Bennet Hattesen

Jelly, a Modern Shell for Constrained Devices

Faculty of Engineering and Computer Science
Department of Computer Science

Submitted: 27. August 2024

Supervising examiner: PROF. DR. THOMAS SCHMIDT

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Contents

1 Introduction

Software is written by, and used by, humans. As it is in the nature of humans, this generates experiences which may or may not be desired, beneficial, pleasant or even repulsive. This aspect of software is part of the User eXperience (UX). The ISO defines UX in 9241-210 as “user’s perceptions and responses that result from the use and/ or anticipated use of a system, product or service” [1]. Part of UX are user interfaces (UI), which are the points of contact between systems and their users [2, p. 5]. A good UI only is not enough for good UX as UX is created and felt internally by the user upon interacting with the system. As such, UX depends on the context (different usage, different user, ..) between the system and the user. It is beneficial to guide design and development of software towards pleasant user experiences as those effect the performance related usability aspects (time per task, productivity) [3, p.19-20].

RIOT OS is an embedded operating system and as it is open source, many programmers contribute code to the project [4]. Programmers are also users of the OS and as such, they are subject to user experience as well. Developer of software systems, like RIOT OS, often develop special software and programming tools, to aid the development of the main project. This particular case of UX, in which programmer use and write programming tools, is named programmer experience (PX) [5]. This differentiation is beneficial as it predefines some of the context parameters in the user interaction which creates the UX. For example, the wording in the UI may be more technical without offsetting the user.

1.1 Preliminary Research Objectives

In this assignment, we explore possible PX improvements for RIOT OS with a focus on the shell interface. Additional motivation is drawn from recent technological developments, such as CoAP, which open the possibility to build new tooling uppon those. These two aspects split the research direction into two categories, a technical aspect with proof of concept (PoC) and a PX aspect:

1. PoC: The new shell:
   1. Can the shell be re-implemented using CoAP over a serial interface?
   2. How does the memory overhead compare to the existing shell?
2. The PX of RIOT when interacting with the shell:

1. Assessing the status quo, what is the current situation, what are benefits and which shortcomings are present?
2. How can a new shell improve the PX?
3. Tentative inspection of the PoC UI in the light of PX

1.2 Outline

First, we give a short review of common UX guidelines and introduce usability rules and design principles in Section 2. Then, in Section 3, we explain the current RIOT shell and assess it in consideration of the previously introduced rules and principles. We provide an assessment of the deficits and technical shortcomings of the RIOT shell. With that, the overview of the current deficits and our reasoning why they are problematic completed. Next, we theorize an improved shell in Section 4 and argue how this idea resolves the current shortcomings and enhances the usability. Finally, we test out our theories by building a proof of concept in Section 5. With it, we show that the technical and usability related deficits can be solved. Lastly, we give an outlook on the remaining work, including further chances and technical challenges, the on going research on the usage of the shell within in the RIOT community and future capabilities that our approach might offer. This outlook completes this review in Section 6.

2 UX Guidelines

2.1 Usability Rules

Usability is a key component for user experiences, as it lifts the software from a necessity to tool that is an extensions of their capabilities to accomplish their work [3, p. xi]. In the book "Software for Use" by L. Constantine and L. Lockwood [2], usability is further characterized by providing five guiding rules:

> *The system should be usable, without help or instruction, by a user who has knowledge and experience in the application domain but no prior experience with the system.*
>
> — *First Rule:* ***Access***

The system should not interfere with or impede efficient use by a skilled user who has substantial experience with the system.

— *Second Rule:* ***Efficacy***

The system should facilitate continuous advancement in knowledge, skill, and facility and accommodate progressive change in usage as the user gains experience with the system.

— *Third Rule:* ***Progression***

The system should support the real work that users are trying to accomplish by making it easier, simpler, faster, or more fun or by making new things possible.

— *Fourth Rule:* ***Support***

The system should be suited to the real conditions and actual environment of the operational context within which it will be deployed and used.

— *Fifth Rule:* ***Context***

(*Software for Use, 1999, p. 47-51*)

2.2 Design Principles

In addition, "Software for Use" provides these six principles, to help evaluation of designs and guide decision making, when creating user centered designs and interfaces:

1. ***Structure Principle*** — *Organize the user interface purposefully, in meaningful and useful ways that put related things together and separate unrelated things based on clear, consistent models that are apparent and recognizable to users.*
2. ***Simplicity Principle*** — *Make simple, common tasks simple to do, communicating clearly and simply in the user's own language and providing good shortcuts that are meaningfully related to longer procedures.*
3. ***Visibility Principle*** — *Keep all needed tools and materials for a given task visible without distracting the user with extraneous or redundant information: What You See Is What You Need (WYSIWYN).*

4. ***Feedback Principle*** *— Through clear, concise, and unambiguous communication, keep the user informed of actions or interpretations, changes of state or condition, and errors or exceptions as these are relevant and of interest to the user in performing tasks.*
5. ***Tolerance Principle*** *— Be flexible and tolerant, reducing the cost of mistakes and misuse by allowing undoing and redoing while also preventing errors wherever possible by tolerating varied inputs and sequences and by interpreting all reasonable actions reasonably.*
6. ***Reuse Principle*** *— Reduce the need for users to rethink, remember, and rediscover by reusing internal and external components and behaviors, maintaining consistency with purpose rather than merely arbitrary consistency.*

(Software for Use, 1999, p. 536)

3 RIOT Shell

The shell offered by RIOT OS consists of plain ASCII input and output, exchanged via a serial/UART interface. The shell never terminates and therefore runs in a loop, which starts with reading (or awaiting) input from the user, as seen in Figure 1. The input is then separated into the command name plus the arguments, if any. Next, the shell checks for byte-by-byte match of the command name with one the available commands. If a match is found, the command function is executed and the arguments get passed along. Once the command completes and returns the flow of execution back to the shell, the shell loops and reads for new user input.

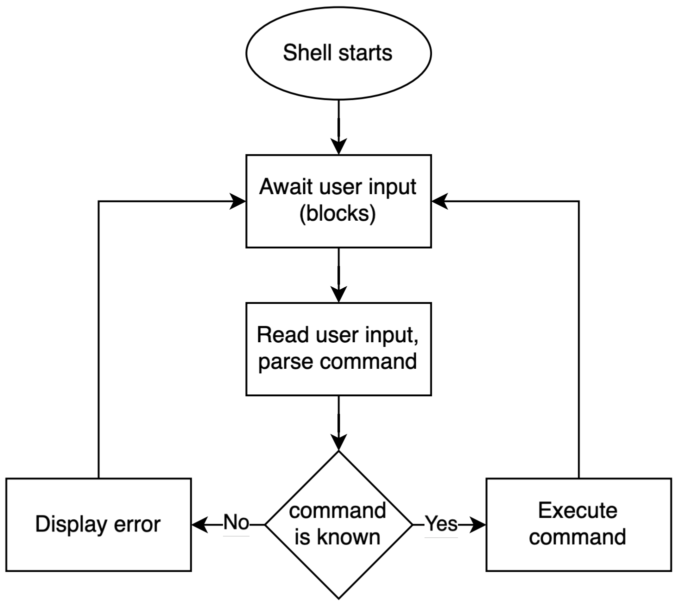


Figure 1: The flow of execution for the shell in RIOT, simplified.

3.1 The Significance of the Shell for the RIOT Community

The shell is one the earliest features of RIOT that is introduced to newcomers. The RIOT default example already showcases the shell and is also the example used in the Getting-Started documentation [6]. Additionally, most developers are already comfortable using a shell due to exposure to the concept via Linux / MacOS terminals, which often run shells such as ZSH or BASH.

The shell also provides an easy way to provide run-time configuration. That is the ability for the user to change settings while the embedded system is already running. Naturally, this can be used to query the current configuration as well.

We believe the shell is one of the most frequently used RIOT modules.

3.2 Assessing the Usability

To assess the usability of the RIOT shell, the five usability rules (Section 2.1) are contrasted:

Access

The shell is mostly easy to use without help or instructions, as its usage parallels well known (Unix) shells, which are assumed to be familiar to the user. As the shell commands do not have a uniform argument or help interface, some might be more difficult to use.

Efficacy

The shell is lacking basic ease-of-use functionality, such as auto-completion, command history or command search. It therefor might impede efficient use by skilled users.

Progression

This rule does not apply as the complexity of the shell is very limited.

Support

The shell provides minimal support only. It does not provide convenience function nor assist the user with frequent tasks, such as converting number formats.

Context

The RIOT shell is suited for its usage context. It is lightweight and does not interrupt the development workflow.

We assess the overall usability of the RIOT shell to be sufficient.

3.3 Assessing the User Interface

The terminal UI of the RIOT shell is not without shortcomings in various user interface aspects and is in some conflict with guiding principles (Section 2.2).

3.3.1 Understanding the Information Output

Providing clear, easy to understand messages is important for any program in order to reach high developer satisfaction (4th principle: “Feedback”). There are two problems with the RIOT shell, that hinder this aspect.

- Logging information gets mixed with the shell. For example, if the shell is currently used for reading sensor values, the flow of text might get interrupted by a debug message from the networking thread. This can lead to confusion. This is also in conflict with the first principle “Structure”.
- Adjacent to that, this makes it hard to evaluate a shell log in hindsight, as the logged output is not directly matched to the input commands.
- Certain information is more often needed than other, e.g., the current IPv6 address is frequently queried via the shell. Yet, it is a manual and repetitive process, done by the user and is then displayed within the complete interface description. This might be in conflict with the third principle “Visibility”.

3.3.2 Operating the Shell

Ease of use is a crucial property of any programming tooling. Developers are more productive, if they can focus on the task at hand rather than working around shortcomings within their tooling (2nd principle: “Simplicity”). The RIOT shell is sparse and minimal, as such, modern and ubiquitously available control interfaces are missing:

- It is not possible to search for previous executed commands.
- There is no auto completion for half typed commands available.
- The help system is very limited, especially compared to the mature manpage system found on many Linux systems.
- The iterative and looping nature of the shell prevents asynchronous workloads. For example, a shell command that runs for a long period of time blocks the shell and no other commands can be run in the meantime. This is an other instance of the afore mentioned mixing of information in Section 3.3.1.
- As commands do not have a uniform way of passing arguments, the user has to remember or rediscover the exact way each command is operated. Instead, the usage of each command should feel consistent as described by the sixth principle “Reuse”.

3.4 Limitations from a Technical Perspective

The technical implementation of the shell restricts advanced use cases.

1. The text input and text output is meant for human consumption.

This makes it difficult to write automated test suits that check the correct operation of specific shell commands. The current workaround by using regular expression is brittle.

In addition, this blocks the shell from being used in an machine-to-machine (M2M) scenario.

2. No matching between input and output.

   Currently, there is no way to tell which command or component of RIOT produced a given output seen in the shell. This, again, makes automated testing and M2M scenarios difficult.

   Additionally, this prevents the shell from being used in asynchronous settings, like timer, event notifications and other long running cyber physical tasks.

3. Text encoding is mandatory.

   Since the shell or shell commands must be executed in a text-based terminal, they cannot send binary data as such. Instead, if it is necessary to transmit binary data, the data needs to be re-encoded in a textual form, e.g. base64.

4. String operations are done on the constrained device

   Because input and output of the shell is in text form ("strings"), the RIOT device must not only decode and translate strings into computer understandable binary values, but also translate the binary output into strings before sending the information to the shell. This creates a considerable overhead.

4 How to Improve

The easiest way to plan a new shell that improves the programmer experience is by eliminating the technical shortcomings first and later building upon that. This approach is, by design, not user centric and is not recommended [3], [2]. It is therefore strictly necessary that this deviation is later caught up on by future research, if the proof of concept is viable.

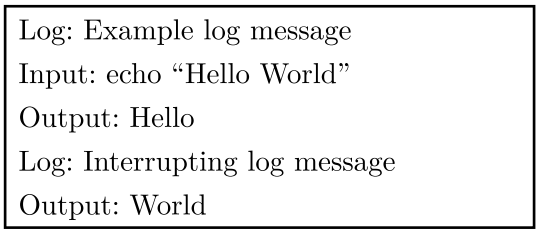
4.1 Elimination of the Technical Shortcomings

The Constrained Application Protocol (CoAP) offers lightweight communication, suitable for M2M scenarios [7]. A variety of content formats are supported, including plain text. It provides a reliable way to match requests to responses, in an asynchronous manner. Further, it enables resource discovery and provides a feature called “observation” in which event generation can be requested for a given resource.

With these properties, CoAP presents a solid option to cover the technical shortcomings. The content format of the payload can be set per message / resource, enabling the shell to switch between text and binary messages, fixing the technical shortcoming 3 (Text encoding is mandatory). This also enables the possibility to address shortcoming 1 (The text input and text output is meant for human consumption), by providing machine readable resources, instead of text based commands. If user input is modeled as a CoAP request, then the output is modeled as a CoAP response. The request and response matching of CoAP can hence be used to match the user command to the output of the command, thereby resolving the issue of interlaced information output (shortcoming 2, No matching between input and output). Lastly, because CoAP supports binary exchanges, there is no longer a requirement to do unnecessary string operations on the constrained device (shortcoming 4, String operations are executed on the constrained device).

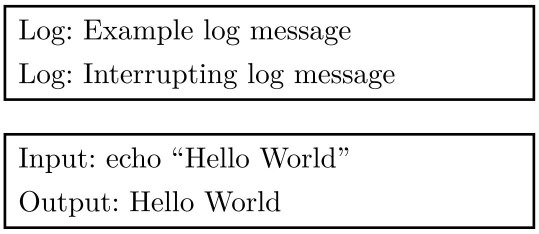
4.2 Creating a New User Interface with CoAP in Mind

With CoAP providing asynchronous interactions as well as input and output matching, the formerly list of all RIOT output, as shown in Figure 2, can be replaced. First, regular logging should get separated from the user based interactions, shown in Figure 3. Next, the list of output gets replaced by a list of request & response pairs. This results in the interface approach shown in Figure 4.



```
Log: Example log message
Input: echo “Hello World”
Output: Hello
Log: Interrupting log message
Output: World
```

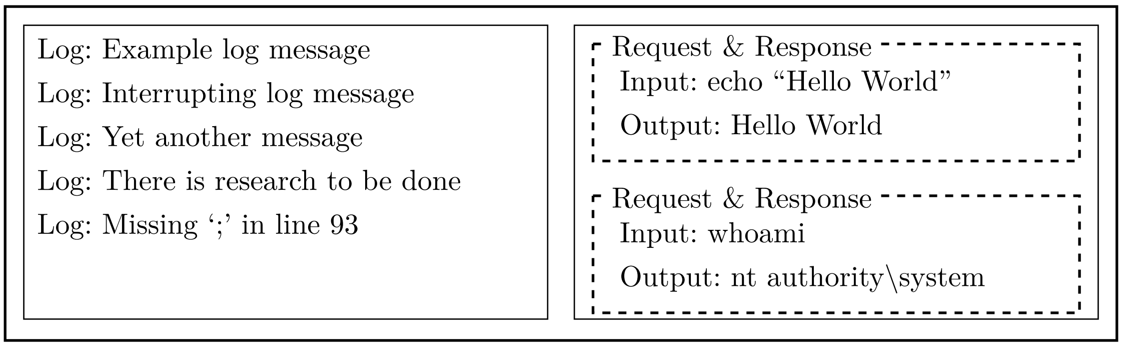
Figure 2: The current output format of the shell. The input command 'echo "Hello World"' is executed, but its output is sliced in half by a logging message.



```
Log: Example log message
Log: Interrupting log message
```

```
Input: echo “Hello World”
Output: Hello World
```

Figure 3: A shell, which separates the logging messages from the command input and output.



```
Log: Example log message
Log: Interrupting log message
Log: Yet another message
Log: There is research to be done
Log: Missing ‘;’ in line 93
```

```
Request & Response
 Input: echo “Hello World”
 Output: Hello World

Request & Response
 Input: whoami
 Output: nt authority\system
```

Figure 4: A combined interface for a shell, where logging is separated from the users activities. The command inputs and outputs, modeled on to the requests and responses from CoAP, are clearly matched together.

4.3 Applying the Design Principles

The “Structured” principle guides us towards separation of concern and consolidation of related things. With the base design introduced in Section 4.2 we already achieved this partially. As by the “Visibility” principle, information that is needed for the task at hand should be visible. Unnecessary or redundant information should be hidden to avoid confusing the user. Together with the “Simplicity” principle, which encourages to make common task simple, we can attempt to collect a list of information that should be shown to the user all the time.

4.3.1 Frequently needed information

- Status of the connection: Am I connected to the RIOT node?
- Which node am I connected with?

- Which development board is it?
- Which firmware version is running on the node?
- What interfaces does the node have?
- What is the IPv6 address(es) of the RIOT node?
- What is the hardware address?

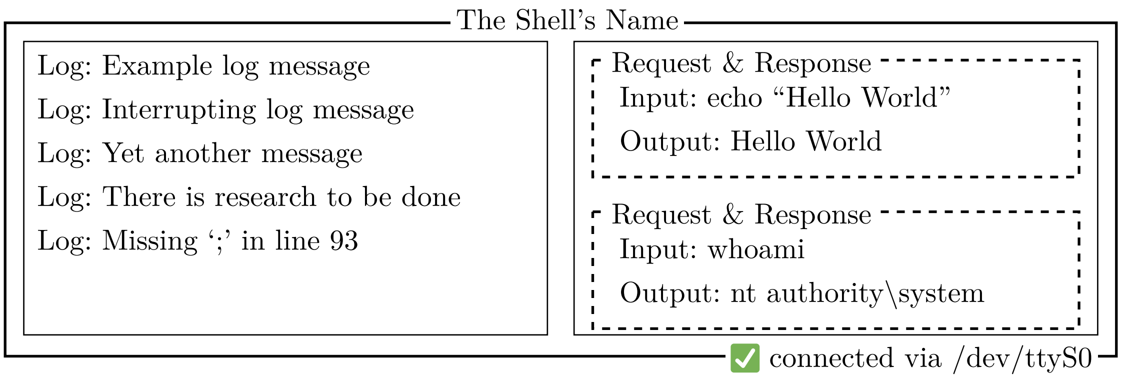
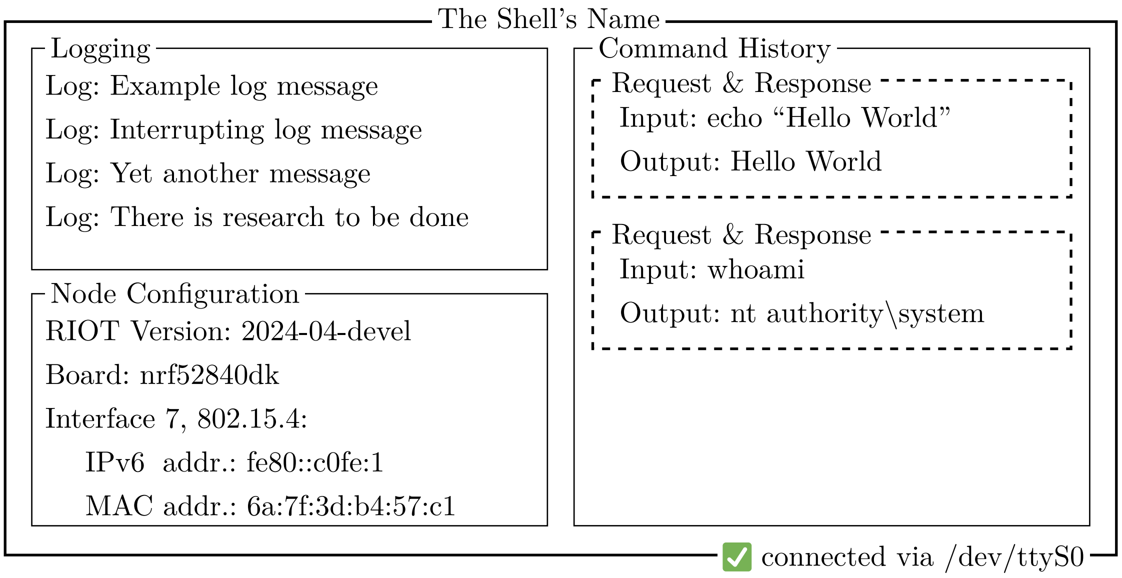

The Shell's Name
Log: Example log message
Log: Interrupting log message
Log: Yet another message
Log: There is research to be done
Log: Missing ';' in line 93
Request & Response
Input: echo "Hello World"
Output: Hello World
Request & Response
Input: whoami
Output: nt authority\system
✅ connected via /dev/ttyS0

Figure 5: Connection information is added via a status line at the bottom, build on top of the interface prototype shown in Figure 4

The frequently needed information, listed in Section 4.3.1, can be divided into two categories: General information on the connection and information related to the particular RIOT node. As by the design principles, those categories should be displayed separately. The general information is not as important for the direct task, but mainly to reduce the cognitive load on the users memory. As such, it is suitable to display this information non-prominently at the top or bottom of the screen in a status line. Care must be taken, as status lines are known to be insufficient for information that changes frequently, which would be a conflict with the "Feedback" principle [2, p.57]. This issue is eased for us, as the connection information changes rarely during the shell usage. Additionally, we can utilize bright colors to indicate a healthy connection or a disconnect, increasing the chance the user will notice a change of state. This idea is shown with a healthy connection in Figure 5, should a disconnect occur, the green checkmark emoji ('✅') can be replaced with a red cross ('❌'), providing a strong contrast. Switching not only the color but also the shape ensures readability and usability for colorblind users.

The second category, information that is related to the particular RIOT node, needs a separate display area. As user interaction is focused on the (growing) list of command requests & responses and the logging list being only passively read, we can re-purpose some of the logging area. As the user interface has already grown further away from its origin, the users will feel less familiar with it - now is a good time to add further descriptions and labels to the interface, in order to keep the software simple to understand. The new design is shown in Figure 6.

The last step for completing the new shell user interface is to add a way of user input. This is a simple, labeled box as presented in the final UI of Figure 7. Its clear separation from the displaying output field complies with the “Structured” principle. In fact, the repeated separation using boxes, with desciptive labels on top, perfectly matches the “Reuse” principle, by maintaining consistency.


The Shell’s Name

Logging
Log: Example log message
Log: Interrupting log message
Log: Yet another message
Log: There is research to be done

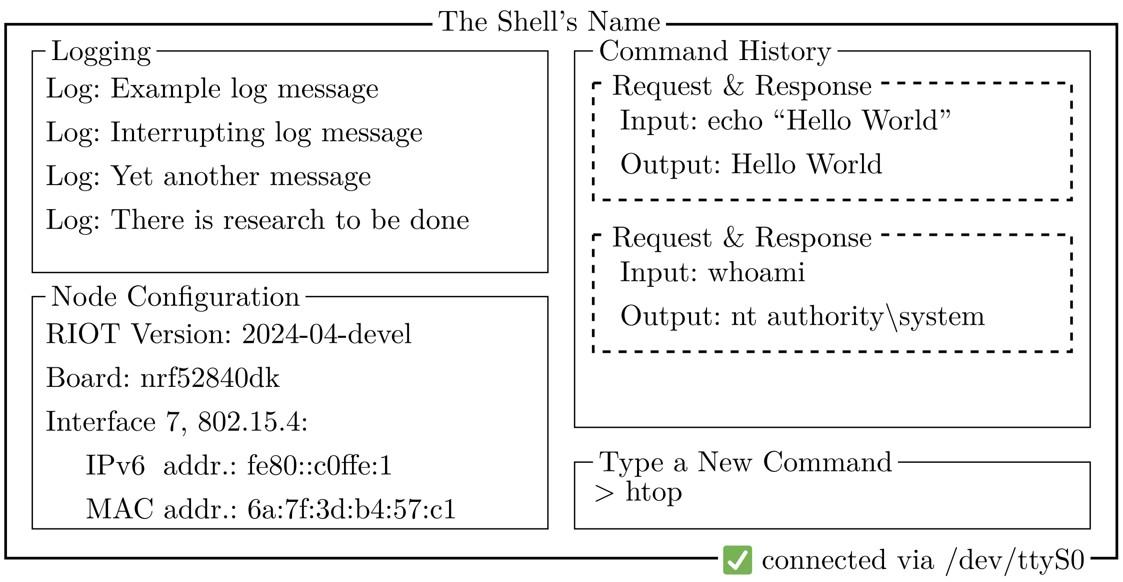
Node Configuration
RIOT Version: 2024-04-devel
Board: nrf52840dk
Interface 7, 802.15.4:
IPv6 addr.: fe80::c0fe:1
MAC addr.: 6a:7f:3d:b4:57:c1

Command History
Request & Response
Input: echo “Hello World”
Output: Hello World

Request & Response
Input: whoami
Output: nt authority\system

✅ connected via /dev/ttyS0

Figure 6: On the lower left side, a configuration frame shows information on the connected node. Iteration of the interface prototype previously shown in Figure 5



The Shell's Name

Logging

Log: Example log message

Log: Interrupting log message

Log: Yet another message

Log: There is research to be done

Node Configuration

RIOT Version: 2024-04-devel

Board: nrf52840dk

Interface 7, 802.15.4:

IPv6 addr.: fe80::c0ffe:1

MAC addr.: 6a:7f:3d:b4:57:c1

Command History

Request & Response

Input: echo "Hello World"

Output: Hello World

Request & Response

Input: whoami

Output: nt authority\system

Type a New Command

> htop

✅ connected via /dev/ttyS0

Figure 7: An input field for user commands is added to the lower right. The command "htop" is typed in but not send yet. Final version of the interface prototype, based on Figure 6

5 The Proof of Concept: Jelly

In order to build a proof of concept for the shell application, we must address the problem of transporting CoAP messages via a serial interface must be solved. We named our proof of concept Jelly and addressed this problem using SLIPMUX [8]. Jelly is available online [9].

SLIPMUX, presumably an abbreviation for Serial Line IP Multiplexing, is an Internet-Draft which proposes a protocol that enables using a single UART/Serial interface for diagnostics, configuration, and IP packet transfer. The packet transfer is kept in line with the original SLIP[10] standard and allows transferring IP packets. This can already be used to transport CoAP messages on top of IP over a UART. That, however, comes with the downside of overhead of the IP layer and the problem of configuration of the IP addresses for both endpoints. Instead, SLIPMUX defines how CoAP messages can directly be transferred over a UART. The draft calls this mode 'configuration'. This mode is desirable for the proof of concept as it has minimal overhead and does not need additional setup between the communicating parties. The third messaging offered by SLIPMUX are 'diagnostic' messages. Those are plain text, UTF-8 strings.

The ability to not only transfer CoAP messages, but also plain text makes SLIPMUX an ideal tool for Jelly. We can leverage CoAP to offer the new capabilities in our shell while being fully backwards compatible, when using plain text instead. As this works over the established UART interfaces, no new hardware or setups are required to use Jelly, which is a breaking requirement otherwise.

5.1 Implementing SLIPMUX on RIOT

SLIPMUX was already partly implemented in RIOT. For pure IP transfer, SLIP is available, which got extended to multiplex diagnostic messages. For Jelly, we extended the module further by adding CoAP to the multiplexing.

For storing the received frames before processing them by a CoAP handler, a chunked ring buffer is used. A buffer has to be used because processing might take longer periods of time and also might get suspended by a hardware interrupt for receiving the next data on UART. Finally, a ring buffer makes it easy for the processing thread to interoperate with the interrupt context, in which the data is read from the UART and stored within the buffer. The buffer is chunked, as data is received byte-wise, but processed message-wise (representing a CoAP message).

Oon RIOT startup a thread is created and run, which waits for a chunk of the buffer to become available. If that happens, the chunk is consumed and the contained CoAP message is processed. The handling of the CoAP message itself does not differ from the regular CoAP processing on RIOT. The only addition is that the responses are not send via the network stack but via SLIPMUX because the messages arrive without a source IP. There is no dedicated sender and receiver matching, so RIOT assumes that all CoAP messages coming via SLIPMUX also receive response via SLIPMUX.

5.2 Building a Terminal User Interface

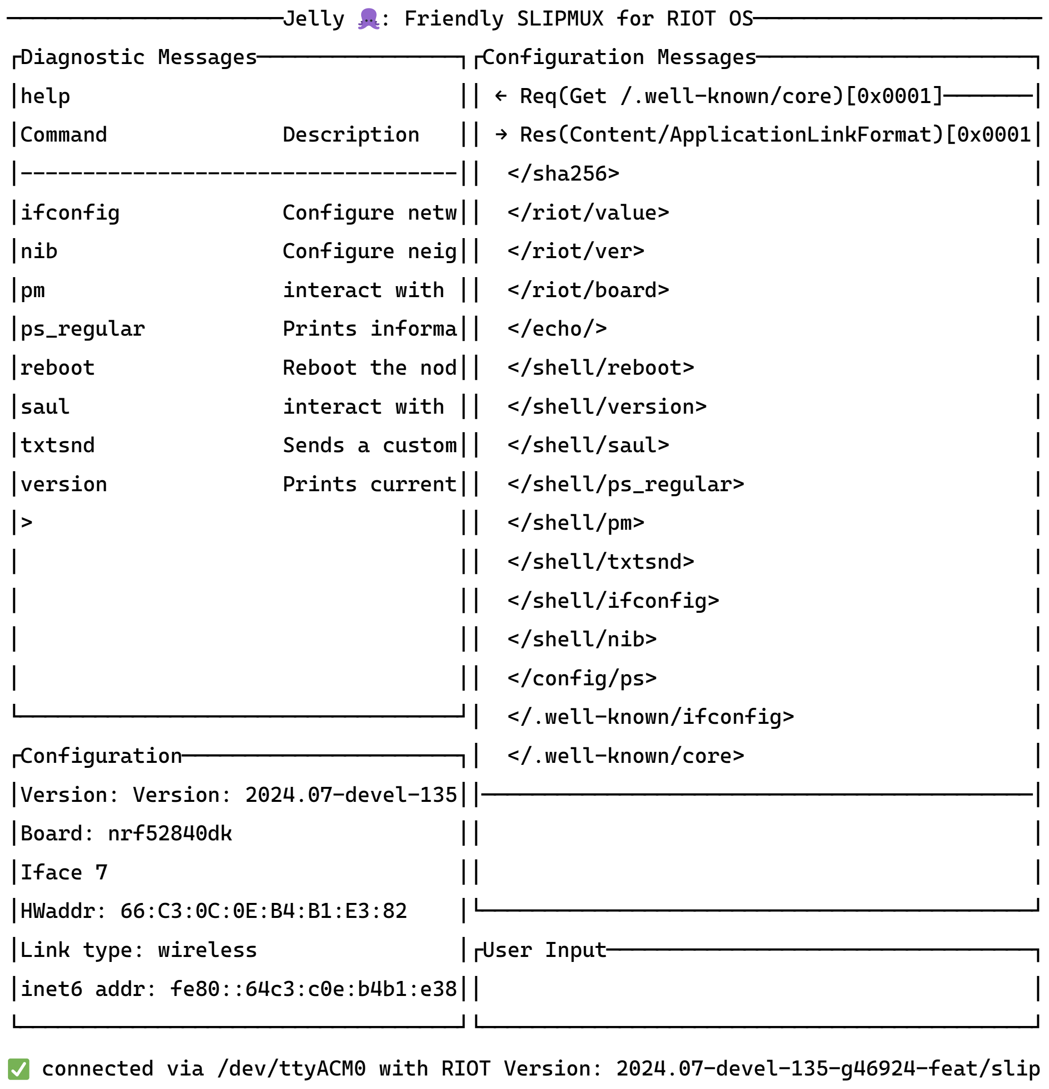
A terminal user interface (TUI) was chosen over a graphical one, as launching a graphical user interface would disrupt the regular developer workflow, which focuses between text editors and the terminal.

Since the development of the shell application does not interact with the RIOT code base, which is mostly written in C, there is no preset choice for the programming language. We opted for the Rust programming language, not for one specific reason, but a multitude of benefits. Rust comes with Cargo, the default build system and package manager, making it easy to include and manage dependencies such as CoAP libraries or TUI frameworks [11]. Rust is well-known for its friendly compiler, preventing many classical programming errors at compilation time while providing helpful error messages. This is further improved upon by Rustfmt and rust-analyzer, two standard toolings helping to format the code in a reasonable manner. In addition, it provides tips for best practices and suggests alternative notations for common code readability issues [12].

With Rust set as the starting point, we can utilize the Rust ecosystem of libraries and frameworks (called crates in the Rust jargon) to get head start by not having to implement all aspect from scratch. First, we need to interact with a serial port. For this, the serialport crate provides an easy way to open, read and write a given tty [13]. The serial_line_ip crate provides the framing and escaping of SLIP [14]. With only a few lines of additional code, we implemented SLIPMUX using this crate. The creation and parsing of CoAP messages is done using the coap-lite crate [15]. For the TUI the Ratatui project fits our usecase [16]. Ratatui makes the creation of TUIs easy, providing widgets, styling and layout to render the interfaces.

5.3 Using Jelly

Jelly provides an effective minimal proof of concept. It shows that it is feasible to build a shell-like experience on top of CoAP, successfully leveraging its benefits. The backwards compatibility via SLIPMUX's diagnostic messages is excellent. Jelly also offers auto-completion, an important quality of use improvement for programmers. We demonstrate the usage of the M2M capabilities of CoAP for automatic fetching and displaying frequently needed information, without any user intervention. The terminal user interface, as shown in Figure 8, fulfills our claim on the usability in regards to the principles set in Section 2.2.



```
──────────────────────Jelly 🐙: Friendly SLIPMUX for RIOT OS──────────────────────
┌Diagnostic Messages─────────────────┐┌Configuration Messages──────────────────────┐
│help                                ││ ← Req(Get /.well-known/core)[0x0001]───────│
│Command               Description   ││ → Res(Content/ApplicationLinkFormat)[0x0001│
│------------------------------------││  </sha256>                                 │
│ifconfig              Configure netw││  </riot/value>                             │
│nib                   Configure neig││  </riot/ver>                               │
│pm                    interact with ││  </riot/board>                             │
│ps_regular            Prints informa││  </echo/>                                  │
│reboot                Reboot the nod││  </shell/reboot>                           │
│saul                  interact with ││  </shell/version>                          │
│txtsnd                Sends a custom││  </shell/saul>                             │
│version               Prints current││  </shell/ps_regular>                       │
│>                                   ││  </shell/pm>                               │
│                                    ││  </shell/txtsnd>                           │
│                                    ││  </shell/ifconfig>                         │
│                                    ││  </shell/nib>                              │
│                                    ││  </config/ps>                              │
└────────────────────────────────────┘│  </.well-known/ifconfig>                   │
┌Configuration───────────────────────┐│  </.well-known/core>                       │
│Version: Version: 2024.07-devel-135││├────────────────────────────────────────────│
│Board: nrf52840dk                   ││                                            │
│Iface 7                             ││                                            │
│HWaddr: 66:C3:0C:0E:B4:B1:E3:82     │└────────────────────────────────────────────┘
│Link type: wireless                 │┌User Input──────────────────────────────────┐
│inet6 addr: fe80::64c3:c0e:b4b1:e38││                                            │
└────────────────────────────────────┘└────────────────────────────────────────────┘
✅ connected via /dev/ttyACM0 with RIOT Version: 2024.07-devel-135-g46924-feat/slip
```

Figure 8: The user interface of a running Jelly shell. The backwards compatible, plain text shell is seen on the left, labeled "Diagnostic Messages". The modern interactions, based on CoAP, can be found on the right under "Configuration Messages". Lastly, on the lower left, "Configuration" displays relevant information for the current context. The shown information was automatically feteched during start-up of Jelly. The observed truncation is due to adaption for the PDF export.

6 Outlook

Jelly is a proof of concept. As such, it is not completed software and not intended to be used in real work scenarios yet. Instead, the purpose of Jelly is to drive further evaluations of the proposed ideas, changes and evolutions for RIOT shell.

6.1 Technical Challenges

The proof of concept needs to answer the question of wether or not the extra work and complexity can satisfy the following questions:

- Is the flexibility and feature richness a good tradeoff with the downside of adding a new program (Jelly) to RIOT?
- Is the increased usability a good tradeoff with the downside of requiring SLIPMUX and a CoAP server on the RIOT node, that serve a shell?

Preliminary results (to be treated with caution) indicate that, when switching from a string based `ps` shell command to a binary based one, the increased ROM usage due to SLIPMUX and CoAP is smaller than the ROM usage for the otherwise required strings. Using a Jelly based `ps` saves ~200 bytes ROM. Further evaluations are needed.

So far left out of scope, the problem of interpretation and schema management will become a fundamental challenge. When binary messages are exchanged and ad-hoc converted into a human readable representation by the shell (Jelly), how does the shell know in which way the binary is to be interpreted? This is further complicated when taking versioning into consideration. For example, a 2024 RIOT version transmits the result of the `ps` command in binary. The format this RIOT node will use is the one present at its compile time. If the shell is expecting the `ps` format from 2025, which differs from the year old version, how can the binary payload be correctly converted into the human readable representation? Further research is needed.

Ideally, Jelly would not use custom formats to exchange data but is build upon preexisting standards. One promising option is CBOR [17]. The flexibility CBOR offers to encode data structures makes it a viable format to encode or decode shell commands and their arguments as well as the results. But again, further research is necessary.

Jelly currently only supports 64-Bit Linux as the host operating system. It needs to be evaluated, if Jelly can also be used from other operating systems, such as MacOS. Platform support is a strong requirement within the RIOT community.

6.2 Research on Programmer Experience

Many aspects and decisions made for Jelly are based on our point of view on RIOT and how it is used or interacted with. For example, the list of frequently required information in Section 4.3.1 is entirely based upon our regular work and may not adequately reflect the overall RIOT user base. To address this fundamental bias in Jelly, research within the RIOT community is needed. This might be in the form of questionaries, interviews or surveys. If Jelly is not tailored towards the actual work and the actual users, it will fail to be a useful once it is deployed, as highlighted by the fifth rule in Section 2.1.

An in complete suggestion, of which no qualitative nor quantitative answers are currently known:

- Why / when is the shell commonly used?
- What repetitive tasks are done using the shell?
- Are there annoyances or shortcomings with the shell?

Further, the proof of concept can be presented to the community to evaluate its design and usability effectiveness. Here the five rules of usability (Section 2.1) are checked by questioning a Jelly test-user in form of an interview:

1. Is Jelly usable without help?
2. Does Jelly impede an expert RIOT user?
3. Can Jelly aid newcomers to gain knowledge over RIOT-shell interactions?
4. Is using Jelly either more fun, easier, simpler or faster?
5. Does Jelly suit the actual work context?
6. Would a user adapt a new command to Jelly formats?

6.3 Even more Capabilities

This work focused on the shell replacement aspect of Jelly. However, a tool like Jelly offers to rethink the shell concept on a greater scale.

Complex shell commands can be implement entirely within Jelly by chaining smaller commands together. For example, instead of offering `ps`, RIOT could just offer the total number of threads and a command to query the properties of a given thread number. Jelly could then query the information of each thread individually. With all information collected, Jelly would generate and display the overview of all threads. While this example might not look impressive, think about it from the perspective of shell scripts, where multiple small commands are chained together to build a new command (the script). As Jelly is not bound by the hardware and software limitations of the constrained device, as RIOT is, there is potential to build large and more extensive commands. We believe there is demand, within the RIOT community for tooling that periodically extracts runtime information and stores it to disk for later analysis (academia, performance profiling, bug hunting, ..). Jelly might be a way to provide such functionality in a simple, extensible and reusable way.

As Jelly utilizes CoAP over serial, it is not hard to imagine a version of Jelly where CoAP over UDP is used. If Jelly manages to provide a way to slowly migrate the shell from strings to CoAP resources, without degrading the user experience, upgrading the shell to be operated over the network becomes a diligent, but routine piece of work. However, for a remotely available shell security considerations are foundational requirement. For example, while secure transport and application layer security might be available through existing work, like Datagram Transport Layer Security (DTLS) or Object Security for Constrained RESTful Environments (OSCORE), it might be challenging to provide user authentication and authorization.

Bibliography

[1] International Organization for Standardization, “ISO 9241-210: Ergonomics of human-system interaction - Part 11: Usability: Definitions and concepts.” 2019.

[2] L. A. D. L. LARRY L. C ONSTANTINE, *Software for use.* Pearson Education, Inc., 1999.

[3] P. S. P. Rex Hartson, *The UX Book*, 1st ed. Morgan Kaufmann, 2012.

[4] E. Baccelli *et al.*, “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

[5] J. Morales, C. Rusu, and D. Quiñones, “Programmer Experience: A Systematic Mapping,” *IEEE Latin America Transactions*, vol. 18, no. 6, pp. 1111–1118, 2020, doi: 10.1109/TLA.2020.9099749.

[6] RIOT OS, “Getting Started.” [Online]. Available: https://doc.riot-os.org/getting-started.html

[7] Z. Shelby, K. Hartke, and C. Bormann, “The Constrained Application Protocol (CoAP).” [Online]. Available: https://www.rfc-editor.org/info/rfc7252

[8] 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/

[9] “Jelly.” [Online]. Available: https://github.com/teufelchen1/jelly

[10] “Nonstandard for transmission of IP datagrams over serial lines: SLIP.” [Online]. Available: https://www.rfc-editor.org/info/rfc1055

[11] Alex Crichton, Steve Klabnik and Carol Nichols, with contributions from the Rust community, “The Cargo Book.” [Online]. Available: https://doc.rust-lang.org/cargo/index.html

[12] Steve Klabnik and Carol Nichols, with contributions from the Rust community, “The Rust Programming Language.” [Online]. Available: https://doc.rust-lang.org/book/ch00-00-introduction.html

[13] “serialport-rs.” [Online]. Available: https://docs.rs/serialport/latest/serialport/

[14] “serial-line-ip.” [Online]. Available: https://docs.rs/serial-line-ip/latest/serial_line_ip/

[15] “coap-lite.” [Online]. Available: https://docs.rs/coap-lite/latest/coap_lite/

[16] “ratatui.” [Online]. Available: https://ratatui.rs/

[17] C. Bormann, “Concise Binary Object Representation (CBOR) Sequences.” [Online]. Available: https://www.rfc-editor.org/info/rfc8742