Bennet Hattesen

Jelly: A new Shell for RIOT OS

Faculty of Engineering and Computer Science
Department of Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Contents

1 Introduction **1**
1.1 Related Work 1
1.2 Objective 2
1.3 Outline 2

2 Shell Commands in Jelly **3**

3 Requirements **4**
3.1 Ease of Use 4
3.2 Backwards Compatibility 4
3.3 Low Barrier for Implementing Commands 5

4 Design of Jelly **6**
4.1 Adhere to Ease of Use 6
4.1.1 Display 6
4.1.2 Input 6
4.2 Adhere to Backwards Compatibility 7
4.2.1 Output in the TUI 7
4.2.2 Input in the TUI 7
4.2.3 Transparent Pass-through Mode 8
4.2.4 Migration Path 8
4.3 Adhere to Low Barrier for Implementing Commands 8
4.4 Events 9
4.5 Managing Threads and Reactivity 9

5 Implementation of Jelly **11**
5.1 Event Loop 11
5.2 Application Logic 11
5.3 Commands 12
5.4 Command Handler 13
5.4.1 Shortcomings 16
5.5 Input Handling 16
5.6 Terminal User Interface 17
5.7 Pass-Through Modes 18
5.8 Command Line Interface 19

6 Evaluation **21**
6.1 Ease of Use 21
6.2 Backwards Compatibility 22
6.3 Low Barrier for Implementing CoAP-based Commands 23

7 Conclusion and Outlook **27**

Bibliography **28**

1 Introduction

The embedded operating system RIOT 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 and more permissive concurrency models. The difference in computational resources allows to partially externalize computations from the constrained device to the host system. Externalizing computation reduces the resource requirements of the embedded application. This is especially desirable for the RIOT shell, where text string operations such as argument parsing and data conversion are common.

Slipmux is a protocol designed for serial communication [2]. It offers lightweight framing to encapsulate and transport three different types of data over an Universal Asynchronous Receiver Transmitter (UART). As a successor 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 a position for aiding 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.

In this work we developed a tool to externalize the RIOT shell via slipmux.

1.1 Related Work

The work in this report is a continuation of the reports [5] and [6] by the same author. The first report 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. The second report summarized the implementation of two slipmux drivers, which are necessary infrastructure for building this new kind of shell. One driver targets Linux / macOS

while the other is integrated into the RIOT OS. Together those reports provide the foundational building blocks for this work.

In 1965, Louis Pouzin coined the term shell [7]. He defines a shell as a default procedure that is automatically run when a user types in some message into their terminal at a time where no other procedure is currently running. The shell is the local operating system interface between terminal messages and the procedures run by the shell. It offers the ability to call any command by an entry name.

1.2 Objective

The goal of this work is to build an application that implements this new type of shell. The result is not a proof of concept, but the implementation of a usable early-stage prototype. Its purpose is to uncover practical caveats and obstacles, and to extrapolate the requirements and difficulties that a production-grade implementation will face. This application is called Jelly and should work seamlessly with RIOT OS [1].

1.3 Outline

This chapter introduced the context of Jelly, reviewed the related work and stated the objectives of this work. The remainder of this report is structured as follows:

Section 2 highlights what differentiates the shell commands in Jelly from the commands in traditional shells. It also introduces the three distinct types of commands that Jelly provides. Section 3 derives the functional and non-functional requirements from the stated objectives. Based on those requirements, Section 4 describes the derived component design and provides an overview of the internal architecture of Jelly. To put this design into practice, Section 5 walks through the implementation of the key modules and the technical decisions behind them. To verify that the implementation meets the stated requirements, Section 6 discusses and evaluates an exemplary usage of Jelly. Lastly Section 7 concludes the work by discussing the remaining challenges and providing an outlook for future development.

2 Shell Commands in Jelly

The nature of a command in the context of Jelly differs from the definition by L.Pouzin [7, R-1, I]. We share that a command is initiated from the terminal, by typing a message to the shell. We differ in the assumption that a command is a program. For us, a command is a sub procedure that architecturally resides inside the shell application. Pouzin states that no additional subsystem is needed to interpret the user message and that the user message is only lightly pre-processed before handing it as an argument to the called command. This remains true for Jelly as well. Sub procedures receive the interpreted message as their arguments, where the first argument is the name of the sub procedure. The transformation from message to list of arguments is performed by interpreting any blank gap as a delimiter of arguments.

In traditional shells, commands can call other commands, providing (data) input to them and process their output. This is desirable as the nature of a call from within a command and directly from the user differs but going through the same shell interface ensures identical behavior (as opposed to a direct library function call). However, Jelly currently does not support, nor aims to be able to support this at the moment. This is due to non technical reasons, for example to maintain a smaller scope for this work.

Furthermore, Jelly has to differentiate between three types of commands: Commands that are not executed by Jelly but transparently passed through to the RIOT shell, commands that are implemented in and executed by Jelly and Jelly internal commands. The first type is a result of the backwards compatibility requirement to keep the original RIOT shell functioning (see Section 3.2). In transport those commands are handled as plain text (diagnostic) frames by slipmux. The second type builds on top of the ability of slipmux to transport CoAP messages. These commands are implemented within Jelly and communicate with RIOT using CoAP. In this document, we call this second command type "CoAP-based command" to clearly differentiate it from the first command type.

The third and last type are Jelly-internal commands, that alter settings or behaviors of the application itself. As those are self-contained they do not interact or affect the connected RIOT device directly. For the purpose of this work, this type of commands are ignored as out of scope.

3 Requirements

In order to design the internal architecture and external interfaces of Jelly, it is first necessary to define what is needed and why.

3.1 Ease of Use

Jelly should be easy to use. It should automate repetitive tasks and communicate its operative state clearly to the user.

1. Show connection status
2. Show basic, frequently needed information
3. Show only what is needed
4. Expose user input options
5. Follow the line-editing conventions established by GNU Readline and common POSIX shells [8]

3.2 Backwards Compatibility

RIOT OS already provides a string-based shell. It is unfeasible to replace this existing shell with a new one in a single deployment. Because the existing shell is deeply integrated and widely depended upon, a single deployment replacement would couple the migration of all consumers to the correctness of the new implementation in one step. A parallel approach with backward compatibility instead allows migrating consumers incrementally and enables a rollback at every stage. Therefore, Jelly must be build in a way that allows for a continuous process of extension and integration into RIOT, without interrupting the status quo.

1. Existing shell commands remain available
2. Logging is displayed as before (`printf` remains functional)
3. Changes to the RIOT codebase are limited to feature extensions and refactoring. Removing existing features is not acceptable
4. The existing tooling used to interact with the RIOT shell seamlessly work using Jelly

3.3 Low Barrier for Implementing Commands

Jelly's usefulness is determined by the set of CoAP-based commands available for it. Jelly itself is only the platform they are built on. It is therefore essential that implementing a new command is quick and straightforward. The effort must scale with the command's business logic, not with tool-specific boilerplate, since high implementation cost would deter potential contributors and, in consequence, adopters.

The CoAP-based commands are build into Jelly and as such are executed on the developers host computer. They use CoAP for interacting with the connected node. In the context of RIOT, this means that the newly needed CoAP endpoints are implementable with internal modules and do not require third party libraries or any other Jelly specific dependencies. This requirement is relaxed for CoAP itself, as RIOT already supports multiple CoAP libraries. Therefore the barrier for implementing the newly needed CoAP endpoints is low, as it is the same process as implementing any other, non Jelly related, endpoint.

4 Design of Jelly

The operational context of Jelly is highly reactive. At any time, a user may input data, the properties of the terminal window might change, a new message from RIOT might arrive, the device might get disconnected. In addition, some actions like sending data to the RIOT device can be slow or even blocking operations. This context places additional constraints on the software architecture to ensure that Jelly is a responsive application.

4.1 Adhere to Ease of Use

The requirements of Section 3.1 concern two distinct aspects of the user interface: the presentation of information to the user, and the acquisition of user input. Since the two aspects differ fundamentally in their technical characteristics, they are treated separately in the design.

4.1.1 Display

All user-facing output is rendered through a text-based user interface (TUI). The TUI fulfills the displaying requirements: It shows the connection status, presents frequently needed information at a glance, and omits everything not required in the current context, following the principle of minimal necessary presentation.

Rendering must be performed by a separated thread. This is necessary because the other components of the application run concurrently and produce output at unpredictable times; a separate thread allows the display to be updated coherently without blocking, or being blocked by the rest of the application.

4.1.2 Input

In contrast to output, keyboard, mouse and window event input is inherently reactive and blocking: The application must wait for the user to press a key before it can react, and each keystroke must be processed before the next can be interpreted meaningfully. Input handling is therefore assigned to its own thread, decoupled from both the rendering thread and the application logic.

As we anticipate to make use of hotkeys to change settings or alter the viewing state of the TUI, we may not be able to implement the line-editing within the input thread. Instead, it is beneficial to move this responsibility to the general application logic to gain

access to context relevant information. For example features such as input history or auto complete might need additional information.

4.2 Adhere to Backwards Compatibility

The requirements of Section 3.2 demand that the application integrates into RIOT without interrupting the status quo. This section shows that backwards compatibility reduces to a single design requirement: The application must be transparent with respect to the existing text-based communication. It must reproduce the RIOT text output faithfully and forward user input unmodified when needed; nothing in the communication path may alter the observable behavior of the existing shell.

4.2.1 Output in the TUI

The RIOT shell communicates through string-based messages, as produced for instance by `printf`. Jelly therefore does not interpret or restructure these messages at the protocol level; it renders them verbatim. This guarantees that logging behaves exactly as before, satisfying the requirement that `printf` remains functional, and that the output of existing shell commands is displayed unchanged.

Where the TUI presents such messages (embedding them in its layout) it does so strictly without modifying the messages intention. Care must be taken around whitespace and control characters which may interfere with the TUI rendering. Those characters will need to be removed or replaced without altering the messages intention. For example, tabulators may be replaced as multiple spaces. The requirement of limited codebase changes is respected: RIOT requires no changes to its messaging behavior.

4.2.2 Input in the TUI

User input is likewise forwarded as-is. Lines entered by the user are passed through to the traditional RIOT shell unmodified. The input handling described in Section 4.1.2 (line editing, history) affects only how the line is composed, not what is ultimately delivered. Existing shell commands therefore behave identically whether invoked through the old tooling or through Jelly. One exception applies: Should the input match a CoAP-based command, the input is not sent to RIOT but instead is processed by Jelly, since those commands are implemented within Jelly. See Section 5.3 for the command processing.

4.2.3 Transparent Pass-through Mode

The RIOT shell is sometimes used by software tools. Theses tools interact with the RIOT shell in a manner analogous to human users, by sending text commands as the raw input and parsing the response from the raw output. The task of interacting with the serial interface is outsourced to yet another application. By default this is `Pyterm`. In order to use the tools with Jelly the default application for the serial interface is set to Jelly instead.

Using this existing tooling together with Jelly imposes one further requirement: Because that tooling operates on the raw input and output text stream from the RIOT device, rendering the output from RIOT verbatim into a TUI is not acceptable for this tooling. Elements of the TUI would be confused as output from the RIOT device. The Jelly application therefore provides a transparent pass-through mode in which the TUI is disabled entirely. Data is relayed byte-for-byte between RIOT and standard input and output. In this mode Jelly is behaviorally indistinguishable from a raw connection (without slipmux) to the node, which allows it to seamlessly interact with that tooling: scripts and pipelines built on the text interface continue to work without modification, and the migration required of consumers is reduced to substituting one executable for another.

4.2.4 Migration Path

Together, these properties realize the incremental migration demanded by Section 3.2: existing commands remain available, logging is unchanged, and Jelly can be introduced alongside the existing tooling, with a rollback to the previous setup possible at every stage.

4.3 Adhere to Low Barrier for Implementing Commands

To adhere to the requirement of a low barrier to entry, Jelly's internal API must be flat and small: a programmer implementing a command should need to understand only a handful of functions, and each of them should be usable without studying its interactions with the rest of the API, minimizing their cognitive load. Further, the API should abstract the asynchronous nature of CoAP requests and responses. Again, not having to account for blocking and failing interaction lowers the barrier to implement a command. Since RIOT is written in the C language, we anticipate that most RIOT developers only have

limited Rust knowledge and experience. It is therefore desirable to limit the API in terms of Rust feature usage. For example, being able to implement a command without needing to know about async-Rust or complex lifetimes reduces the amount of Rust a developer needs to know. Lastly, Jelly should come with documented example commands that can be duplicated and trivially adjusted to fit new needs.

4.4 Events

External interfaces are handled by individual threads. Those threads need to communicate with each other the state changes they experience or they might need to pass on the data they received. They do this by sending events to each other. Events have a distinct data type. Some event types carry additional data payloads which also have a distinct type. The combination of an event type with (optional) data makes them an algebraic data type, which in Rust can be defined as an enumerated type.

We expect distinct events for receiving each of the three slipmux frame types (Diagnostic, Configuration, Network packet) as well as for sending them. We also expect various events to communicate changes to and user input from the terminal. Lastly, we need two events for connecting and disconnecting the constrained device to or from the host.

4.5 Managing Threads and Reactivity

We expect multiple threads which need to be able to exchange messages or events. A common approach to this is an event loop in a "Reactor Pattern" [9]. The loop waits for an event (message) to arrive and then dispatches it to the appropriate handler. Events are send to the event loop by other concurrently running threads. For Jelly there are multiple sources of events, therefore multiple threads are needed: the input thread when new data arrives via Stdin and the slipmux reader thread when a new slipmux frame is received. Another thread is needed for sending data to the attached device. The interactions and relations of these threads is depicted in Figure 1.

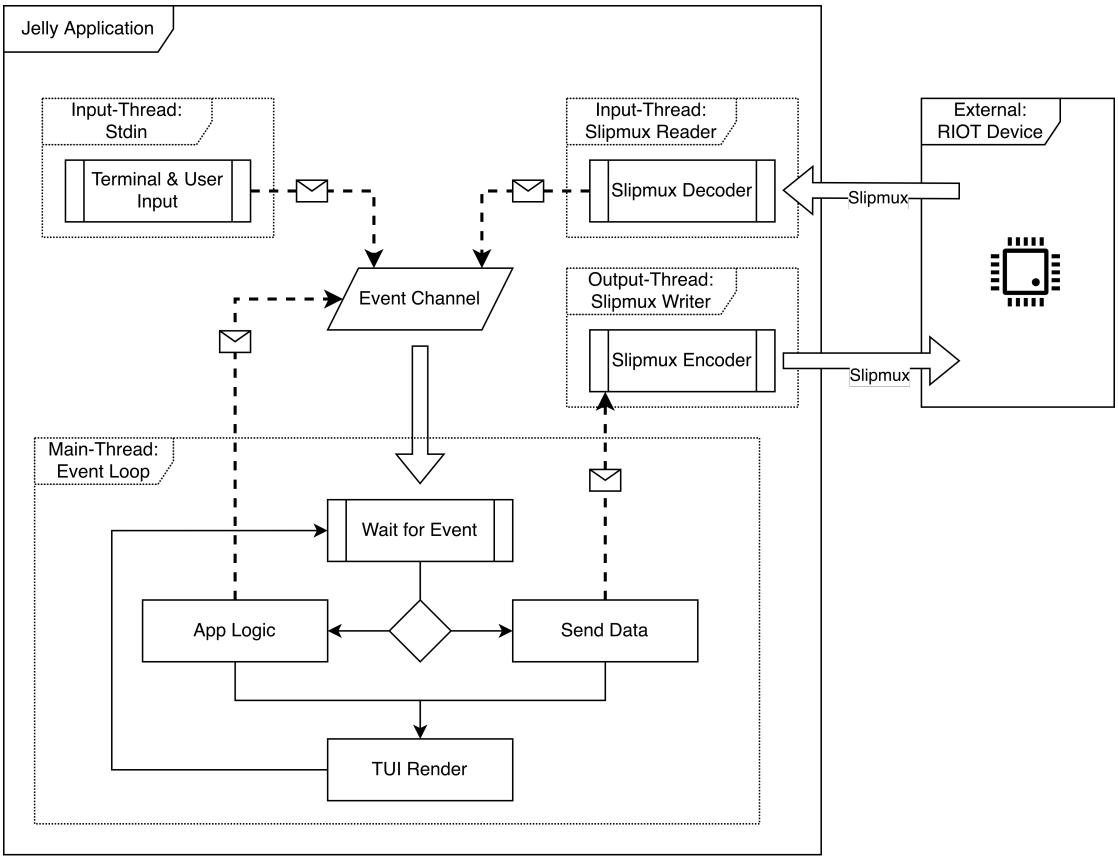


Figure 1: Abstract overview on the reactor path of Jelly when using the TUI. Four threads are depicted: the main event loop, user input and slipmux reader / writer. The center piece is the event loop which awaits events and acts on them by either passing messages to the application logic via method calls or by sending messages to the slipmux writer thread via a message queue. Finally it redraws the terminal user interface. Events are send to the event loop by the user input thread when new data arrives via Stdin, by the slipmux reader thread when a new slipmux frame got decoded and by the application logic. In case Jelly is used in a pass-through mode, the event loop gets replaced by a specialized version.

5 Implementation of Jelly

Jelly is written in Rust. We chose Rust because it offers temporal and spatial memory safety in addition to its robust data-race free concurrency management.

5.1 Event Loop

Events are sent and received via the multi-producer, single-consumer FIFO queues that are available in the Rust standard library as `std::sync::mpsc`. The main event loop begins by awaiting events on the central queue. We defined the events in the algebraic datatype `enum Event` as listed in Listing 1. The dispatching is achieved using Rusts `match` statement. Finally, before starting over again, the event loop checks if the content of the terminal user interface (TUI) has changed and if so, performs re-rendering.

```
pub enum Event {
    Diagnostic(String),
    Configuration(Vec<u8>),
    Packet(Vec<u8>),
    SendDiagnostic(String),
    SendConfiguration(Vec<u8>),
    SendPacket(Vec<u8>),
    SerialConnect(String),
    SerialDisconnect,
    TerminalString(String),
    TerminalKey(KeyEvent),
    TerminalMouse(MouseEvent),
    TerminalResize,
    TerminalEOF,
}
```

Listing 1: The definition of global events as a Rust enum. Threads send and receive these events to / from the main event loop.

5.2 Application Logic

The application logic for Jelly is mostly encapsulated as reactions to certain events by matching handlers to them. For example, when the user presses the key `F2` the event

`Event::TerminalKey(..)` is send to the event loop which then dispatches this to a call to the application logics handler `on_key(..)`. Similarly, those handlers are used to react to incoming slipmux frames.

5.3 Commands

The traditional text based shell commands together with Jelly internal commands and the new CoAP-based commands have common behavior with regards to the command line interface that Jelly provides. This shared behavior is encapsulated by the `Command` structure in Listing 2.

This structure needs to be instantiated once for each known command. It stores the name of the command (which is what the user has to type-in to run this command), a textual description of itself, an array of CoAP endpoints that are required by this command (if applicable) and a parser function.

The parser function is run by Jelly when the user types-in and submits the name of the command with or without additional arguments. The user input is passed to the parser function which then parses the command line arguments. If the user input is a valid command line, the parser constructs the applicable `CommandType` as the result. Otherwise an error message is returned which Jelly displays to the user. The `CommandType` encapsulates if the parsed command is a traditional text command, Jelly internal or a CoAP-based command. If the result is a `CommandType::Text(String)` (a traditional text command) Jelly will pass the contained string directly to the connected node via a diagnostic message. This is used to provide backwards compatibility with the existing RIOT shell. A result of `CommandType::Jelly` indicates an internal command that should change settings of the running Jelly application. The last possible `CommandType` is `CoAP(Box<dyn CommandHandler>)`. This type corresponds to the new CoAP-based commands and the payload is a handler that can send CoAP requests and react to responses, see Section 5.4.

```rust
/// The type of the command, Jellys behaviour towards that command depons on
this
pub enum CommandType {
    /// A raw diagnostic text message, will be send as is to the device
    Text(String),
    /// A command that does CoAP interactions
    CoAP(Box<dyn CommandHandler>),
    /// A Jelly internal command, e.g. Help
    Jelly,
}

/// Represents a command that the user can type into jelly
pub struct Command {
    /// The name of the command, this is what the user types into jelly.
    pub cmd: String,

    /// A description of what this command will do.
    pub description: String,

    /// The CoAP end-point(s) this command requires, if any.
    pub required_endpoints: Vec<String>,

    /// Parses a cli string.
    ///
    /// On success, returns a `CommandType`.
    /// On error, returns a human readable usage error.
    pub parse: fn(&Self, args: &str) -> Result<CommandType, String>,
}
```

Listing 2: The enum which differentiates the command types and the struct that describes an individual command.

5.4 Command Handler

CoAP-based commands are expected to implement the `CommandHandler` trait in Listing 3. This trait defines a set of behaviors that Jelly invokes to run the command while

abstracting the inner and outer interactions. The callback nature of this API shields commands from the asynchronous manner of communication via CoAP.

First the `init()` functions is invoked which return the first CoAP request that this commands needs to perform. Jelly sends this request via slipmux to the device. Using the `want_display()` functions Jelly can check if there is content that the command wants to display to the user. If so, the `display()` function gets invoked in which Jelly passes a mutable buffer into which the command can write arbitrary UTF-8 text. This buffer is then rendered by Jelly. Once the CoAP response arrives, Jelly passes it to the waiting command via the `handle()` function. The command may return another CoAP request and repeat the cycle. At any point, Jelly might ask the command if it has completed via the `is_finished()` function. Sometimes the user might indicate that the final result of the command is not rendered in a human readable way but instead in a machine readable form. In that use case, Jelly uses the `export()` method instead of the `display()` method.

```rust
/// Callback API for handling a command.
pub trait CommandHandler {
    /// This function is called exactly once. It is always the first call for
any handler.
    /// Returns a coap request that is send to the attached device.
    fn init(&mut self) -> CoapRequest<String>;

    /// Called everytime a response is found to the last request this handler
has sent.
    fn handle(&mut self, _payload: &Packet) -> Option<CoapRequest<String>> {
        None
    }

    /// Asks the handler if it wants to display anything. Usually called after
a response was
    /// processed.
    fn want_display(&self) -> bool {
        false
    }

    /// Asks if the handler has completed its task and might get removed from
the tracking.
    fn is_finished(&self) -> bool {
        true
    }

    /// Provides a buffer into which the handler can write the result.
    fn display(&self, _buffer: &mut String) {}

    /// Asks the command to provide a 'reasonable' binary export of its content
    fn export(&self) -> Vec<u8> {
        let mut buffer = String::new();
        self.display(&mut buffer);
        buffer.as_bytes().to_vec()
    }
}
```

Listing 3: The interface that all CoAP-based commands must implement. The methods are called by Jelly upon events.

5.4.1 Shortcomings

1. Timeout: When a command sends a CoAP request but the response never arrives, the command will get stuck at waiting indefinitely as there is no way for Jelly to remedy the situation. Ideally, a configurable timeout would be used after which Jelly can inform the command that no response has been received. This is planned for the next iteration.
2. Inflexible: Commands can only implement tasks that can be achieved within the boundaries of the `CommandHandler` trait. This mostly concerns interactivity with the user and limits commands to be one-off (can not be used interactively by the user).
3. Only one request per time: Commands can only dispatch one CoAP request during initialization and additional one after processing the response of the previous request. This might result in a performance penalty when multiple responses might be needed simultaneously.
4. Timing (periodic commands): Commands cannot implement behaviors that depend on time. For example, it is not possible to implement a command that writes `HelloWorld` to the user every three seconds.
5. Limited rendering: Jelly offers a rich terminal user interface and it is beneficial to use it for displaying the results of commands in a way that utilizes its advantages. Instead, commands are forced to either export machine readable binary or print into a string buffer. It is desirable to offer the rendering capabilities of Jelly to the individual commands. For example, a data logging command might whish to render a table directly to the TUI.

5.5 Input Handling

Handling of the users text input is done in the `UserInputManager`. This manager keeps track of the current cursor position, inserts and deletes characters as well as maintaining a history of last commands and provide suggestions on what the user might try to type. The suggestions work by looking up which known commands in the `CommandLibrary` are prefixed with the current user input buffer. For example, if there are two known commands: `Foo` and `FooBar`, and the user types in `Fo`, the `UserInputManager` will suggest to auto complete the input with `Foo` as the first choice, followed by `FooBar` as the second option.

Since this behavior is not strictly Jelly specific, there is an option to factor-out the manager into its own library.

5.6 Terminal User Interface

The terminal user interface (TUI) is build using the third-party framework Ratatui [10]. It provides an immediate-mode rendering API, in which the client application has to issue each frame directly. We chose to use Ratatui as it is well established, widely used, supports dynamic layouts and is lightweight. We opted for a TUI instead of a traditional graphical user interface (GUI) as the typical embedded development workflow is centered around the terminal. Switching to an external graphical program windows is likely to interrupt the common workflows.[1]

The TUI is tabulated and each tab provides a different context of work. It is beneficial to restrict the displaying of information to a specific context that the user can set. It eliminates the need for Jelly to predict ("guess") what information the user might be interested in and it also forfends overwhelming the user with information that is not relevant. The downside is that the user is required to learn how to set or change the context and it also assumes that the user inherently knows which context suits their task. Jellys default context tries to minimize the need for switching the context by providing information from different contexts together in a single view but with reduced details. Contexts are switched using the function keys.

[1] While not reaching academic standards, we did informal questionnaires with embedded developers for common behaviors and expectations around workflows and tooling at the RIOT Summit 2024 Vienna, Austria.

Fn Key	Name	Context
F1	Overview	The default. With reduced details, shows diagnostic messages, Jelly-based commands, CoAP requests & responses and some status information
F2	Text	Diagnostic messages log with timestamps
F3	CoAP	CoAP requests & responses log with timestamps
F4	Commands	Jelly-based command log with timestamps
F5	Net	IP packet log with timestamps
F12	Help	Help about using Jelly

Table 1: Jellys TUI is tabulated and each tab provides a different context. The tabs can be quickly accessed using their assigned function keys.

5.7 Pass-Through Modes

Sometimes a TUI is not desirable, as motivated in Section 4.2.3. For example, a TUI can hardly be passed around using pipes. Jelly provides multiple pass-through modes for such use cases. It calls this mode of operation "headless" to match the common naming scheme from other software ecosystems. In headless mode, Jelly does not render a TUI but reads and writes directly standard input and output. This enables Jelly to be used inside shell scripts. Three headless modes are currently available:

1. A diagnostic mode, which transparently passes text messages.
2. A configuration mode, which reads in CoAP-based commands, executes them and awaits their completion before printing the collected output.
3. A network mode, which only transports in- and outgoing IP packets. It does not read in user input but prints a summary of the network traffic.

The modes are selected using Jellys command line interface, see Section 5.8.

When using a headless mode, the default event loop (see Figure 1) is replaced with a specialized version for the selected mode. For example in the headless diagnostic mode, shown in Figure 2, the rendering of the TUI is omitted, as well as the application logic. Since we are not providing auto completion (or any other custom input options) and are not interested in CoAP, we can pass input right through to the RIOT node. Any output is printed directly to standard output.

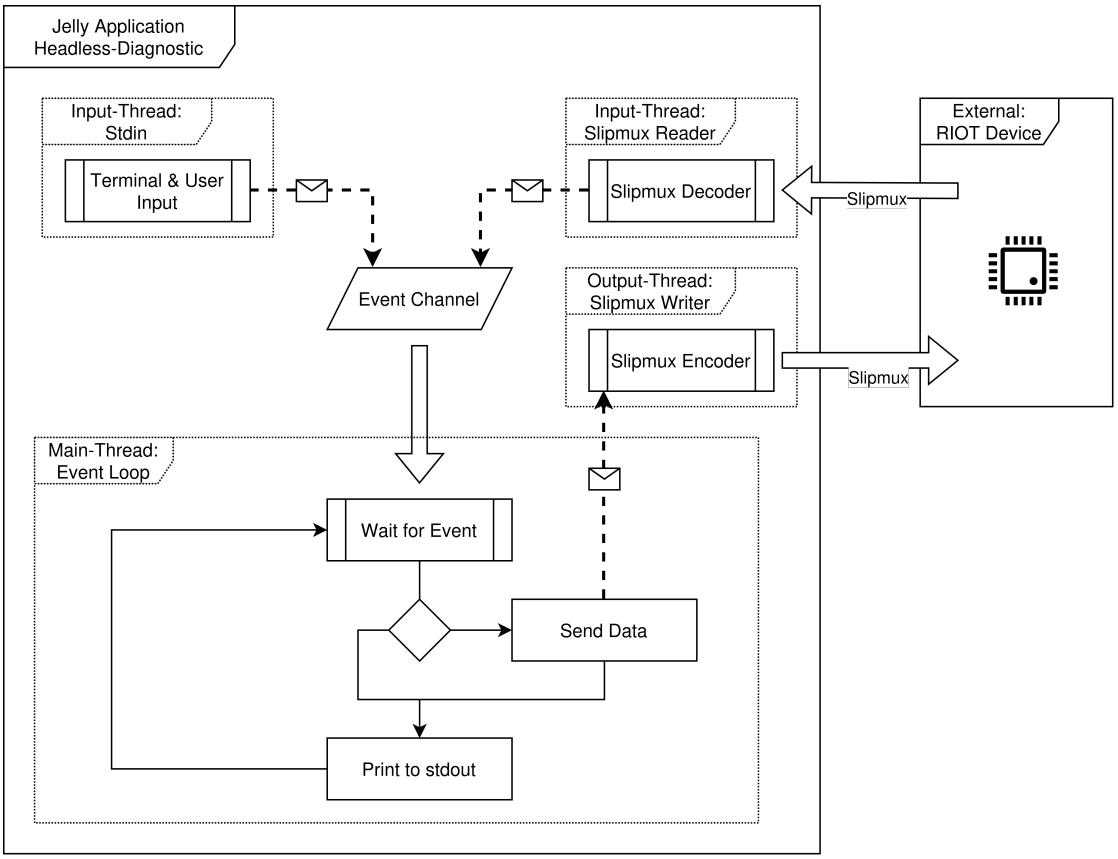


Figure 2: Abstract overview on the reactor setup when using the headless diagnostic mode. The mode specific event loop awaits events and acts on them by either printing them directly or by sending messages to the slipmux writer thread via a message queue first, depending on the type of the event.

5.8 Command Line Interface

Jelly makes use of a rich command line interface (CLI) for configuration during invocation. The actual parsing of the arguments is done by the third party crate CLAP (Command Line Argument Parser) [11]. The CLI of Jelly has one required parameter which is the location of the serial interface that should be used. Optionally the headless modes, networking and TUI color theme can be set, as shown in Listing 4.

```
Usage: Jelly [OPTIONS] <TTY_PATH>

Arguments:
  <TTY_PATH>
          The path to the UART TTY interface
Options:
  -t, --network [<NETWORK>]
          If enabled, attaches the SLIP network to the given host TUN interface
          Default interface name is slip0

  -d, --headless-diagnostic
          If true, disables the TUI and passes diagnostic messages via stdio
          This is intended for interactive, shell-like usage.
          Jelly will await input and output indefinitely.
          Configuration messages are ignored. This means that any pre-known or
          configuration-based commands are not available.
          May be used with `--network`.

  -c, --headless-configuration
          If true, disables the TUI and passes configuration messages via stdio
          Use this mode inside scripts and pipe commands into Jelly.
          This may be used interactively. Jelly will await input unitl EOF.
          Jelly will wait for output until all commands are finished or
          the time-out is reached. The output will only be displayed once EOF is
          reached. This is to preserve the order of input commands regardless of
          the commands run time. Diagnostic messages are ignored.
          Pre-known, configuration-based commands are available.
          May be used with `--network`.

  -n, --headless-network
          If true, disables the TUI and shows the in-/out-going packets.
          Diagnostic messages are ignored. Configuration messages are ignored.
          Must be used with `--network`.

      --color-theme <COLOR_THEME>
          Sets the color theme of the Jelly TUI
          This setting has no effect when not using the TUI.
          [default: auto]
          [possible values: auto, light, dark, riot, none]

  -h, --help
          Print help (see a summary with '-h')
```

Listing 4: The CLI of Jelly as obtained by running `Jelly --help`.

6 Evaluation

We performed a tentative evaluation of Jelly to inspect whether the requirements have been met.

6.1 Ease of Use

The requirement from Section 3.1 states that Jelly should be easy to use. A major aspect of this is, that it should automate repetitive tasks and communicate its operative state clearly to the user. During startup, Jelly automatically queries the constrained device for its RIOT version and the development board it is running on. This information is then displayed to the user, as shown in Listing 5.

```
Overview (F1) Text (F2) CoAP (F3) Commands (F4) Net (F5) Help (F12)
┌Text & Commands──────────────────────────────────────────────────┐┌CoAP Req & Resp──────────────────┐
│main(): This is RIOT! (Version: 2025.01-devel-3960-HEAD)        ││ ← Req(Get /.well-known/core)    │
│Welcome to RIOT!                                                ││ → Res(Content/ALF)              │
│> ToggleLed0                                                    ││</jelly/board>,                  │
│Toggled the LED0!                                               ││</jelly/ver>,                    │
│>                                                               ││</jelly/led0>,                   │
│                                                                ││</.well-known/core>              │
│                                                                ├┤──────────────────────────────── │
│                                                                ││ ← Req(Get /jelly/board)         │
│                                                                ││ → Res(Content)                  │
│                                                                ││nrf52840dk                       │
│                                                                ├┤──────────────────────────────── │
│                                                                ││ ← Req(Get /jelly/ver)           │
│                                                                ││ → Res(Content)                  │
│                                                                ││2025.01-devel-3960-HEAD          │
│                                                                ├┤──────────────────────────────── │
│                                                                ││ ← Req(Get /jelly/led0)          │
│                                                                ││ → Res(Changed)                  │
│                                                                ││(Empty payload)                  │
└┌User Input──────────────────────────────────────────────────────┘└─────────────────────────────────┘
 │Type a command, for example: help, Wkc, /.well-known/core       ┌Board Info──────────────────────┐
 └────────────────────────────────────────────────────────────────┘│Version:                        │
                                                                   │2025.01-devel-3960-HEAD         │
                                                                   │Board: nrf52840dk               │
                                                                   └────────────────────────────────┘
──────────────────────────────────────────────────────────────────────────────✅ connected via /dev/ttyACM0
```

Listing 5: Interacting with RIOT through Jelly using: `Jelly /dev/ttyACM0`. The `ToggleLed0` command (see Section 6.3) was executed. This command accesses the CoAP resource `/jelly/led0` and the successful request and response can be seen on the right.

6.2 Backwards Compatibility

To evaluate the compatibility requirement from Section 3.2, we compare RIOT's current behavior with `Pyterm` against its behavior with Jelly. Pyterm is the default software that RIOT uses to display and interact with serial input/output.

As a concrete example, we use the default shell application on a `nrf52840dk` development board and execute the `version` command in the RIOT shell.

Building and flashing the application without any modifications, then connecting to the board via Pyterm, yields the standard RIOT shell prompt and command output, as shown in Listing 6.

For use with Jelly, we build and flash the application once more. This time, including the module required for Jelly, `stdio_slipdev`, which enables transport of stdio via slipmux. Using Jelly in headless (non-TUI) mode transparently passes through to the user's terminal, just as Pyterm does. The result is shown in Listing 7.

The major difference in the user-perceived output is that Jelly omits the timestamps of received lines. This can be added in future versions of Jelly, since the timestamps are already recorded and displayed in the TUI. Besides the metadata, the shell output is functionally identical, demonstrating that Jelly is backwards-compatible with the default RIOT workflow. This functional parity is further evidenced by RIOT's shell test suite (`RIOT/tests/sys/shell`), which passes when run using Jelly (see footnote)[2].

```
2026-10-06 16:49:13,590 # Connect to serial port /dev/ttyACM0
Welcome to pyterm!
Type '/exit' to exit.
> version
2026-10-06 16:49:15,783 # version
2026-10-06 16:49:15,785 # 2025.01-devel-3956-gaa5493
>
```

Listing 6: Interacting with the RIOT shell through Pyterm. The `version` command was executed.

[2] `PORT="-d /dev/ttyACM0" BOARD=nrf52840dk USEMODULE=jelly make -C tests/sys/shell flash test`

```
Serial connect with /dev/ttyACM0
> version
version
2025.01-devel-3956-gaa5493
>
```

Listing 7: Interacting with the RIOT shell through Jelly running headless using: `Jelly -d /dev/ttyACM0`. The `version` command was executed.

6.3 Low Barrier for Implementing CoAP-based Commands

To demonstrate that the requirement from Section 3.3 is met, an exemplary implementation for a primitive led toggling command is provided. This command is called `ToggleLed0` and when successfully executed toggles the first LED on the attached RIOT device and reports the success to the user. Should the action fail, the user is informed of the failure.

First, we implement the RIOT side of this command. The necessary C code is shown in Listing 8. It consists of two parts: the definition of a CoAP resource and a handler function that processes incoming requests for this resource. The resource is called `riot_led0` and its definition covers the path (`/jelly/led0`) at which the resource is exposed, the methods that are allowed for it (only `GET` in this case), which handler to call (`_riot_led0`) and the transport protocols over which this resource is exposed. We restrict the transport to `slipmux` as used by Jelly to talk to the device. The handler function `_riot_led0` is invoked by the CoAP server whenever a request for our resource is received. Since our example is deliberately minimal, the handler ignores all passed arguments. It toggles the first LED using a RIOT-provided macro and returns the CoAP response code that indicates a successful state change

```
static int _riot_led0(
    unicoap_message_t *message, const unicoap_aux_t *aux,
    unicoap_request_context_t *ctx, void *arg)
{
    (void)message;
    (void)aux;
    (void)ctx;
    (void)arg;
    LED0_TOGGLE;
    return UNICOAP_STATUS_CHANGED;
}

UNICOAP_RESOURCE(riot_led0)
{
    .path = UNICOAP_PATH("jelly", "led0"),
    .methods = UNICOAP_METHODS(UNICOAP_METHOD_GET),
    .handler = _riot_led0,
    .protocols = UNICOAP_PROTOCOLS(UNICOAP_PROTO_SLIPMUX),
};
```

Listing 8: CoAP-based command endpoint declaration in RIOT. The declared CoAP resource toggles LED 0 on GET requests.

Next, on the Jelly side, we create a file for the ToggleLed0 command that uses the `riot_led0` CoAP resource. The file is split into three parts, Listing 9 a-c, the definitions, the setup and the behavior.

The definitions in Listing 9 (a) consist of the needed imports and the structure that holds the state of our command.

Setup is performed by providing a public function (`cmd()`) that tells Jelly about our new command as shown in Listing 9 (b). The returned `Command` structure tells Jelly the command name, a user-facing description, the CoAP resources it requires and the function Jelly should call when instantiating it. The parse function takes the typed-in command line and returns either an instance of the previously defined command-specific structure or an error message with the reason why parsing the CLI failed. For this deliberately minimal example, we have nothing to parse and always return a success with an instance of the `ToggleLed0` structure.

```
use coap_lite::RequestType as Method;
use coap_lite::{CoapRequest, Packet};

use super::{Command, CommandHandler, CommandType};

struct ToggleLed0 {
    location: String,
    buffer: String,
    finished: bool,
}
```

Listing 9 (a): Imports required for a CoAP-based command: the CoAP-related parts of the `coap_lite` library, Jelly's command-related interfaces and types, and a `ToggleLed0` structure that holds the command's execution state.

```
pub fn cmd() -> Command {
    Command {
        cmd: "ToggleLed0".to_owned(),
        description: "Toggles LED0".to_owned(),
        parse: parse_cli,
        required_endpoints: vec!["/jelly/led0".to_owned()],
    }
}

fn parse_cli(cmd: &Command, _args: &str) -> Result<CommandType, String> {
    Ok(CommandType::CoAP(Box::new(ToggleLed0 {
        location: cmd.required_endpoints[0].clone(),
        buffer: String::new(),
        finished: false,
    })))
}
```

Listing 9 (b): A public `cmd()` function exposes the command to Jelly: its name, a user-facing description, and the CoAP resources that must be available for the command to be offered. A `parse_cli()` function is invoked when the user requests the command; in this minimal example it performs no parsing and simply returns a ToggleLed0 instance.

Our command is executed through the `CommandHandler` interface, which is implemented for `ToggleLed0` in Listing 9 (c). Jelly first calls the `init()` function and expects a CoAP request in return, which is then sent to the connected RIOT device. Since our example is minimal, we only return a direct GET request to the `/jelly/led0` endpoint without further payloads.

Once the response arrives, Jelly executes the `handle()` callback. Our command checks if the response indicates an error and stores the corresponding message to the user in a buffer. Since no further CoAP request is returned, Jelly then checks if the command wants to display information to the user and if it is finished. Lastly, Jelly uses the `display()` callback to collect the information that our command wants to show to the user and displays it to the user.

```
impl CommandHandler for ToggleLed0 {
    fn init(&mut self) -> CoapRequest<String> {
        let mut request: CoapRequest<String> = CoapRequest::new();
        request.set_method(Method::Get);
        request.set_path(&self.location);
        request
    }
    fn handle(&mut self, response: &Packet) -> Option<CoapRequest<String>> {
        let resp_code = match response.header.code {
            coap_lite::MessageClass::Response(ref code) => code,
            _ => &coap_lite::ResponseType::UnKnown,
        };
        if resp_code.is_error() {
            self.buffer.push_str("An error occurred!");
        } else {
            self.buffer.push_str("Toggled the LED0!");
        }
        self.finished = true;
        None
    }
    fn want_display(&self) -> bool {
        true
    }
    fn is_finished(&self) -> bool {
        self.finished
    }
    fn display(&self, buffer: &mut String) {
        buffer.push_str(&self.buffer);
    }
}
```

Listing 9 (c): Implementation of the `CommandHandler` trait for ToggleLed0: init builds the CoAP GET request, handle processes the response and records a user-facing message, and display reports the result back to Jelly.

7 Conclusion and Outlook

This first iteration of Jelly shows working externalization of computation via a shell interface. It provides the necessary infrastructure for evaluating the operational parameter such as resource usage, performance or flexibility.

One key challenge yet to solve is the mixing of responsibility and dependability of the CoAP-based commands and the endpoints they rely on. These commands are implemented in Jelly but require specific endpoints in the constrained device. This creates a dependency between the Jelly version and the version of the firmware of the device. Related to that, while the code of the endpoints for a custom RIOT command are located within the RIOT repository, the Jelly side is located within the repository of Jelly. To ease both issues there is the idea to not implement those commands in Rust within Jelly but in a script language in outside files. Jelly would load these commands at runtime from their script files. This allows to store not only the endpoints but also the Jelly side within the RIOT repository. The matching of versions would be ensured by the version control system used by RIOT. Further a setup like this removes the need to recompile Jelly to change commands and also enables editing or adding of commands during runtime. An alternative approach is to add Jelly to the RIOT project as an external package. Packages are downloaded and compiled on the fly when needed. Crucially, RIOT supports to patch packages prior compilation. The CoAP-based commands would be implemented in patches for Jelly, stored inside the RIOT project. This would enable the version control system to ensure that Jelly always get compiled with commands that match the endpoints and their versions that are available in RIOT.

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, Internet-Draft draft-bormann-t2trg-slipmux-03, Nov. 2019. [Online]. Available: https://datatracker.ietf.org/doc/draft-bormann-t2trg-slipmux/03/

[3] "Nonstandard for transmission of IP datagrams over serial lines: SLIP," RFC Editor, technical report 1055, June 1988. doi: 10.17487/RFC1055.

[4] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)," RFC Editor, technical report 7252, June 2014. doi: 10.17487/RFC7252.

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

[6] B. Hattesen, "Slipmux for RIOT." [Online]. Available: https://www.inet.haw-hamburg.de/teaching/ss-2025/project-class/prj1_bennet_hattesen.pdf

[7] L. Pouzin, "The SHELL: A Global Tool for Calling and Chaining Procedures in the System," MASSACHUSETTS INSTITUTE OF TECHNOLOGY, technical report, 1965. [Online]. Available: https://people.csail.mit.edu/saltzer/Multics/Multics-Documents/MDN/MDN-4.pdf

[8] C. Ramey and B. Fox, "The GNU Readline Library, Edition 8.3." [Online]. Available: https://tiswww.case.edu/php/chet/readline/rltop.html

[9] D. C. Schmidt, *Pattern Languages of Program Design*, 1st ed., vol. 1. Addison-Wesley, 1995. [Online]. Available: https://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf

[10] "ratatui." [Online]. Available: https://ratatui.rs/

[11] "clap." [Online]. Available: https://github.com/clap-rs/clap