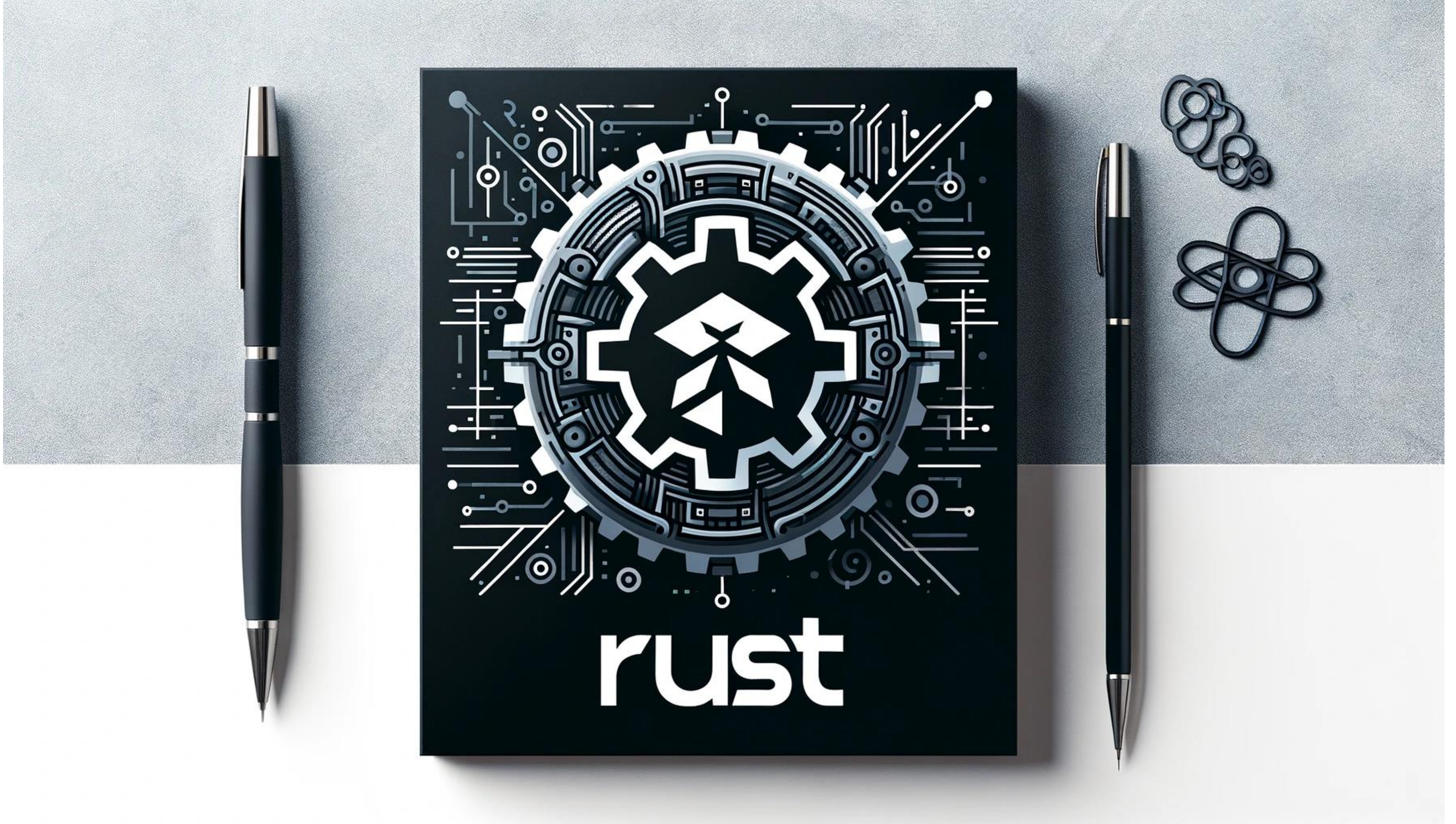
Zephyr® Project
Developer Summit

Rust on Zephyr

Status and State

THE LINUX FOUNDATION #EmbeddedOSSummit @handle


rust

Last Year

- Rust on Zephyr: Hello World
- Very manual process: Issues with linking llvm/gcc

Where I’m at now

- rust-embedded keyboard firmware
- ported to Zephyr
- Still a hack
- But, a lot is there: Mutex/Condition vars, threads, gpio, USB, Uart
- Lots of C wrappers for things

SEGGER

The RFC

https://github.com/zephyrproject-rtos/zephyr/issues/65837

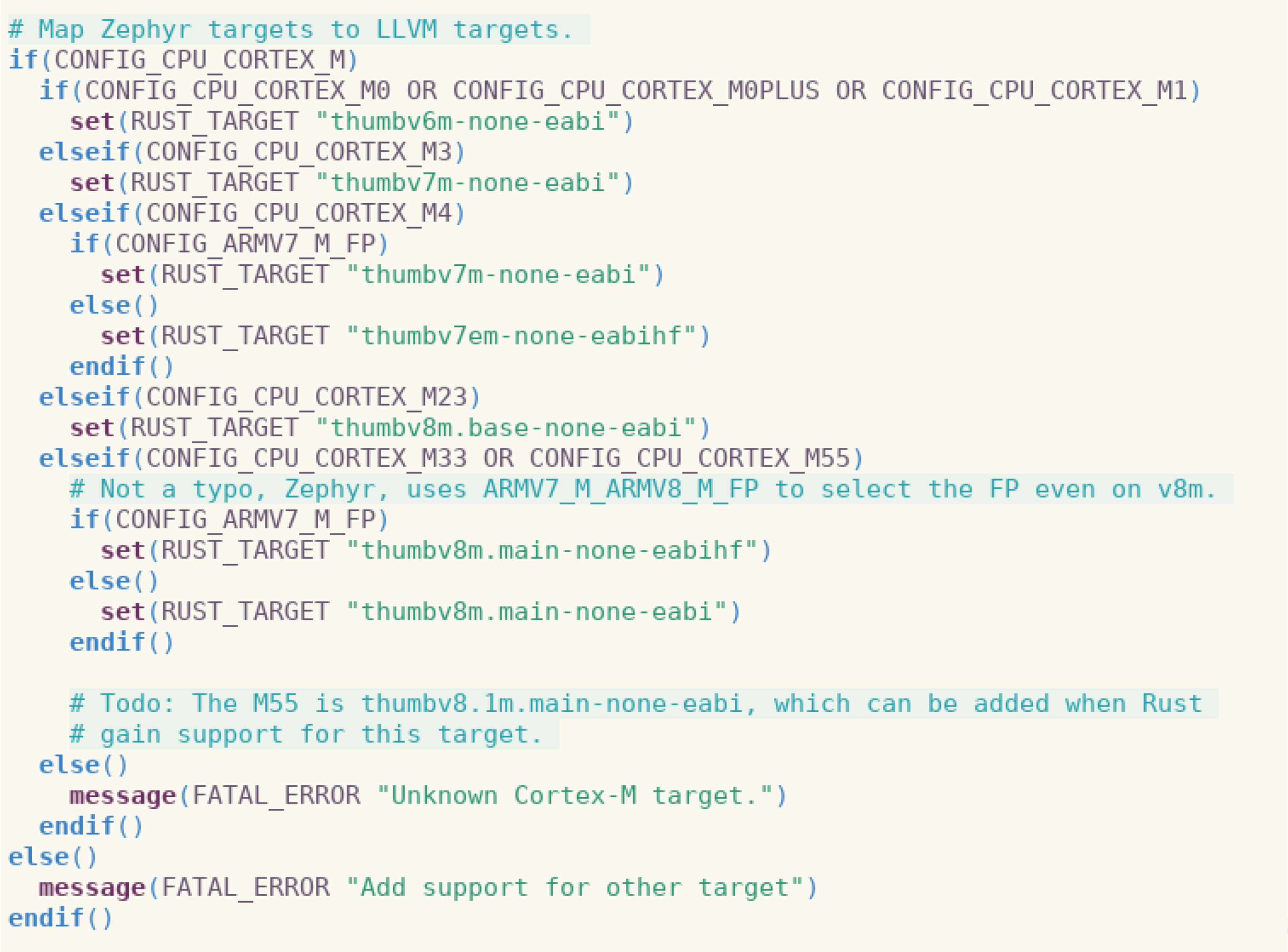
- Goal is application development in Rust (Zephyr code in rust would be a separate effort)
- Should be integrated in Zephyr tree
- Application should feel like Rust, but also not be too distant from Zephyr

Building: cmake

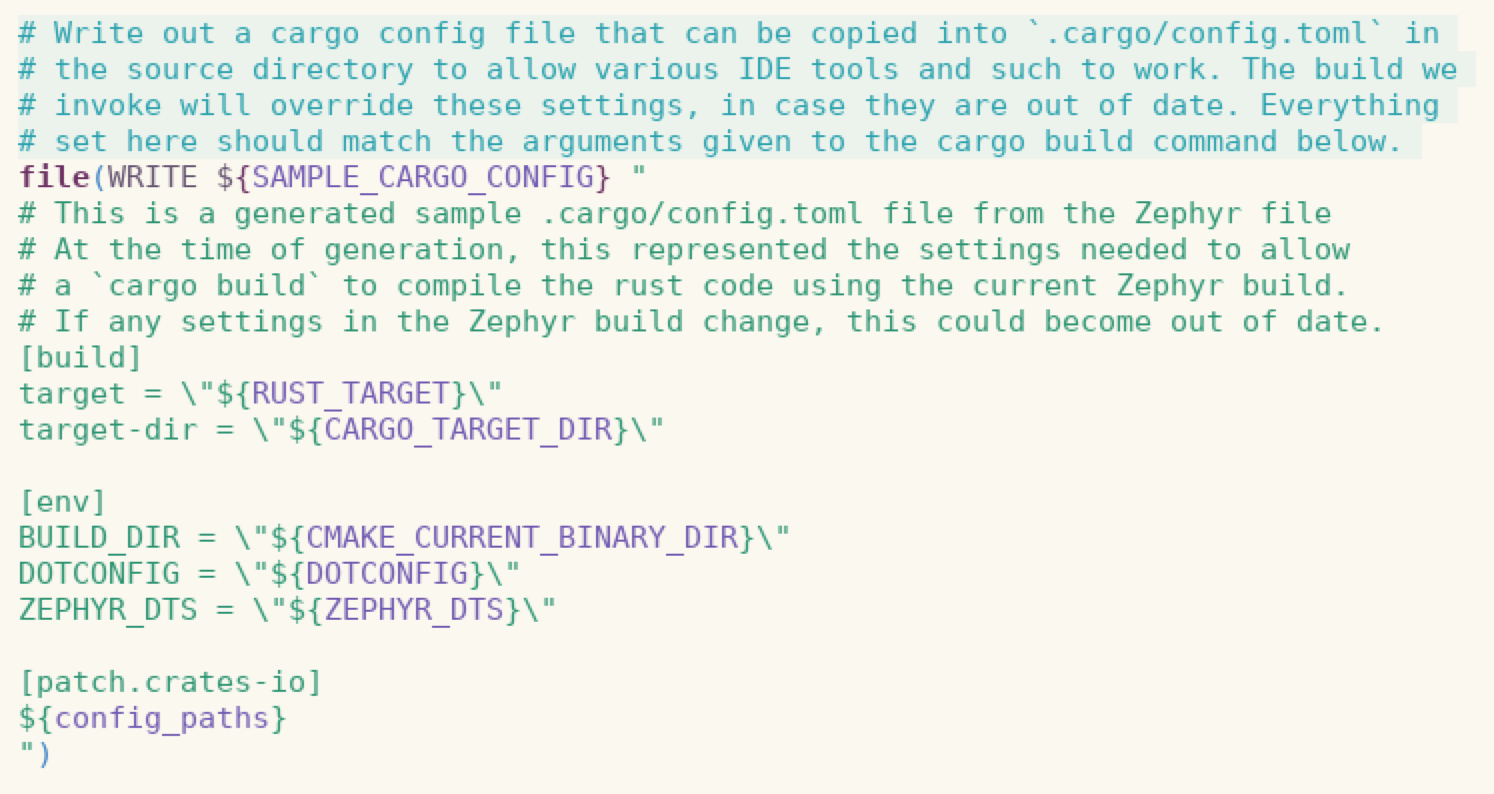
- Zephyr cmake builds “owns” the build process.
- cargo also wants to “own” the build.
- We let cargo build rust code, with cmake giving a bunch of args:
    - Build goes in Zephyr build directory, not just ‘target’.
    - Allows cargo/crate ecosystem for app.
    - Allows IDEs/rust-analyzer to just work (with some help).
    - Cargo builds a ‘.a’ which cmake links in

cmake/cargo: Under the hood

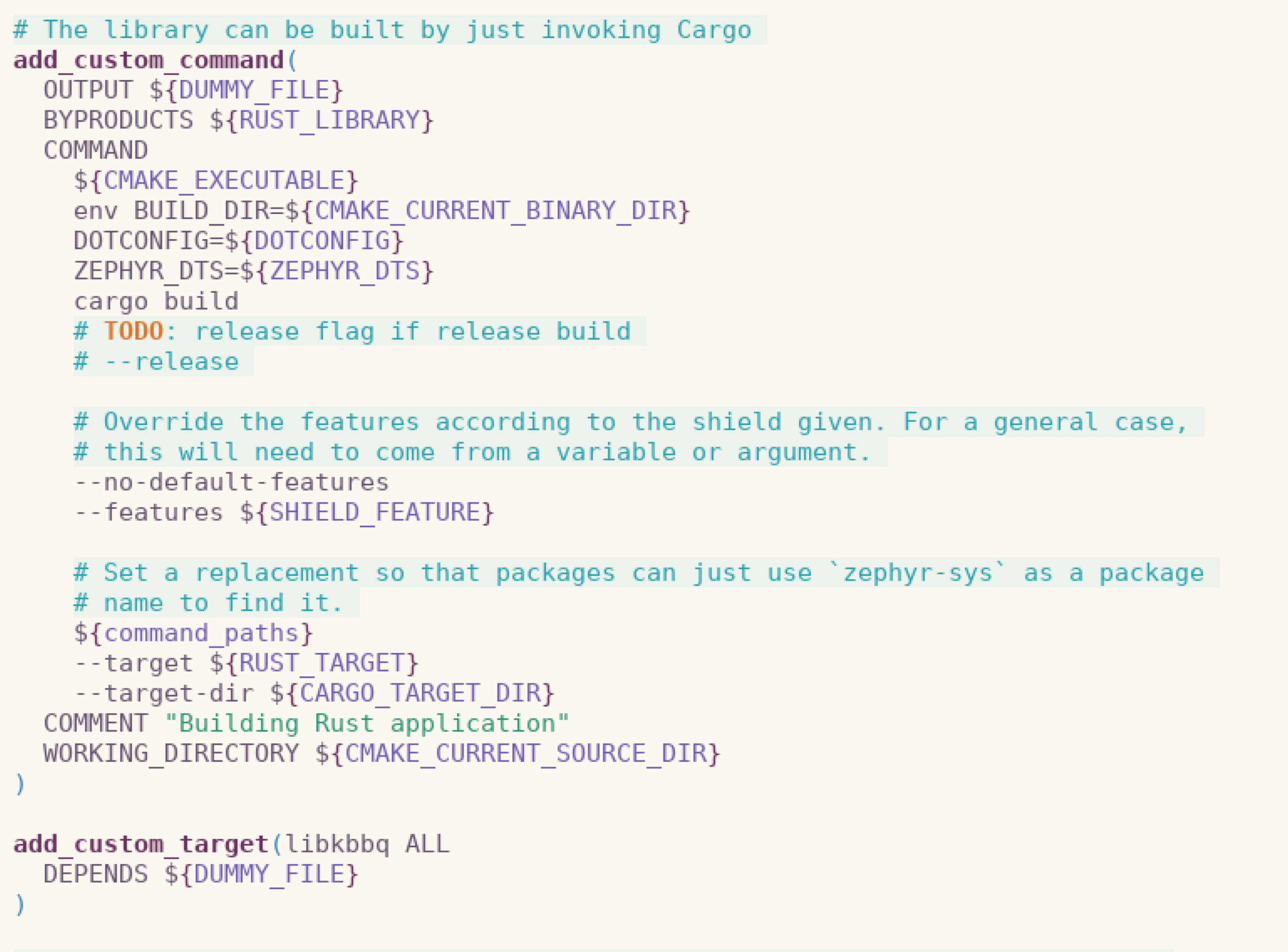
- cmake maps between Zephyr build targets and rust/llvm targets
- Adds path overrides so cargo finds zephyr provided crates
- Generates a template .cargo/config.toml to allow rust-analyzer/IDE’s to work (or just `cargo check`)
- The CMakeLists.txt in the app is small, similar to a C app.
- West build works, targets, etc.



```
# Map Zephyr targets to LLVM targets.
if(CONFIG_CPU_CORTEX_M)
  if(CONFIG_CPU_CORTEX_M0 OR CONFIG_CPU_CORTEX_M0PLUS OR CONFIG_CPU_CORTEX_M1)
    set(RUST_TARGET "thumbv6m-none-eabi")
  elseif(CONFIG_CPU_CORTEX_M3)
    set(RUST_TARGET "thumbv7m-none-eabi")
  elseif(CONFIG_CPU_CORTEX_M4)
    if(CONFIG_ARMV7_M_FP)
      set(RUST_TARGET "thumbv7m-none-eabi")
    else()
      set(RUST_TARGET "thumbv7em-none-eabihf")
    endif()
  elseif(CONFIG_CPU_CORTEX_M23)
    set(RUST_TARGET "thumbv8m.base-none-eabi")
  elseif(CONFIG_CPU_CORTEX_M33 OR CONFIG_CPU_CORTEX_M55)
    # Not a typo, Zephyr, uses ARMV7_M_ARMV8_M_FP to select the FP even on v8m.
    if(CONFIG_ARMV7_M_FP)
      set(RUST_TARGET "thumbv8m.main-none-eabihf")
    else()
      set(RUST_TARGET "thumbv8m.main-none-eabi")
    endif()

    # Todo: The M55 is thumbv8.1m.main-none-eabi, which can be added when Rust
    # gain support for this target.
  else()
    message(FATAL_ERROR "Unknown Cortex-M target.")
  endif()
else()
  message(FATAL_ERROR "Add support for other target")
endif()
```



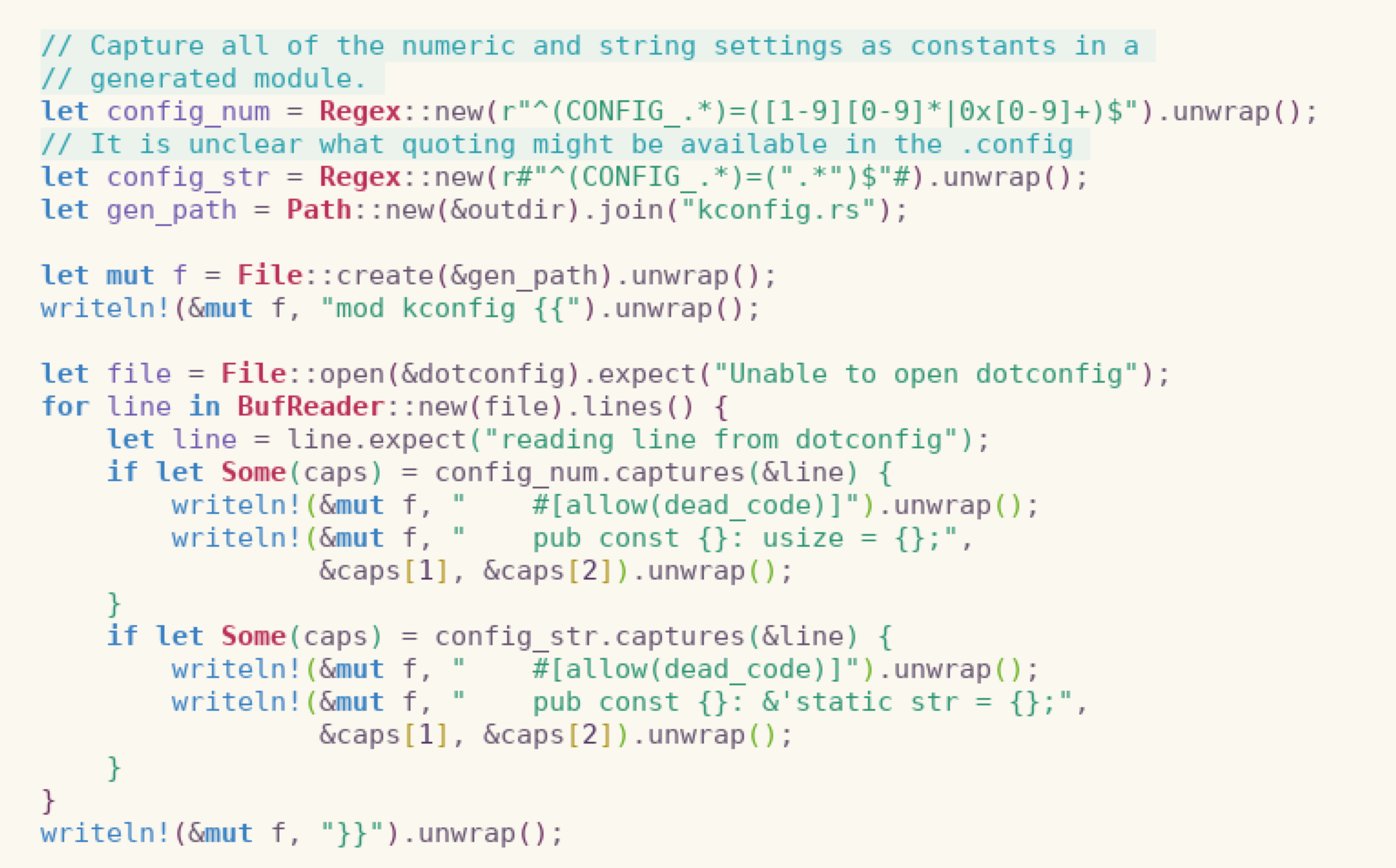
```
# Write out a cargo config file that can be copied into `.cargo/config.toml` in
# the source directory to allow various IDE tools and such to work. The build we
# invoke will override these settings, in case they are out of date. Everything
# set here should match the arguments given to the cargo build command below.
file(WRITE ${SAMPLE_CARGO_CONFIG} "
# This is a generated sample .cargo/config.toml file from the Zephyr file
# At the time of generation, this represented the settings needed to allow
# a `cargo build` to compile the rust code using the current Zephyr build.
# If any settings in the Zephyr build change, this could become out of date.
[build]
target = \"${RUST_TARGET}\"
target-dir = \"${CARGO_TARGET_DIR}\"

[env]
BUILD_DIR = \"${CMAKE_CURRENT_BINARY_DIR}\"
DOTCONFIG = \"${DOTCONFIG}\"
ZEPHYR_DTS = \"${ZEPHYR_DTS}\"

[patch.crates-io]
${config_paths}
")
```



```
# The library can be built by just invoking Cargo
add_custom_command(
  OUTPUT ${DUMMY_FILE}
  BYPRODUCTS ${RUST_LIBRARY}
  COMMAND
    ${CMAKE_EXECUTABLE}
    env BUILD_DIR=${CMAKE_CURRENT_BINARY_DIR}
    DOTCONFIG=${DOTCONFIG}
    ZEPHYR_DTS=${ZEPHYR_DTS}
    cargo build
    # TODO: release flag if release build
    # --release

    # Override the features according to the shield given. For a general case,
    # this will need to come from a variable or argument.
    --no-default-features
    --features ${SHIELD_FEATURE}

    # Set a replacement so that packages can just use `zephyr-sys` as a package
    # name to find it.
    ${command_paths}
    --target ${RUST_TARGET}
    --target-dir ${CARGO_TARGET_DIR}
  COMMENT "Building Rust application"
  WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
)

add_custom_target(libkbbq ALL
  DEPENDS ${DUMMY_FILE}
)
```

Kconfig

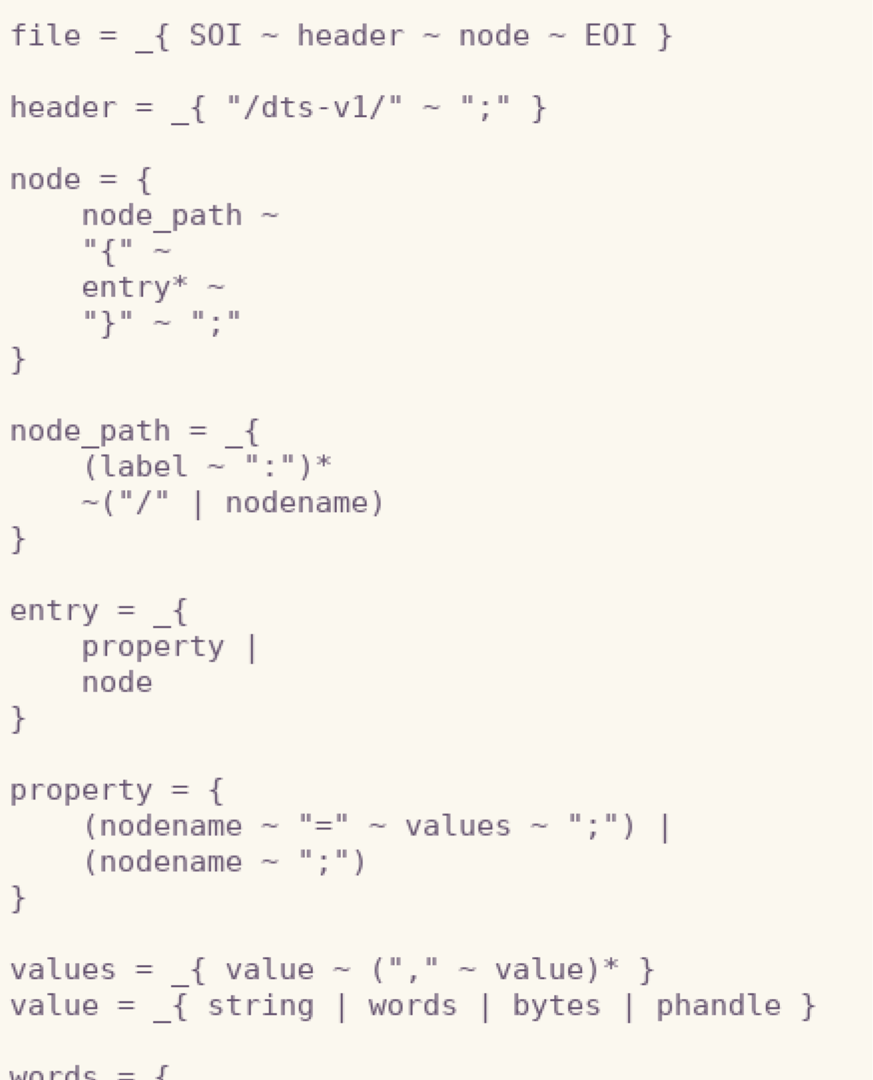
- A provided crate and a short build.rs process the Kconfig results
- Mostly will be used within zephyr-sys, app only needs if it wants to conditionalize on Kconfig options
- Boolean Kconfig options become cfg to Rust #[cfg(CONFIG_FOO_BAR)] works.
- Numeric and string become constants within zephyr-sys



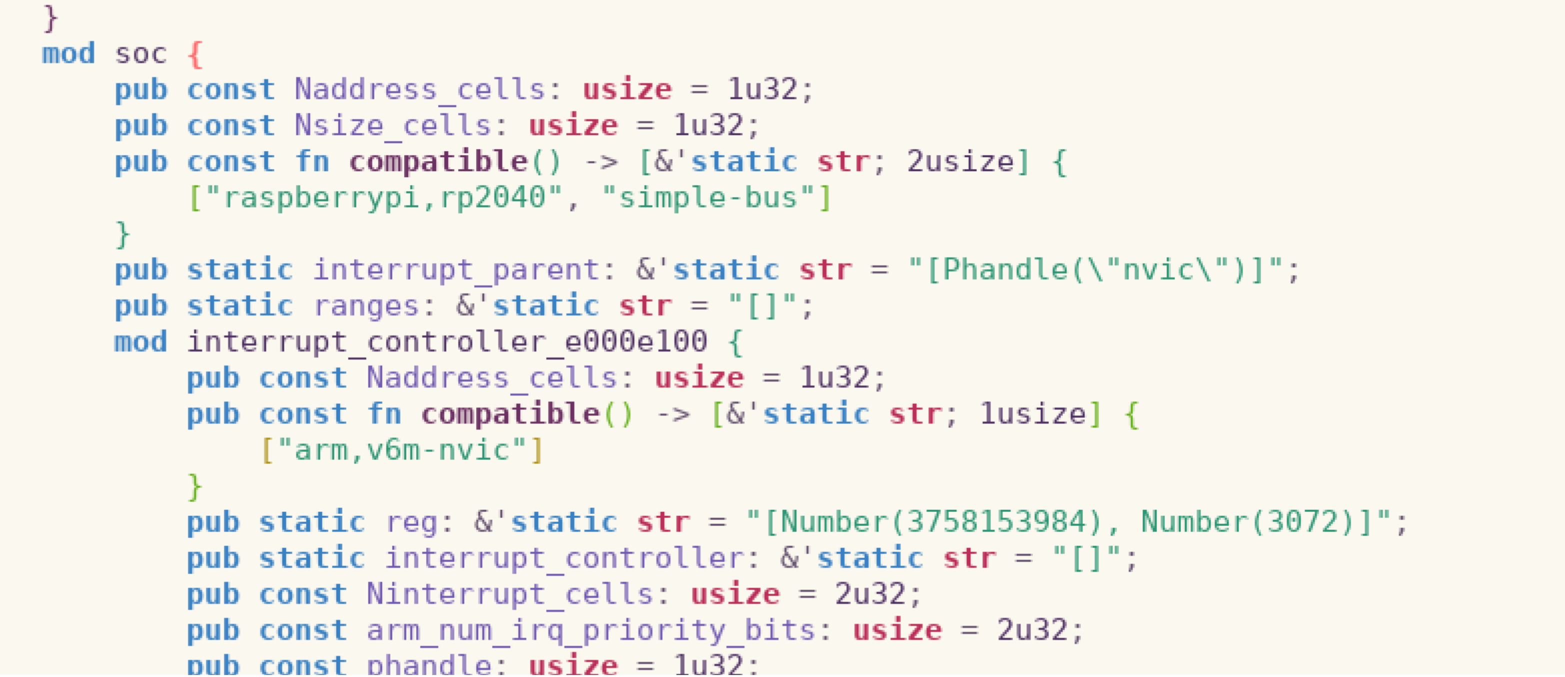
```rust
// Capture all of the numeric and string settings as constants in a
// generated module.
let config_num = Regex::new(r"^(CONFIG_.*)=([1-9][0-9]*|0x[0-9]+)$").unwrap();
// It is unclear what quoting might be available in the .config
let config_str = Regex::new(r#"^(CONFIG_.*)=(".*")$"#).unwrap();
let gen_path = Path::new(&outdir).join("kconfig.rs");

let mut f = File::create(&gen_path).unwrap();
writeln!(&mut f, "mod kconfig {{").unwrap();

let file = File::open(&dotconfig).expect("Unable to open dotconfig");
for line in BufReader::new(file).lines() {
    let line = line.expect("reading line from dotconfig");
    if let Some(caps) = config_num.captures(&line) {
        writeln!(&mut f, "    #[allow(dead_code)]").unwrap();
        writeln!(&mut f, "    pub const {}: usize = {};",
                 &caps[1], &caps[2]).unwrap();
    }
    if let Some(caps) = config_str.captures(&line) {
        writeln!(&mut f, "    #[allow(dead_code)]").unwrap();
        writeln!(&mut f, "    pub const {}: &'static str = {};",
                 &caps[1], &caps[2]).unwrap();
    }
}
writeln!(&mut f, "}}").unwrap();
```

Devicetree

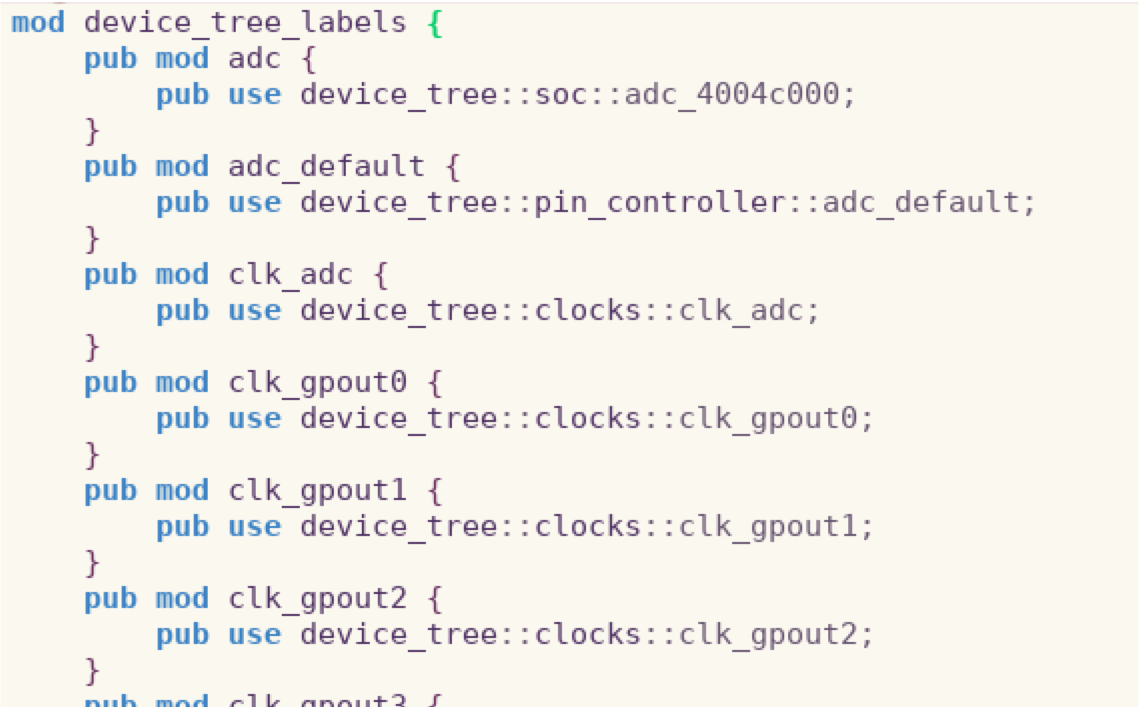
- DT converted to lots of #defines in C. Name stitching used to represent path.
- In Rust, we have modules, use those to map to hierarchy of DT.
- Some things are just const in Rust. No code generated.
- Other things become const fn, also not gen unnecessary code.
- Aliases, phandles, are other modules with a `pub use …` to make names appear in new place in hierarchy.



```
file = _{ SOI ~ header ~ node ~ EOI }

header = _{ "/dts-v1/" ~ ";" }

node = {
    node_path ~
    "{" ~
    entry* ~
    "}" ~ ";"
}

node_path = _{
    (label ~ ":")*
    ~("/" | nodename)
}

entry = _{
    property |
    node
}

property = {
    (nodename ~ "=" ~ values ~ ";") |
    (nodename ~ ";")
}

values = _{ value ~ ("," ~ value)* }
value = _{ string | words | bytes | phandle }

words = {
```



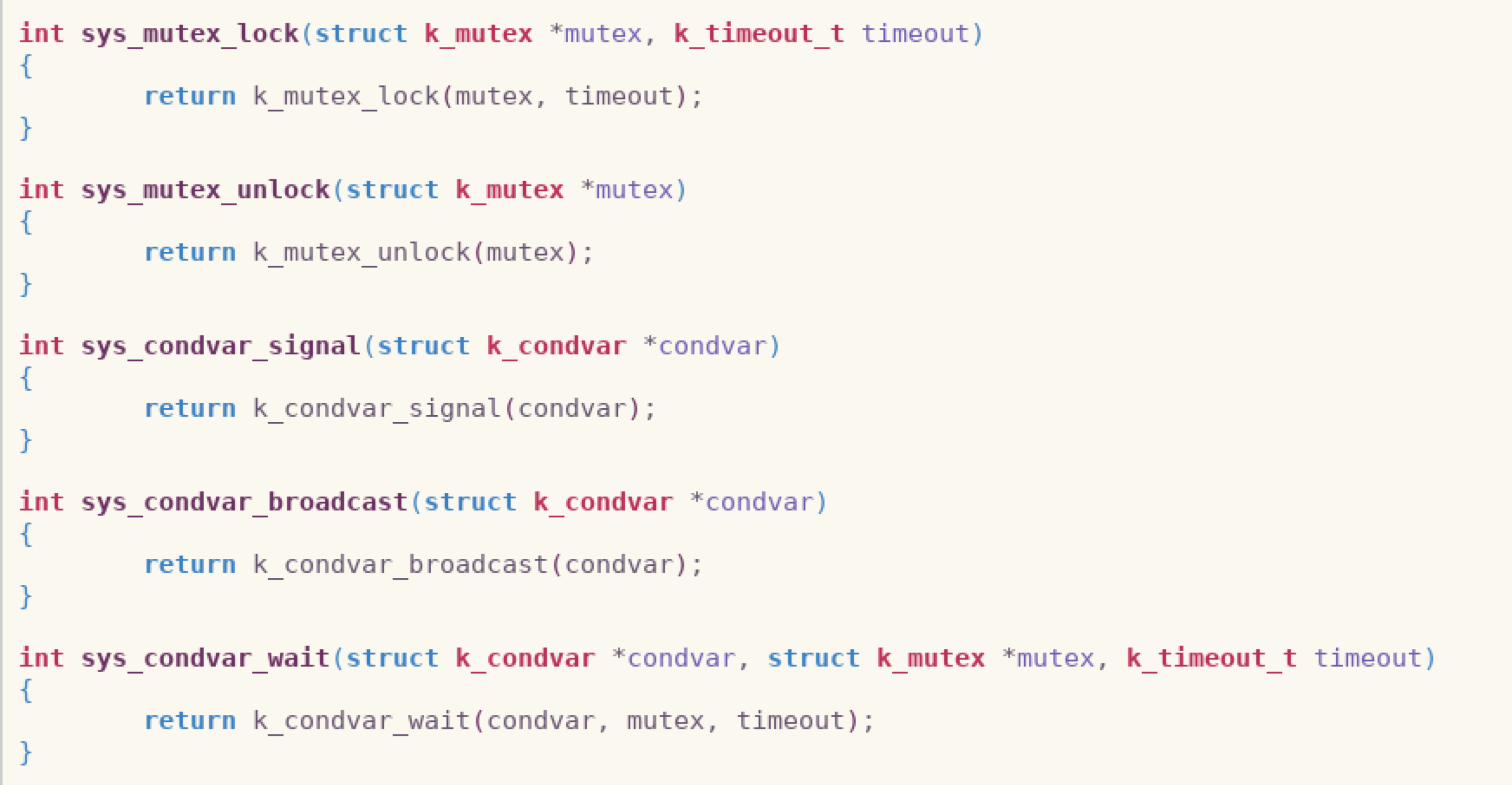
```
}
mod soc {
    pub const Naddress_cells: usize = 1u32;
    pub const Nsize_cells: usize = 1u32;
    pub const fn compatible() -> [&'static str; 2usize] {
        ["raspberrypi,rp2040", "simple-bus"]
    }
    pub static interrupt_parent: &'static str = "[Phandle(\"nvic\")]";
    pub static ranges: &'static str = "[]";
    mod interrupt_controller_e000e100 {
        pub const Naddress_cells: usize = 1u32;
        pub const fn compatible() -> [&'static str; 1usize] {
            ["arm,v6m-nvic"]
        }
        pub static reg: &'static str = "[Number(3758153984), Number(3072)]";
        pub static interrupt_controller: &'static str = "[]";
        pub const Ninterrupt_cells: usize = 2u32;
        pub const arm_num_irq_priority_bits: usize = 2u32;
        pub const phandle: usize = 1u32;
```



```
mod device_tree_labels {
    pub mod adc {
        pub use device_tree::soc::adc_4004c000;
    }
    pub mod adc_default {
        pub use device_tree::pin_controller::adc_default;
    }
    pub mod clk_adc {
        pub use device_tree::clocks::clk_adc;
    }
    pub mod clk_gpout0 {
        pub use device_tree::clocks::clk_gpout0;
    }
    pub mod clk_gpout1 {
        pub use device_tree::clocks::clk_gpout1;
    }
    pub mod clk_gpout2 {
        pub use device_tree::clocks::clk_gpout2;
    }
    pub mod clk_gpout3 {
```

Syscalls

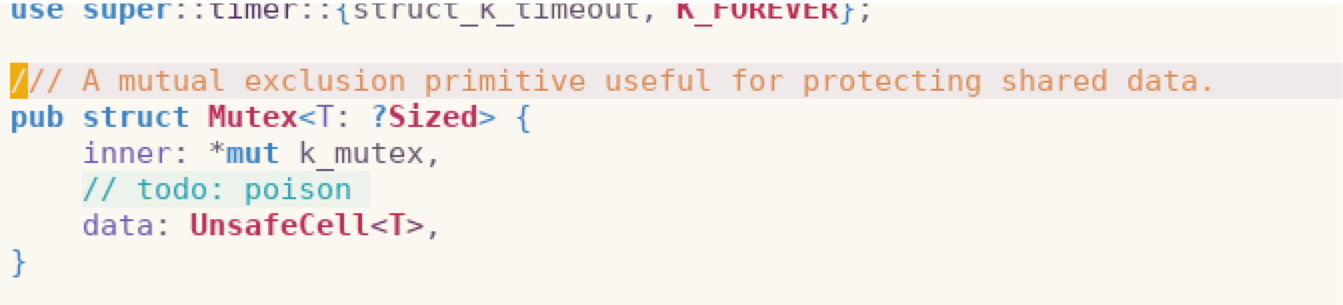
- Marked in C code. Generation depends on features.
- To generate based on C generation.
- Rust syscalls will have similar overhead to C syscalls.
- These are all unsafe in Rust, but available in zephyr-sys for those times they are needed.



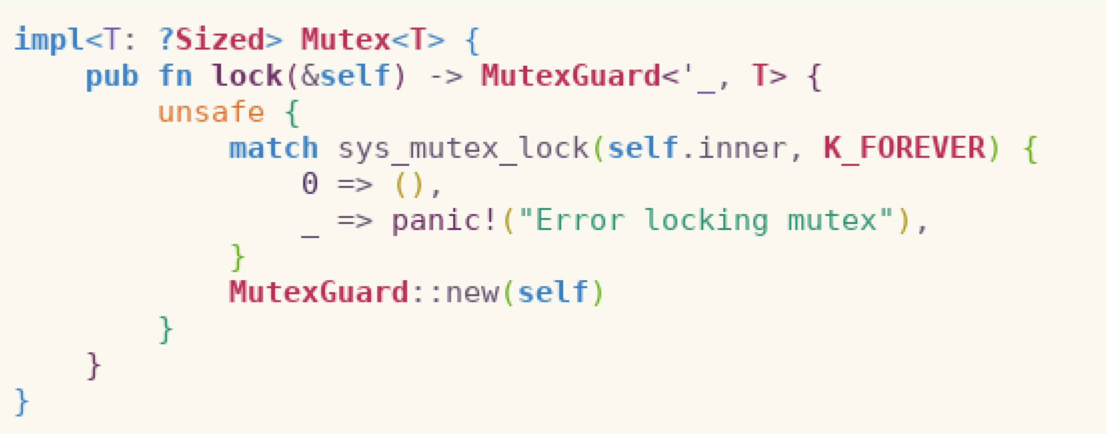
```
int sys_mutex_lock(struct k_mutex *mutex, k_timeout_t timeout)
{
        return k_mutex_lock(mutex, timeout);
}

int sys_mutex_unlock(struct k_mutex *mutex)
{
        return k_mutex_unlock(mutex);
}

int sys_condvar_signal(struct k_condvar *condvar)
{
        return k_condvar_signal(condvar);
}

int sys_condvar_broadcast(struct k_condvar *condvar)
{
        return k_condvar_broadcast(condvar);
}

int sys_condvar_wait(struct k_condvar *condvar, struct k_mutex *mutex, k_timeout_t timeout)
{
        return k_condvar_wait(condvar, mutex, timeout);
}
```

Abstractions

- Build Rust abstractions, Mutex/Condition, Channels, etc.
- Rust apis generally assume underlying primitives can be dynamically allocated.
- Futex and sys_mutex can’t be. Need to resolve, or use a pool.
    - Dynamic kobjects in Zephyr are less efficient.
    - Maybe a primitive specifically for these. Zync?
- Choices: cleaner rust apis, but not compatible with C code, or less safe apis, but compatibility with C. Do we need both?



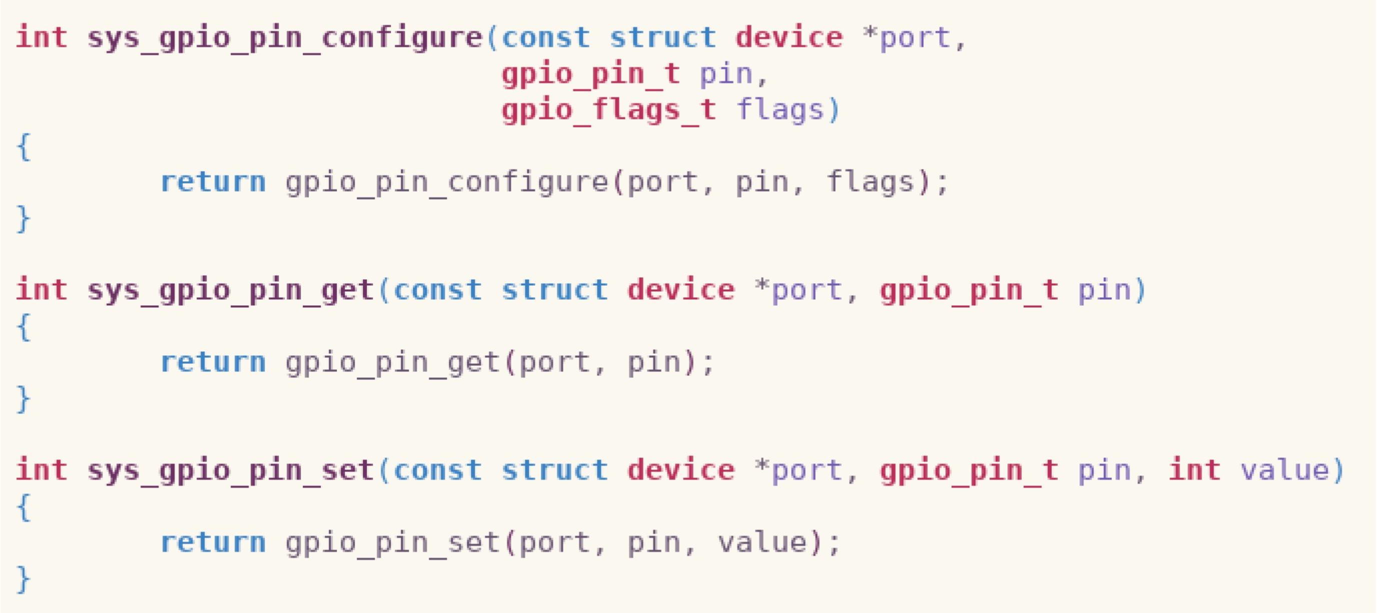
```
use super::timer::{struct_k_timeout, K_FOREVER};

/// A mutual exclusion primitive useful for protecting shared data.
pub struct Mutex<T: ?Sized> {
    inner: *mut k_mutex,
    // todo: poison
    data: UnsafeCell<T>,
}
```



```
impl<T: ?Sized> Mutex<T> {
    pub fn lock(&self) -> MutexGuard<'_, T> {
        unsafe {
            match sys_mutex_lock(self.inner, K_FOREVER) {
                0 => (),
                _ => panic!("Error locking mutex"),
            }
            MutexGuard::new(self)
        }
    }
}
```

Drivers

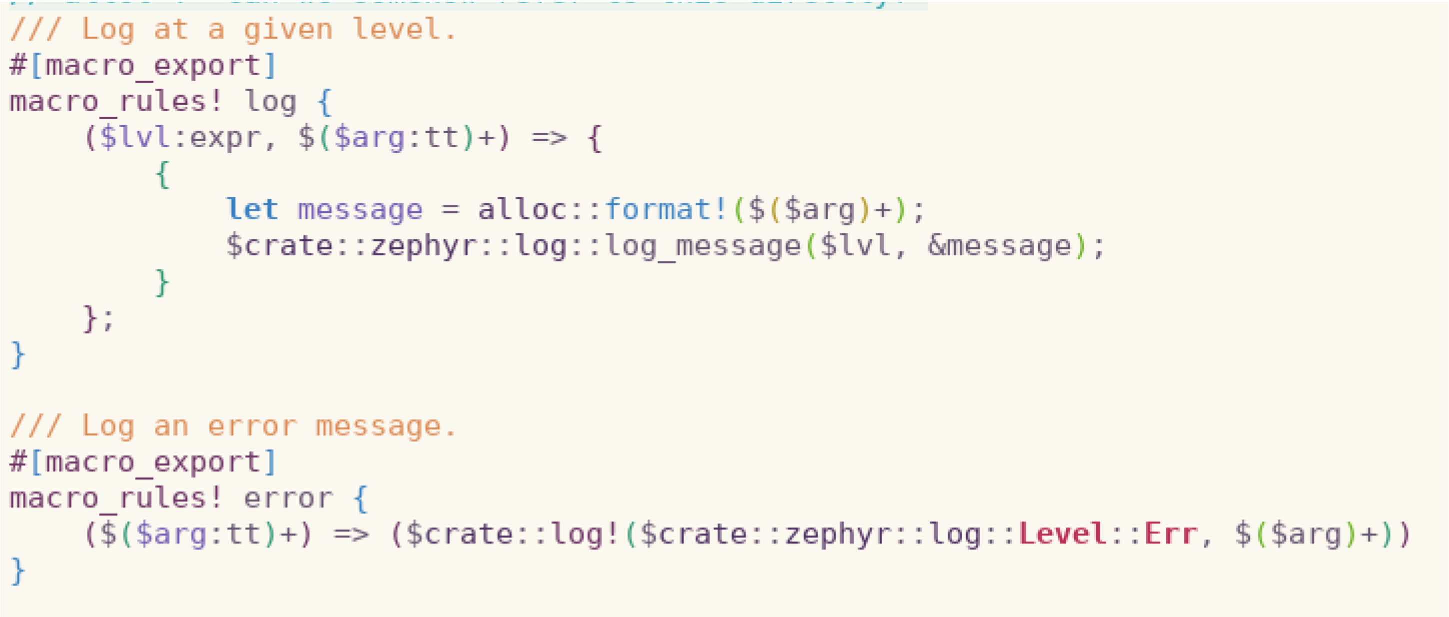
- Integrate with DT generation. The DT should have fns that will return device abstractions to Rust code.
- These will have clean APIs around the underlying Driver.
- Lots of callbacks and such, how much do we try to cleanly wrap these?
- Lots of ad-hoc in Zephyr, probably address case-by-case.



```c
int sys_gpio_pin_configure(const struct device *port,
                           gpio_pin_t pin,
                           gpio_flags_t flags)
{
        return gpio_pin_configure(port, pin, flags);
}

int sys_gpio_pin_get(const struct device *port, gpio_pin_t pin)
{
        return gpio_pin_get(port, pin);
}

int sys_gpio_pin_set(const struct device *port, gpio_pin_t pin, int value)
{
        return gpio_pin_set(port, pin, value);
}
```

Logging

- My code, malloc’ed strings passed to logging. Not very efficient.
- Need to address formatting semantic mismatch.
- Perhaps custom formatter in Zephyr logging to support delayed formatting.
- Rust has defmt, which is host-deferred formatting. Could this be integrated?
- Ideally, low-overhead, and works directly with existing Zephyr logging.



```
/// Log at a given level.
#[macro_export]
macro_rules! log {
    ($lvl:expr, $($arg:tt)+) => {
        {
            let message = alloc::format!($($arg)+);
            $crate::zephyr::log::log_message($lvl, &message);
        }
    };
}

/// Log an error message.
#[macro_export]
macro_rules! error {
    ($($arg:tt)+) => ($crate::log!($crate::zephyr::log::Level::Err, $($arg)+))
}
```

Zephyr® Project

Developer Summit