoming main project, an application (Jelly [16]) gets developed that requires a slipmux driver as well. Instead of developing that driver inside the application, the driver is externalized into a library of its own, hostet on Teufelchen1/slipmux [17]. The library is written in Rust and was initially published on 4th April 2025 in version 0.1.0. The current version available is 0.3.2, published on 18th July 2025. The Rust community calls libraries “Crates”. Hence this driver is a crate called “slipmux” and is available for other Rust developers on crates.io [18]. Rust has the generation of documentation from code comments and examples as a build-in feature. This includes the compilation, execution and testing of these documentation examples as well. The extensive documentation on the usage of this driver gets published automatically on docs.rs/slipmux whenever a new driver version is released [19]. The development of the driver is almost completed and all required functionality from the slipmux draft is implemented. Only minor documentation enhancement and API polishing is left to do. An extra feature of the driver is that it cannot only be used on conventional computers but also on embedded systems (bare metal). Such systems, where the Rust standard library (“std“ or “stdlib”) is not available, are called no_std (which is also the name of the compiler directive to omit the standard library) targets by the Rust community. Therefore the slipmux crate is no_std compatible.

4.1 Software Architecture

The crate is split into encoding and decoding. Encoding can be done stateless while decoding is much more involved as it requires splitting a stream of encoded data into an unknown count of frames and returning their respective unencoded data.

4.1.1 Encoding

The encoding is stateless and only provides functions that encode data into slipmux frames. Encoding can either be done using a user provided buffer (for `no_std` targets, using no heap operations) or on the heap, returning a new buffer.

4.1.2 Decoding

Because of the streaming nature of decoding, the user has to create a `Decoder` object that keeps track of the decoding state. The state is managed using a state machine. This state machine is functional identical to the one used by the RIOT driver shown in Figure 2. The decoding is done bytewise by calling the decode method on the object with the new byte from the input stream and a `FrameHandler`. The `FrameHandler` is another object the user must create, either using one of the provided ones or by implementing their own.

4.1.3 Framehandler

The `FrameHandler` component is used to decouple memory management from the decoding logic. It enables to choose where to store the decoded data depending on the FT. This allows the user fine grained control over the memory usage, which is important when targeting embedded devices. On those devices, memory usage is often interleaved between different functionalities. For example, this approach enables users to build a custom `FrameHandler` that stores packets directly in the network stack, reducing the overall memory required and cut down on expensive memory copy operations. When implementing their own, the user must satisfy the `FrameHandler` trait listed in Listing 2. A Rust trait is an interface, which communicates and defines if a type (e.g. a Struct) has a specific behavior [20]. It does not define data, neither constant nor mutable. The `FrameHandler` decouples by providing an excerpt of all the state transitions that might happen during decoding. This subset consists of three events: beginning a new frame of a specific type, adding a new data byte and finishing a frame (including error report, if any). The events happen in this exact order everytime a frame is decoded, with adding new data being the only event that can happen zero or more times before moving to the

next event (ending the current frame). One possible sequence of these events is shown in Figure 3.

```rust
/// Callback handler for the decoder
///
/// This is typically driven by [`Decoder::decode()`], which calls it strictly
in the sequence of
/// [`.begin_frame()`][Self::begin_frame()], any number of [`.write_byte()`]
[Self::write_byte()]
/// and then [`.end_frame()`][Self::end_frame()], starting over after that.
pub trait FrameHandler {
    /// Called when the decoder identifies a frame and starts filling it
    fn begin_frame(&mut self, frame_type: FrameType);

    /// Called with each new byte that belongs to the current frame
    fn write_byte(&mut self, byte: u8);

    /// Called when a full frame has been received
    fn end_frame(&mut self, error: Option<Error>);
}
```

Listing 2: The `FrameHandler` trait that decouples decoding from memory management.

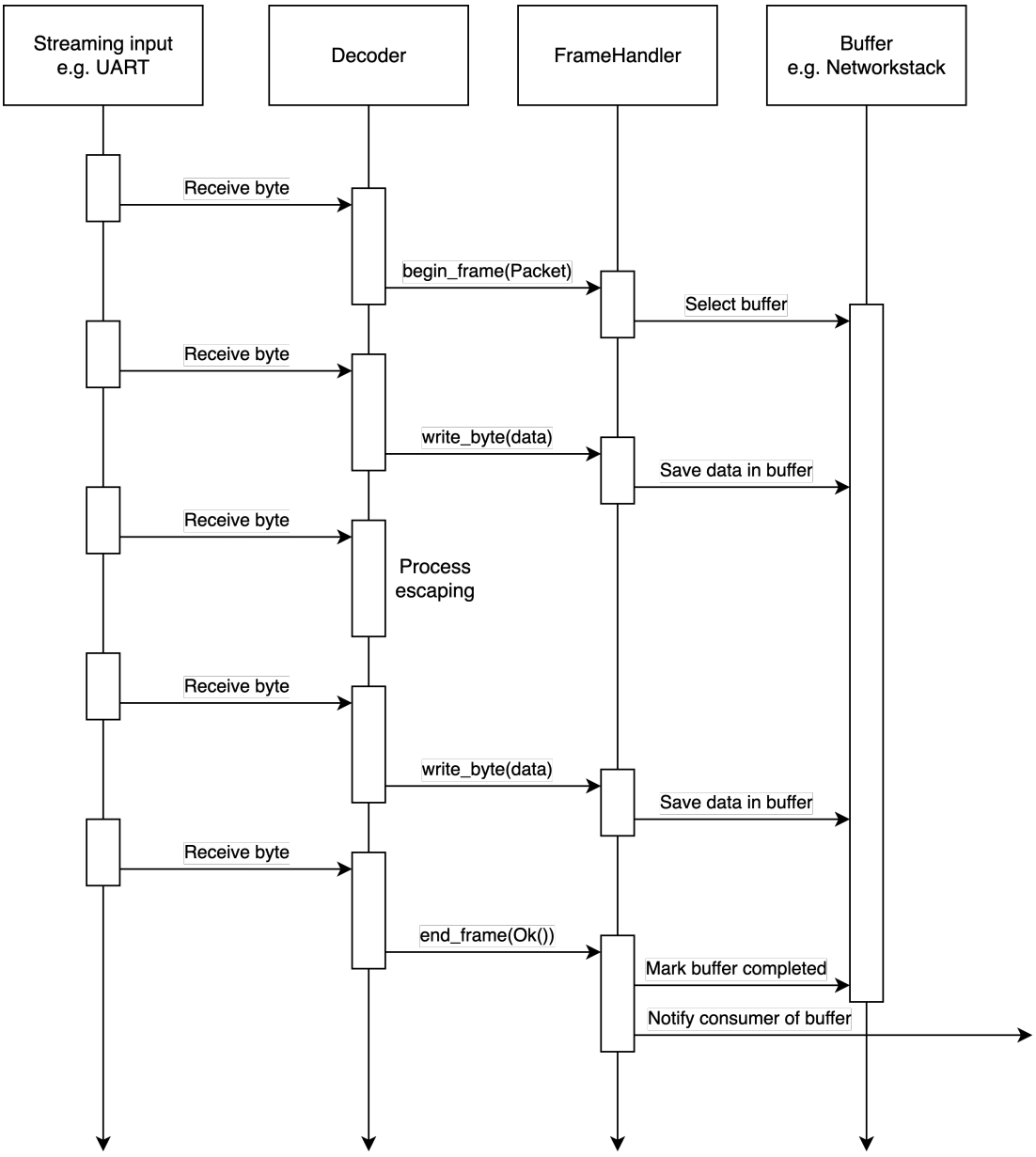

Figure 3: The interactions between the data input, the `Decoder` and the `FrameHandler` over time. The first incoming byte starts a new frame. The seconds byte is pure data and gets saved. The third is an escape character and the fourth is the escaped data byte. Lastly, the fifth byte ends the frame.

5 Conclusion and Outlook

This report documents the successful extension of the RIOT SLIP driver into a slipmux driver and the creation of a new slipmux library. It outlined that both the RIOT slipmux driver and the slipmux crate are in a good shape and can be used for future work. The driver can benefit from further work such as buildsystem and configuration clean up. The CoAP server situation is non-ideal, but is functionally solid and can be replaced by future RIOT APIs such as Unicoap [15]. The Rust crate is fully functional and ready to be used in applications. The implementations provided will serve as a solid foundation for future development of tooling that make use of the features provided by slipmux. In particular, a RIOT shell that externalizes the UX and UI computation is a prime target for further research as described in [5]. Outside of user interactions, the possible gains in performance and code size are of high interest.

Bibliography

[1] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, Dec. 2018, [Online]. Available: http://doi.org/10.1109/JIOT.2018.2815038

[2] C. Bormann and T. Kaupat, "Slipmux: Using an UART interface for diagnostics, configuration, and packet transfer," Internet Engineering Task Force, Nov. 2019. [Online]. Available: https://datatracker.ietf.org/doc/draft-bormann-t2trg-slipmux/03/

[3] J. Romkey, "Nonstandard for transmission of IP datagrams over serial lines: SLIP," RFC Editor, Jun. 1988. doi: 10.17487/RFC1055.

[4] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC Editor, Jun. 2014. doi: 10.17487/RFC7252.

[5] B. Hattesen, "Jelly, a Modern Shell for Constrained Devices." Accessed: Oct. 29, 2025. [Online]. Available: https://www.inet.haw-hamburg.de/teaching/ss-2024/project-class/fw1_bennet_hattesen.pdf

[6] "Lobaro GmbH." Accessed: Oct. 29, 2025. [Online]. Available: https://www.lobaro.com/

[7] "util-slip: C Slip RFC10-55 and SlipMux implementation in C." Accessed: Oct. 29, 2025. [Online]. Available: https://github.com/lobaro/util-slip/

[8] "Slip & slipmux: Implementation of SLIP (rfc-1055) in GoLang." Accessed: Oct. 29, 2025. [Online]. Available: https://github.com/Lobaro/slip

[9] "Latest public commit to util-slip by lobaro." Accessed: Oct. 29, 2025. [Online]. Available: https://github.com/lobaro/util-slip/commit/3495959bd3484aa46755a2e7219863e932e84f24

[10] "tunslip by Adam Dunkels in contiki OS." Accessed: Oct. 29, 2025. [Online]. Available: https://github.com/contiki-os/contiki/blob/a4206273a5a491949f9e565e343f31908173c998/tools/tunslip.c

[11] G. Selander, J. P. Mattsson, F. Palombini, and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)," RFC Editor, Jul. 2019. doi: 10.17487/RFC8613.

[12] “RIOT OS Pull request #21551: checksum: Add crc16-fcs / IBM-SDLC.” Accessed: Oct. 29, 2025. [Online]. Available: https://github.com/RIOT-OS/RIOT/pull/21552

[13] “Slip driver in RIOT OS.” Accessed: Oct. 29, 2025. [Online]. Available: https://github.com/RIOT-OS/RIOT/tree/master/drivers/slipdev

[14] “RIOT OS Pull request #21418: drivers/slipmux: Add to RIOT.” Accessed: Oct. 29, 2025. [Online]. Available: https://github.com/RIOT-OS/RIOT/pull/21418

[15] C. Seifert, “RIOT OS Pull request #21582: net/unicoap: Unified and Modular CoAP stack: Messaging and Minimal Server (pt 2).” Accessed: Oct. 29, 2025. [Online]. Available: https://github.com/RIOT-OS/RIOT/pull/21582

[16] Bennet Hattesen, “Jelly.” Accessed: Oct. 29, 2025. [Online]. Available: https://github.com/teufelchen1/jelly

[17] B. Hattesen, “Slipmux library.” Accessed: Oct. 29, 2025. [Online]. Available: https://github.com/teufelchen1/slipmux

[18] B. Hattesen, “Slipmux crate.” Accessed: Oct. 29, 2025. [Online]. Available: https://crates.io/crates/slipmux

[19] B. Hattesen, “Documentation of the Slipmux crate.” Accessed: Oct. 29, 2025. [Online]. Available: https://docs.rs/slipmux/lastest/slipmux/

[20] Steve Klabnik and Carol Nichols, with contributions from the Rust community, “Traits: Defining Shared Behavior (The Rust Programming Language).” [Online]. Available: https://doc.rust-lang.org/book/ch10-02-traits.html