Rust Programming Language

Jerry W. Rice, Principal Systems Software Consultant

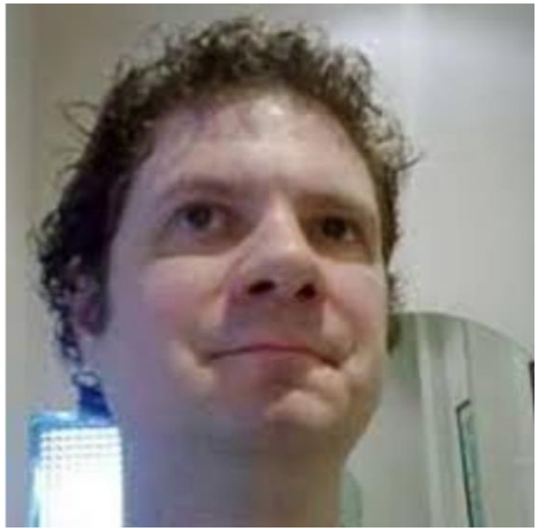
jerrywrice@fabnexus.com
www.fabnexus.com

Vers 0.3.2

Prelude

- Immediately following the slides, we’ll conduct a (live) Rust firmware demo using an STM32F091RC Nucleo development board.
- These slides will be available following today’s meeting.

History



- The Rust language began in 2006 as the personal open-source project of Mozilla software engineer Graydon Hoare - pictured above. His goal was a *modern* systems programming language, whose design incorporated techniques to eliminate ‘undefined behavior’ and other subtle reliability issues well understood from the c/c++ collective experience.
- In 2009 his results led Mozilla to sponsor and fund the project.

History (continued)

- The first stable Rust compiler tool-chain version (1.0) released in 2015 (Edition 2015).
- Additional Rust Editions were released in 2018, 2021, and just last month the 2024 Edition released.
- Mozilla sponsored the Rust project until mid-2020 (covid pandemic), when business conditions resulted in down-sizing at Mozilla.
- Following this, the Rust Foundation was launched in early 2021 https://foundation.rust-lang.org/

Overview of the Rust Language

- Rust is a high-level programming language suitable for firmware and operating system kernel/driver/module development.
- It's also used for run-time efficient infrastructure applications, network stacks, distributed software systems, cryptography, safety-critical automation, and aerospace and defense applications.
- Other than Rust, the 'c' language (1972) is the only major PL option in the systems programming domain. c++ (introduced in 1985) was designed as the successor to 'c', but recent history indicates for advanced low-level software efforts, c++ is at a disadvantage due to its legacy design and associated 'baggage'.

Overview of the Rust Language

- Rust is carefully designed to evolve/improve without abruptly obsoleting previously developed (Rust) software.
- To achieve this 'friendly' evolution the Rust language project uses Semantic Versioning (https://semver.org/) with its six (6) week rolling minor releases, and three (3) year Edition major releases.
- Rust Edition releases are the singular time that 'breaking' language changes will be introduced - with auto-migration tools provided.

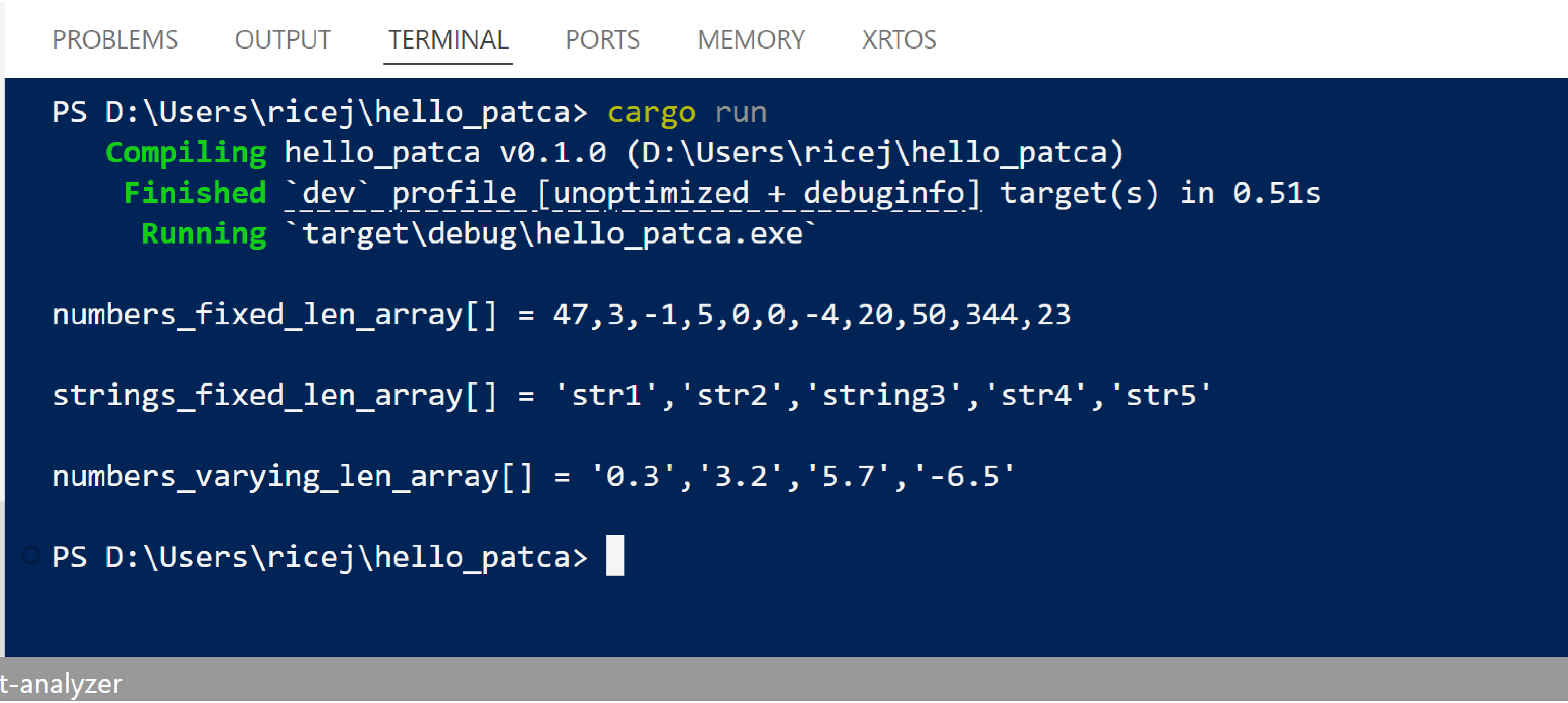
Overview of the Rust language (continued)

- To install the Rust Tool-chain on one's desktop computer visit the link https://www.rust-lang.org/tools/install
- There's a world-wide open-source ecosystem of community provided add-on Rust support tools and component libraries (called 'crates').
- Key examples of additional open-source Rust resources include:
  - Microsoft Visual Code IDE (with Rust extensions) https://code.visualstudio.com/download
  - Crates.io library registry: https://crates.io/

Overview of the Rust language (continued)

- Next let’s examine a basic Rust program’s source code (snippet).

```rust
fn main() { // hello_patca basic desktop Rust application
    fn get_id() -> i32 { 47 } // example local function with a i32 return value
    let numbers_fixed_len_array: [i32; 11] = [get_id(), 3, -1, 5, 0, 0, -4, 20, 50, 344, 23];
    let strings_fixed_len_array: [&str; 5] = ["str1", "str2", "string3", "str4", "str5"];
    print!("\nnumbers_fixed_len_array[] = ");
    for (position: usize, value: &i32) in numbers_fixed_len_array.iter().enumerate() {
        print!("{}", value);
        if position < numbers_fixed_len_array.len()-1 {
            print!(",");
        }
    }
    println!("");
    print!("\nstrings_fixed_len_array[] = ");
    for (position: usize, value: &&str) in strings_fixed_len_array.iter().enumerate() {
        print!("\'{}\'", value);
        if position < strings_fixed_len_array.len()-1 {
            print!(",");
        }
    }
    println!("");
    let numbers_varying_len_array: Vec<f64> = vec![0.3, 3.2, 5.7, -6.5];
    print!("\nnumbers_varying_len_array[] = ");
    for (position: usize, value: &f64) in numbers_varying_len_array.iter().enumerate() {
        print!("\'{}\'", value);
        if position < numbers_varying_len_array.len()-1 {
            print!(",");
        }
    }
    println!("\n");
} fn main
```



```
PS D:\Users\ricej\hello_patca> cargo run
   Compiling hello_patca v0.1.0 (D:\Users\ricej\hello_patca)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.51s
     Running `target\debug\hello_patca.exe`

numbers_fixed_len_array[] = 47,3,-1,5,0,0,-4,20,50,344,23

strings_fixed_len_array[] = 'str1','str2','string3','str4','str5'

numbers_varying_len_array[] = '0.3','3.2','5.7','-6.5'

PS D:\Users\ricej\hello_patca> 
```

Overview of the Rust language (continued)

- By convention Rust public crates (libraries) use the MIT and/or Apache licenses. The Rust tool-chain itself also uses these licenses. These licenses do not incur the ‘copy left’ constraints of GPLx licenses, thus developers may create Rust code which is proprietary closed-source, while leveraging available open-source Rust code.
- While many Rust software tools/libraries are free open-source, some high-value Rust software packages are privately developed and have commercial licenses.

Overview of the Rust language (continued)

- The Rust tool-chain consists of the following integrated component tools
  - ‘rustc’ => The Rust tool-chain cross-platform compiler
  - ‘cargo’ => The Rust tool-chain build-manager
  - ‘rustup’ => The Rust tool-chain installer and updater
  - ‘clippy’ => The Rust tool-chain lint static analyzer

Overview of the Rust language (continued)

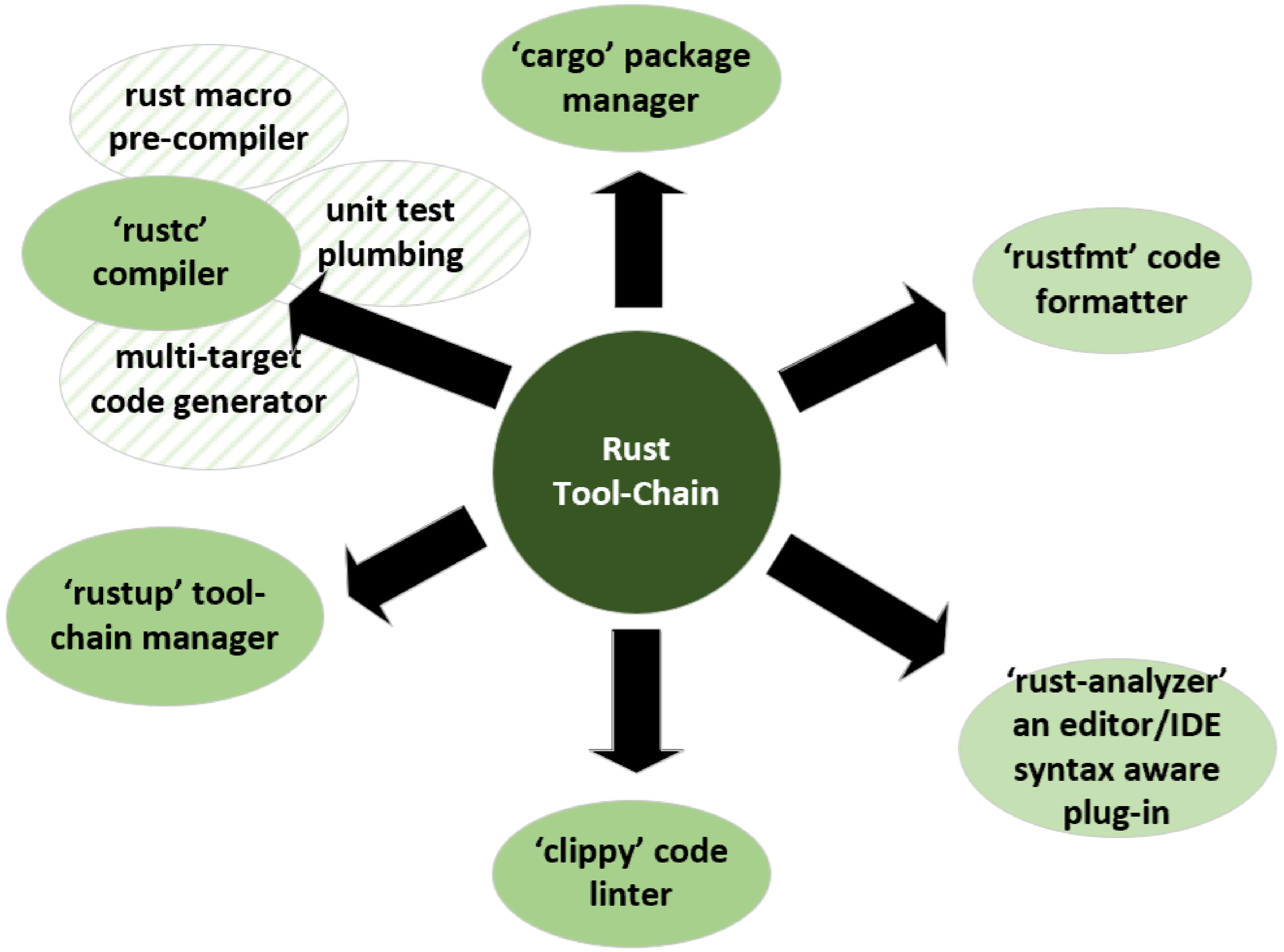
- ‘rustfmt’ => The Rust tool-chain source-code doc tool

‘rust-analyzer’ is a free add-on provided by Ferrous Systems and the Rust open-source community. It’s a Rust source syntax editor/debugger plug-in.

The ‘cargo’ tool provides an integrated unit-test facility, and ‘rustc’ has a powerful macro code generation capability: see https://doc.rust-lang.org/book/ch20-06-macros.html

- As we’ll see later, Rust macros are used extensively throughout the Rust standard library, and all other non-embedded Rust code.
- In practice developers infrequently create new macros and more frequently use available published/documented macros which have been extensively verified/tested.

Overview of the Rust Language (continued)



Overview of the Rust language (continued)

- The cross-platform Rust tool-chain downloads and runs on MS Windows, Linux, and Apple Mac Os desktops.
- The latest stable Rust tool-chain continues to build/run legacy Rust projects published as early as Rust's initial 1.0 stable release.
- The Rust compiler team releases new (minor version) tool-chain upgrades on a ~six (6) week cycle.

Overview of the Rust language (continued)

- The Rust tool-chain generates binary code for a large collection of target platforms (271 targets as of version 1.85).
- Each project's target is specified in its cargo project file(s), or defaults to the current build host target.
- When building for a target other than one's development host, the resulting binary code files must be loaded/copied to the target platform (or emulation environment) before execution.
- Rust targets span a wide range of hardware platforms: Arm32/64, x86-64, Riskv32/64, ESP, NRFxxxx, RPxxxx, MIPS, and some others.
- For the full target list, run the terminal command

  'rustc --print target-list' .

Overview of the Rust language (continued)

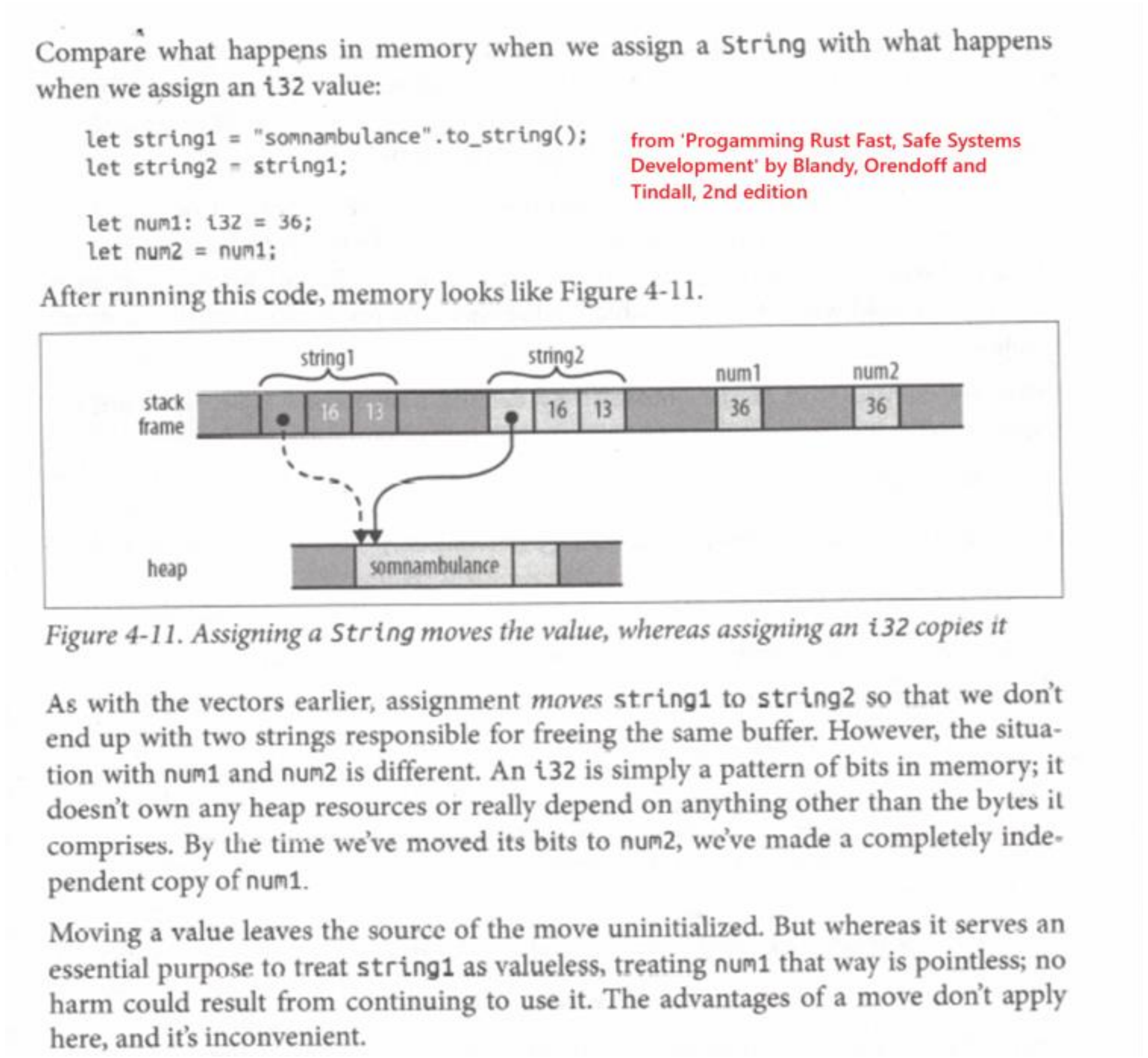
- Rust data and concurrency safety ‘guard rails’ are very beneficial.
- These consist of Rust ‘ownership system’ with ‘borrow checker’, ‘data life-time’ tracking, and data reference mutability rules. The Rust compiler prevents *dangling pointers*, *data races*, and *reference-use-after-free* issues by rejecting and reporting source code sequences that contain these issues.
- Such error scenarios can be ‘subtle’ and are sometimes difficult for a developer to immediately recognize without the compiler expressly erroring and reporting them. The Rust compiler’s error messages are high quality and well crafted.
- This is why developers new to Rust sometimes express frustration when initially ‘fighting with the compiler’s borrow-checker’. Nevertheless, for those who persevere familiarity and time mitigates this annoyance.

Overview of the Rust language (continued)

- Rust allows low-level code with direct access to hardware registers, raw memory addresses, DMA controllers and such.
- The Rust annotated ‘unsafe’ keyword supports this ability. Only in ‘unsafe’ code blocks or functions does the Rust compiler permit memory address reference operations with its ‘raw pointer’ feature.

Overview of the Rust Programming Language (continued)

- Rust data values use either static, stack or heap-allocated memory.
- Rust heap-allocated data-items are not deallocated via a run-time ‘garbage collection’, nor overtly by a developer supplied command.
- At compile-time the Rust compiler silently injects assembler code that efficiently deallocates each heap allocated data variable – typically when its ‘owner’ exits (falls out of) scope.
- Briefly peaking ‘under the covers’: Heap-based values have a hidden stack allocated ‘Fat Pointer’ control block which contains the internal (private) address to the data value’s contiguous heap-memory space. The hidden control block also has a ‘current-size’ and ‘capacity’ (integer) field. See next slide.



Compare what happens in memory when we assign a `String` with what happens when we assign an `i32` value:

```
let string1 = "somnambulance".to_string();
let string2 = string1;

let num1: i32 = 36;
let num2 = num1;
```

from 'Progamming Rust Fast, Safe Systems Development' by Blandy, Orendoff and Tindall, 2nd edition

After running this code, memory looks like Figure 4-11.

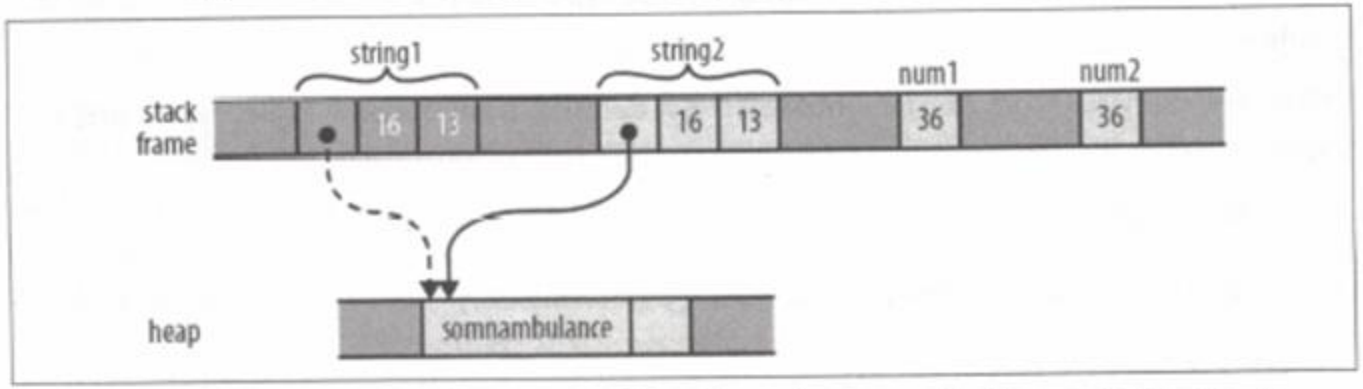


Figure 4-11. Assigning a `String` moves the value, whereas assigning an `i32` copies it

As with the vectors earlier, assignment *moves* `string1` to `string2` so that we don't end up with two strings responsible for freeing the same buffer. However, the situation with `num1` and `num2` is different. An `i32` is simply a pattern of bits in memory; it doesn't own any heap resources or really depend on anything other than the bytes it comprises. By the time we've moved its bits to `num2`, we've made a completely independent copy of `num1`.

Moving a value leaves the source of the move uninitialized. But whereas it serves an essential purpose to treat `string1` as valueless, treating `num1` that way is pointless; no harm could result from continuing to use it. The advantages of a move don't apply here, and it's inconvenient.

Overview of the Rust Programming Language (continued)

- The Rust compiler uses the LLVM infrastructure for its target code generation.
- This facilitates Rust multi-platform target code generation, by leveraging LLVM’s intra-module and cross-module optimization functionality.
- Compiled Rust binary executables are statically linked by default. On hosted Rust platforms, dynamically linked libraries are an option.

Overview of the Rust Programming Language (continued)

- Rust has generic data types, struct/enum/tuple composition fundamental declared types, fixed-size arrays and slices.
- Rust heap allocated resizable arrays are provided via the generic std::Vec<T> type.
- Threads, the allocator, and certain other useful Rust features are provided by its standard library. This helps reduce compiler complexity.
- The Rust language emphasizes a functional programming style rather than the classic procedural approach. Many Rust 'statements' are also expressions, only terminated with a semicolon.
- Rust does not support exceptions/exception handlers. It instead relies on its standard Option and Result types, coupled with its 'match' or other conditional statements. All errors are reported as result statuses.
- Rust panic!() facility is actually a 'notify and halt' debug feature.

Rust’s Async Programming Model

- Rust provides an opt-in ‘asynchronous’ mode of program flow sequencing. Its asynchronous task feature use ‘async’ keyword prefixed function declarations, the ‘await’ operator, async Future trait objects, a ‘poll’ and ‘waker’ item, as well as a library supported async executor mechanism.
- Rust async provides concurrent execution by way of multiplexed ‘async tasks’, which do not incur the costly run-time overhead of conventional threads-based context-switching.

https://rust-lang.github.io/async-book/

Rust Embedded App for bare metal target

```
// main.rs

#![no_std] // don't link the Rust standard library
#![no_main] // disable all Rust-level entry points

use core::panic::PanicInfo;

#[no_mangle] // don't mangle the name of this function
pub extern "C" fn _start() -> ! {
    // this function is the entry point, since the linker looks for a function
    // named `_start` by default
    loop {}
}

/// This function is called on panic.
#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
```

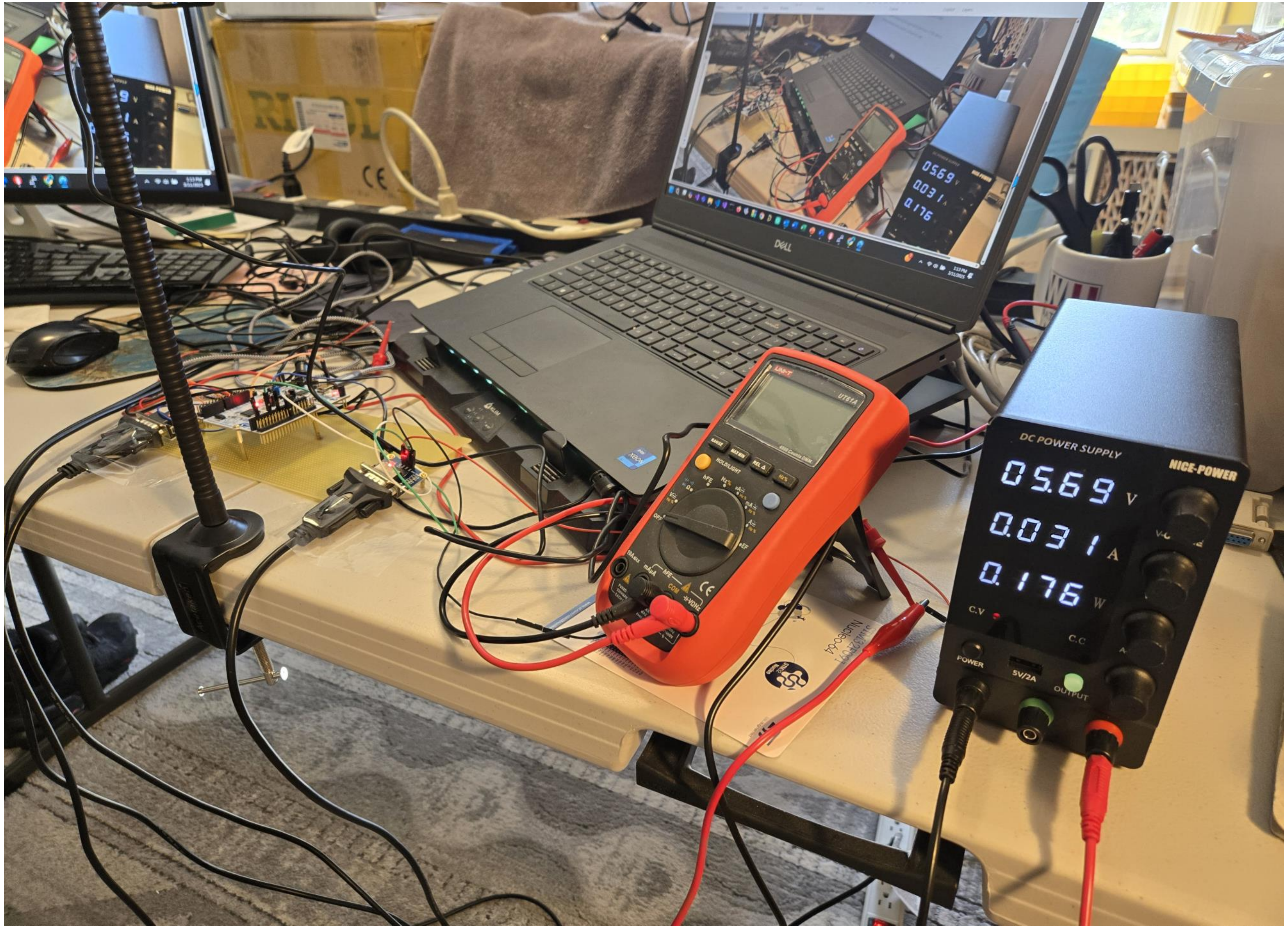
Rust Embassy Embedded Async App

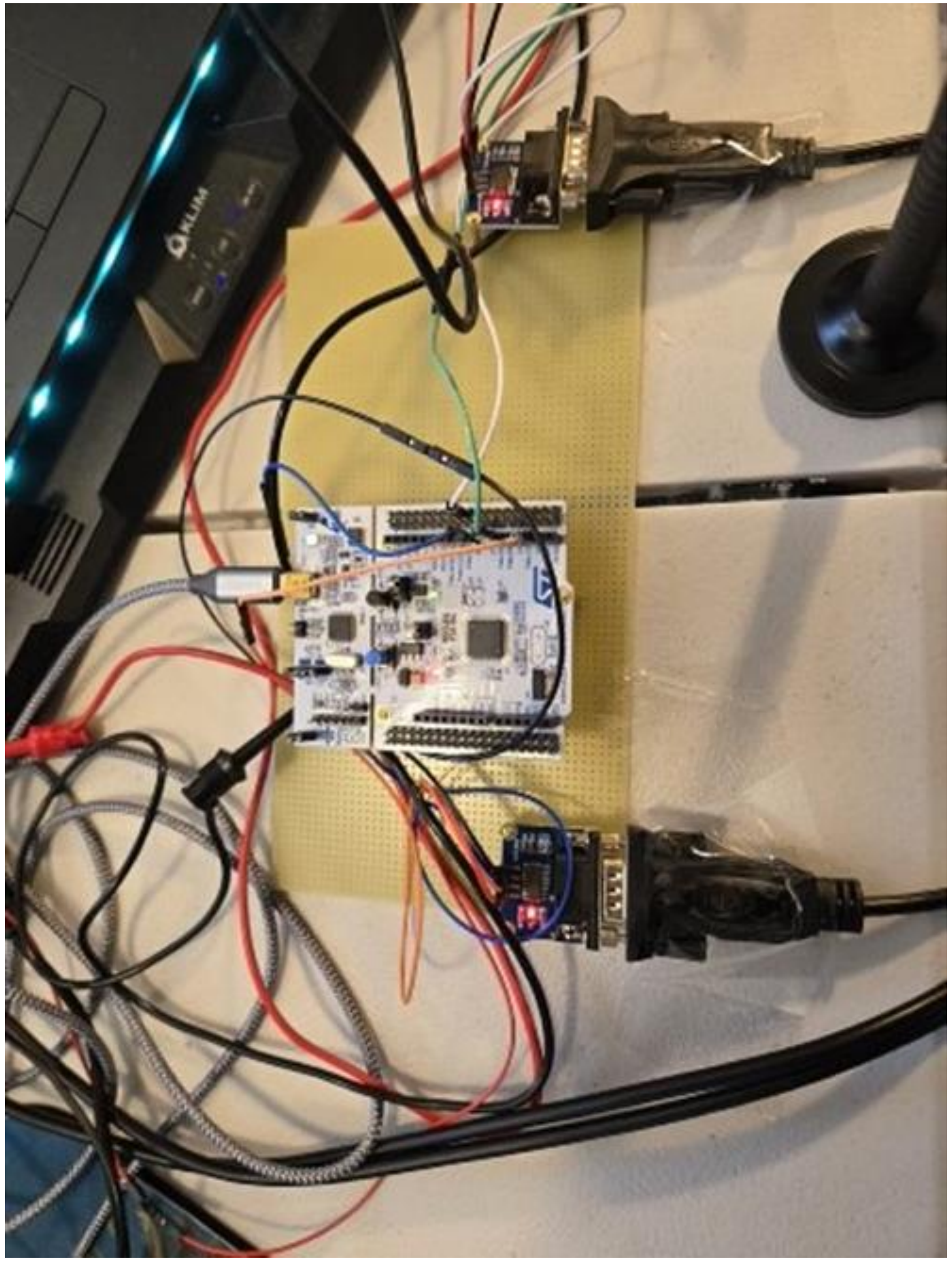
Demo time! The demo consists of a Rust embassy async firmware app running on a STM32 ARM M0 low-power dev board.

Embassy reference: https://embassy.dev/book/

First let’s view an image of the hardware setup (next slide)

Then a high-level block-diagram of our firmware’s architecture


DC POWER SUPPLY
NICE-POWER
05.69 V
0.031 A
0.176 W
C.V
C.C
POWER
5V/2A
OUTPUT
UNI-T
UT61A
NUCLEO-64



STM32F091RC M0 ARM 32 Kb SRAM 256 Kb Flash

Embassy async executor support and startup library

demo pseudo Rust firmware code

```
Embassy async main ()
   Init interrupts, DMA handlers, peripherals
   Init/config 1st RS-232 uart (with DMA)
   Init/config 2nd RS-232 uart (DMA)
   Spawn async task ‘led_task’
   Spawn async task ‘button_monitor_task’
   Spawn async task ‘cmd_menu_task’
   Spawn async task ‘uart_continous_write_task’

   loop
        { print button mode changes to debug out }
```

```
Embassy async task ‘led_task’:

loop {
        check blink rate setting (fast/slow)
        if slow blink rate {
             do slow blink seq
        }
        } else {
             do fast blink seq
        }
}
```

```
Embassy async task ‘button_monitor_task’:

loop {
     await button ‘down’ event
     capture system timer value
     await  button ‘up’ event
     capture new system timer value
     calc elapsed button down time in ms
     if button down time > 1000 ms {
           toggle current blink rate mode
     } else {
           do nothing
     }
}
```

```
Embassy async task ‘cmd_menu_task’:

loop {
     write cmd prompt to 1st uart
     read (kb) input for user response
     if ‘1’ {
          perform stack hi-water mark scan
     } else if ‘2’ {
          perform second dummy cmd
     } else /* invalid input */
          write error msg to (first) uart
     }
}
```

```
Embassy async task ‘uart_continous_write_task’:

loop {
     loop until 1 Mb sent out 2nd uart {
          revise next sequenced ~150 byte text msg  buffer
          write text msg buffer to 2nd uart
     }
     send repeat (or exit) prompt to 2nd uart
     read (kb) input for remote serial user response
     if ‘Y’ {
          continue loop
     } else
          break; // exit this async task
     }
}
```

Rust: Areas of Continued Language Evolution

- Rust’s async programming model is an area of continued debate and study by the Rust community. It will certainly evolve and improve in the future.
- Rust’s type system, generics, traits, and type bounds expressivity are areas of continued examination and improvement.
- Self-referencing struct fields and the Pin and !Pin reference annotations, are a matter of complexity which may be revised/simplified.
- Compiler error messages are an on-going area of improvement.

When is Rust a Reasonable Option?

- The project should have a capable Rust developer (or team). Some organizations have invested in hiring outside (new) Rust developers, or training in-house developers.
- Rust is an excellent option if your project is a forward-looking research and/or experimental or training type development. This often ties directly to the prior bullet item.
- Rust is a reasonable option when the project is a (well-funded) ‘commercial grade’ project‘ that is starting with a clean state (no legacy software constraints).

When is Rust a Reasonable Option (continued)?

- Rust offers much to software projects which require high reliability and/or safety-critical operation.
- If software portability and reliability are highly desirable for your project, Rust may be a good option.
- Rust is a niche tool/language, and IMO seems unlikely to diminish in importance in the foreseeable future. It's still early enough in its adoption and business 'life cycle' that there should be ample opportunities for emerging small companies which astutely understand and navigate this technology sector.

Potential reasons for not using Rust

- For poorly funded, or very tight completion schedules, or burdensome legacy constraints, Rust is not likely a reasonable option.
- If an organization’s management is hesitant/conservative and normally reluctant to adopt newer/fresher technologies, then Rust is likely not a suitable language choice. Rust, while gaining popularity and gaining more use in recent years, is in certain aspects still a ‘developing/early-stage’ technology. A given Rust project may in-fact need to make course corrections and contingency actions should specific Rust ‘maturity related issues’ arise.

Reasons for not using Rust

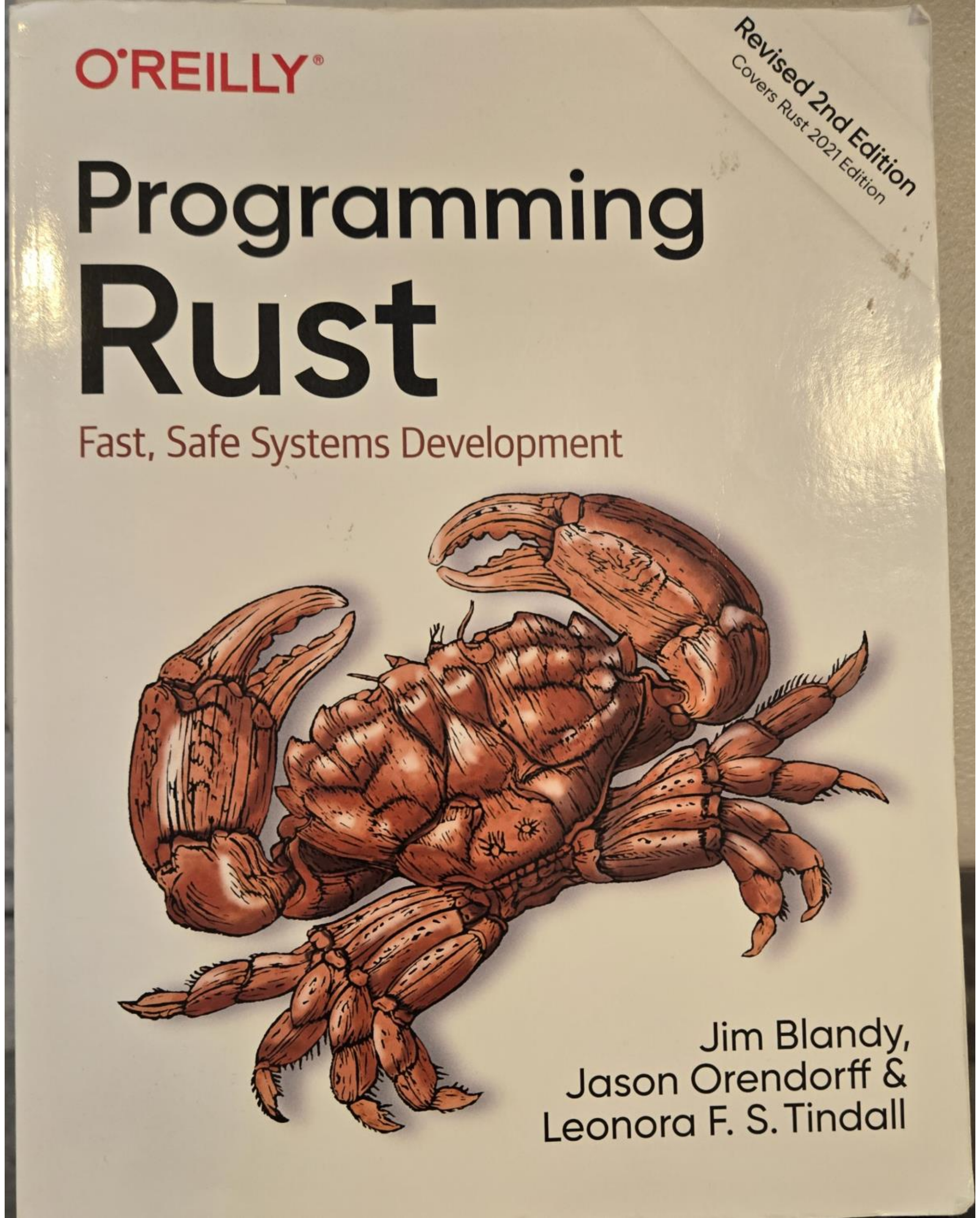
- There are certain software niches where Rust currently lags in terms of specific crate support. Full-featured GUI frameworks are the most notable example. While GUI options have improved for Rust in the past couple of years with SLint (https://slint.dev/) emerging, and certain Rust Web-app frameworks having been published, but new Rust projects requiring a sophisticated multi-page in-process GUI should carefully evaluate this topic. There are bridging crates for integrating Qt, Gtk, Windows native, and other legacy GUI frameworks into Rust applications, but such bridging has complexity issues of its own. This is an area of concern for the near term.

Reasons for not using Rust (continued)

- Rust isn’t designed for ‘rapid prototyping’ of large software systems. Rust is intended for and has a development model designed for producing reliable and maintainable software (a different emphasis than rapid prototyping). Rust can be used for prototyping if applicable pre-existing Rust source code can be readily adapted. This option is made more likely by the sheer quantity of available open-source code in public repos.

Reasons for not using Rust (continued)

- If a project requires interfacing to legacy libraries from another programming language, one should carefully evaluate the available options for this. Rust provides built-in support for interfacing directly to C code, but interfacing to c++ is more complex. Interfacing with non-c languages is often about or more complex as interfacing to.
- On this topic, there is a significant effort underway focused on generating tools and techniques that assist when calling Rust code from C++, and Rust code calling into C++ libraries. This effort is funded by Google in collaboration with the Rust Foundation. It's fairly early stage at this time.



O'REILLY

Revised 2nd Edition
Covers Rust 2021 Edition

Programming Rust

Fast, Safe Systems Development

Jim Blandy,
Jason Orendorff &
Leonora F. S. Tindall

Additional Recommended Reading/Viewing

- https://doc.rust-lang.org/book/
- https://www.amazon.com/Programming-Rust-Fast-Systems-Development/dp/1492052590
- https://www.amazon.com/Rust-Rustaceans-Programming-Experienced-Developers-ebook/dp/B0957SWKBS
- https://www.youtube.com/watch?v=8O0Nt9qY_vo&t=6123s
- https://blog.rust-lang.org/inside-rust/index.html